Föreläsning 6
Datastrukturer (DAT037)

Fredrik Lindblad[1]

2016-11-17

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

Förra gången:

- Leftistheapar
- Mängder/avbildningar
- Hashtabeller
  - Separat kedjning
  - Öppen adressering

Idag: mer om hashtabeller, grafer – definitioner, implementeringar, hitta närmsta väg

Repetition – krav på nyckeltypen

Olika mängd-/avbildningsdatastrukturer har olika krav på nyckeltypen:

- Likhetstest.
- Olikhetstest/komparator.
- Hashfunktion.

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 upptagna; 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, separat kedjning

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

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

Komplexitet, separat kedjning

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

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

Förväntad tidsåtgång

Genomstittsligt antal jämförelser för lyckade sökning som funktion av lastfaktorn:

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

Grafer

Grafer

Grafer kan representera:

- Nätverk.
- Beroenden.
- ...

Grafer

Givet en graf kan man ställa olika frågor:

- Nätverk.
  - Hur tar man sig från A till B? Snabbast? Billigast?
  - Vilken rutt har störst bandbredd?
- Beroenden.
  - Vad måste göras först?
- ...

Terminologi

Terminologi

Varning

Kan variera från författare till författare.

Terminologi

- Graf: $G = (V, E)$.
- V: Ändlig mängd av noder (vertex).
- E: Kanter/bågar (edge).
- Riktad graf: $E \subseteq V \times V$ (ordnade par av noder)
- Oriktad graf: $E \subseteq \{\, U \subseteq V \mid 1 \leq |U| \leq 2 \,\}$ (oordnade par av noder)
- Viktad graf: $E \subseteq V \times V \times W$ eller $E \subseteq \{\, U \subseteq V \mid 1 \leq |U| \leq 2 \,\} \times W$ (där W kan vara $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$, ...).
- I en *multigraf* kan det finnas flera kanter från u till v.

Terminologi

- Direkta efterföljare till u: $\{ v \mid (u, v) \in E \}$.
- Direkta föregångare till v: $\{ u \mid (u, v) \in E \}$.
- Ingrad: Antalet direkta föregångare.
- Utgrad: Antalet direkta efterföljare.

Begreppen definieras på motsvarande sätt för oriktade grafer/multigrafer/viktade grafer.

Terminologi

- Väg (path): $v_1 \to v_2 \to ... \to v_n$.
- Längd: $n - 1$.
- Vägar kan ha längd 0.
- Enkel väg: Alla noder distinkta (utom möjligtvis v_1 och v_n).
- Loop: Kant från nod till sig själv.

Terminologi

För riktade grafer:

- Cykel: Väg av längd ≥ 1 från v till v.
- Enkel cykel: Cykel som är enkel väg.
- (Riktad) acyklisk graf/DAG: Graf utan cykler.

För oriktade grafer:

- (Enkel) cykel:
  Enkel väg av längd ≥ 3 från v till v.

Terminologi

För oriktade grafer:

- Sammanhängande:
  Finns väg från varje nod till varje annan nod.

För riktade grafer:

- Starkt sammanhängande:
  Finns väg från varje nod till varje annan nod.
- Svagt sammanhängande:
  Finns väg från varje nod till varje annan nod,
  om man även får följa kanter baklänges.

Terminologi

Komplett graf:

- Inga loopar.
- I övrigt så många kanter som möjligt.
- Antal bågar:

Terminologi

Komplett graf:

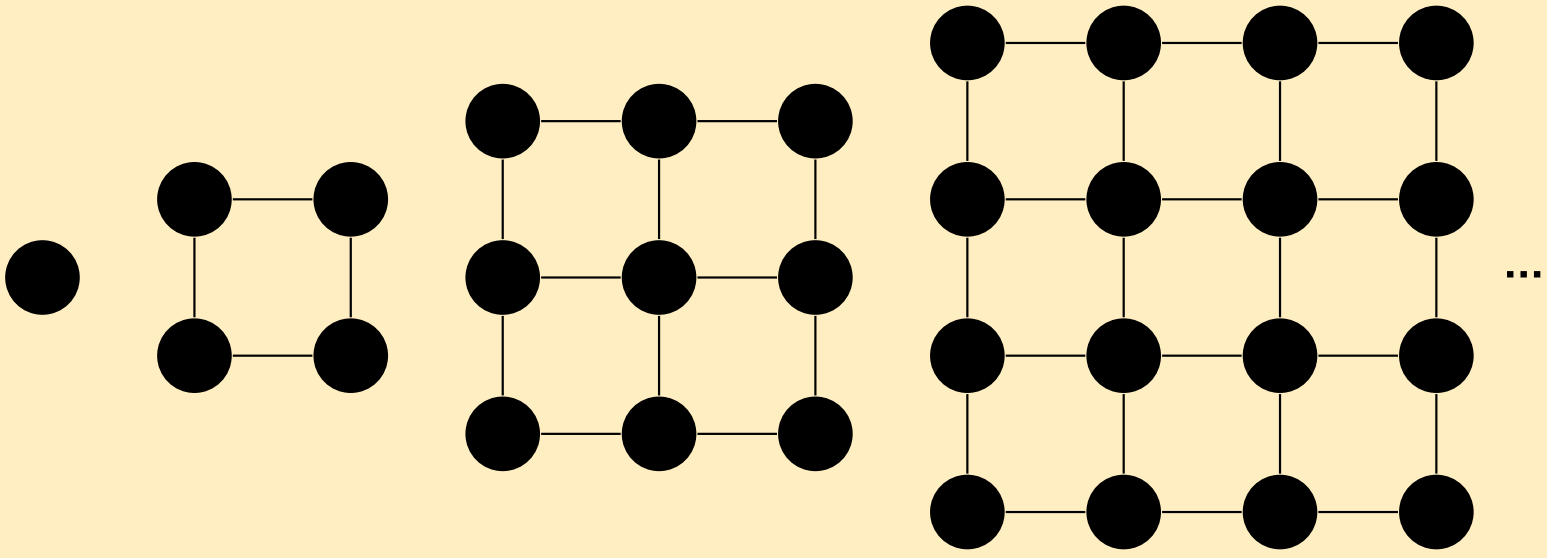
- Inga loopar.
- I övrigt så många kanter som möjligt.
- Antal bågar:
  $|E| = |V|(|V| - 1)/2$

För oriktade, loop-fria grafer i allmänhet gäller
$|E| \leq |V|(|V| - 1)/2$

Terminologi

- Tät graf: Många kanter ($|E| \sim |V|^2$).
- Gles graf: Få kanter ($|E| \lesssim |V|$).

Gäller $|E| = \Theta(|V|^2)$ för följande klass av grafer?



Data-strukturer

Grannmatriser

- Kvadratisk matris med $|V|^2$ element.
- Elementen kan t ex vara `true`/`false`.
- Tar stor plats om grafen är gles.
- Gå igenom en nods direkta efterföljare: $\Theta(|V|)$.
- Gå igenom alla noders direkta efterföljare: $\Theta(|V|^2)$.
- Avgöra om det finns en kant från u till v: $\Theta(1)$.
- Antagande ovan: Känner till noders index.
- Kan använda avbildning (map): nod $\mapsto$ index.

Grannlistor

En variant:

- Array av storlek $|V|$...
- ...innehållandes oordnade listor med direkta efterföljare.
- Ibland också listor med direkta föregångare.
- Gå igenom en nods direkta efterföljare: $O(|V|)$.
- Gå igenom alla noders direkta efterföljare: $\Theta(|V| + |E|)$.
- Avgöra om det finns en kant från u till v: $O(|V|)$.

Grannlistor

En annan variant:

- Ett objekt per nod.
- Avbildning från noder till nodobjekt.
- Objekten innehåller grannlistor med pekare till andra objekt.

Hur stor plats tar en array med grannlistor? (Anta att etiketter/vikter tar liten plats.)

- $\Theta(|V|)$.
- $\Theta(|E|)$.
- $\Theta(|V| + |E|)$.
- $\Theta(|V|^2)$.

Kortaste vägen

Kortaste vägen

Kostnad av väg $v_1, ..., v_n$:

$$\sum_{i=1}^{n-1} c_{i,i+1}$$

I oviktad graf: $n - 1$.

Kortaste vägen

Kortaste vägen-problem:

- Givet två noder u och v,
  hitta en kortaste väg från u till v.
- Givet nod u, för varje nod v,
  hitta en kortaste väg från u till v.
- Hitta kortaste vägen
  från varje nod till varje annan.

Visar sig naturligt att lösa den mittersta varianten ovan.

I algoritmen på nästa slide:

- `d` lagrar kortaste kända vägen till alla noder.
- `p` lagrar föregående nod för kortaste vägen.
- `q` är en kö som bestämmer besöksordningen av noder.

Oviktade grafer: bredden först-sökning

```
d = new array of size |V|, initialised to ∞
p = new array of size |V|, initialised to null
q = new empty queue

q.enqueue(s)
d[s] = 0

while q is non-empty do
  v = q.dequeue()
  for each direct successor v' of v do
    if d[v'] = ∞ then
      d[v'] = d[v] + 1
      p[v'] = v
      q.enqueue(v')

return (d, p)
```

Oviktade grafer: bredden först-sökning

```
d = new array of size |V|, initialised to ∞      O(|V|)
p = new array of size |V|, initialised to null   O(|V|)
q = new empty queue

q.enqueue(s)
d[s] = 0

while q is non-empty do                          O(|V|) ggr
  v = q.dequeue()
  for each direct successor v' of v do           O(|E|) ggr
    if d[v'] = ∞ then
      d[v'] = d[v] + 1
      p[v'] = v
      q.enqueue(v')

return (d, p)
```

Sammanfattning

- Hashtabeller (öppen adressering, storlek, hashfunktion)
- Grafer
- Kortaste vägen