Per Sedholm
DD1320 (tilda12)
2012-09-20

Övning 4

Hashning, sortering, prioritetskö, bästaförstsökning

1. Perfekt hashfunktion

Hitta på en perfekt hashfunktion för atomer. Hur stor blir hashtabellen?

Vi antar

1. inga atombeteckningar är längre än två tecken, dvs tillfälliga namn för syntetiska transuraner som Ununtrium (Uut), Ununquadium (UUq), etc ignoreras
2. algoritmen ska klara alla möjliga tvåteckens-koder (dvs även ”Ad” och ”J” mfl, som inte finns)

Antag att vi låter första bokstaven i namnet (versalen) representeras av hundratal, dvs

$$A = 100, B = 200, \ldots, Z = 2600$$

och ev. andra bokstav av heltalen

$$a = 1, b = 2, c = 3, \ldots, z = 26$$

Då kan vi skapa hashkoden genom att lägga ihop siffrorna: $Ag = 100 + 7$.

Det är dock onödigt att använda hundratal – det finns ju ingen siffra som ger hashkoden 27. Eller 0.

$$A = 0, B = 27, C = 54, \ldots, Z = 675 \qquad (25 \cdot 27)$$

$$a = 1, b = 2, \ldots, z = 26$$

Då blir ”minsta” möjliga atom-hashkod 0, och största möjliga 701:

$$h(A) = 0, \quad h(Ag) = 7\ (0 + 7), \quad h(Zz) = 701\ (675 + 26)$$

Med 702 platser (6 ggr antal element) så blir det inga krockar, och ingen luft förutom det som orsakas av att element inte finns…

I labb 5 ska ni inte göra en så här stor tabell utan istället använda kvadratisk probning.

2. Håll reda på media (*Tildatenta 030308*)

Under gulfkriget var det väldigt svårt för arméstaben att hålla reda på alla TV-bolag som for omkring och rapporterade i öknen. För att hålla reda på dem användes en hashvektor. Koden fungerade inte som avsett och man har nu gett i uppdrag åt en f.d. tildastudent att titta på en misstänkt del av koden:

```
from string import find

p = 100;
hashvektor = [0]*p
alfabet = "abcdefghijklmnopqrstuvxyz"

def put(tvbolagsnamn, tvbolag):
    hashcode = 0
    for i in range(len(tvbolagsnamn)):
        alfanum = find(alfabet, tvbolagsnamn[i])+1
        hashcode += alfanum
    hashcode = hashcode % p
    hashvektor[hashcode] = tvbolag
```

Vad är det för fel på koden? Beskriv hur man kan förbättra den. Namnen på TV-bolagen kan antas bestå av högst tre bokstäver. Det kommer inte att förekomma mer än 75 TV-bolag.

Problem: Saknas krockhantering, och samma bokstäver ger samma kod: *hash("ABC") = hash("CBA")*.

Lösningar:

Krocklistor – varje element i hashvektorn är en lista.

Linjär probning – ifall positionen är upptagen, sök vidare tills man hittar nyckeln*, eller tills man hittar tomt element. (Finns även *kvadratisk probning*.)

En bättre hashkod skulle man få genom att vikta vid uträkningen, ex. multiplicera *alfanum* med 1, 100 och 10 000. (Modulo-operationen gör att man ändå hamnar inom vektorn.)

Större hashvektor vore bättre, och dessutom bör längden vara primtal. (Går att visa matematiskt att det ger bättre spridning.) Lämplig längd är 151, ett primtal dubbelt så stort som förväntat antal element.

3. *Nix* till telefonförsäljning (*Tildatenta 000603*)

Föreningen för konsumentskydd vid marknadsföring per telefon har startat ett register dit den som inte vill bli uppringd av telefonförsäljare kan anmäla sig.

Till att börja kommer kontrollen att ske genom att företaget sänder sin telefonlista till nix *och får tillbaka en lista där de nixade numren markerats.*

Vilka av följande metoder kan föreningen använda sig av? Vilken är bäst?

Binärträd *Bloomfilter* *Hashtabell*

*Notera: *nyckeln*, inte *hashkoden* – den senare är ju redan samma.

Ett framtida mål är att kontroll också skall kunna ske över internet. Då måste kontrollen ske snabbt men man vill också försäkra sig om att ingen ska kunna få ut en lista över alla nixade telefonnummer.

Vilken metod passar bäst för internet-kontrollen?

Just nu ringer många och anmäler sig till NIX-registret så det måste gå lätt att lägga till nya.

Binärträd går snabbt att söka i men man måste se till att det inte blir obalanserat när nya telefonnummer läggs till.

Bloomfilter är besvärligt att stoppa in nya nummer i.

Hashtabell blir inte obalanserad som ett träd, men ifall man får många kollisioner (dålig hashfunktion *eller* inte tillräckligt stor tabell) så blir den mycket oeffektiv.

Bloomfilter är det bästa alternativet för den framtida internet-kontrollen. Man kan räkna med att många har registrerat sig så snabb sökning är nödvändig.

Dessutom har bloomfilter fördelen att ingen användare kan få ut telefonnummerlistan.

(I verkligheten måste man dock även kunna ta bort nummer efter ett antal år. Det gör att Bloomfilter inte är lämpligt ifall man inte kan schemalägga detta, dvs. generera om Bloomfiltret varje dag/vecka/månad. Förmodligen bäst att utgå från en lämpligt stor hashtabell, och generera ”tillfälliga” Bloomfilter för snabb sökning.)

4. Webbtoppen (*Tildatenta 980321*)

Vissa webbsidor räknar hur många besökare dom har eftersom välbesökta webbsidor ger prestige. Du får i uppdrag att skapa webbtoppen, ett program som för varje dag läser av räknarna för tiotusen webbsidor och sedan publicerar dagens tio i topp. Din första tanke är att spara talen i en lista med längd tiotusen, leta fram och skriva ut segraren, nollställa segraren och göra detta tio gånger. Hur många jämförelser skulle krävas för denna algoritm?

Din andra tanke är att spara talen i en trappa (heap) och sedan ta ut och skriva ut tio tal ur trappan. Hur många jämförelser kan det då bli frågan om?

Din tredje tanke är att det borde räcka med en trappa med plats för tio tal. Hur skulle man då göra och hur många jämförelser skulle krävas?

Tio genomletningar av listan tar

$$10 \cdot 10\,000 = 100\,000 \text{ jämförelser}$$

Inmatning i trappan tar cirka $n \log n$, alltså $130\,000$ jämförelser och utplockning cirka $10 \log n$, alltså ytterligare 130 jämförelser.

Det smarta är att ha en tioplatsers min-heap! (Dvs, alla barn-noder är större än rootnoden.)

Överst ligger då det tionde i ordningen av alla tal man sett och det är ju det man ska jämföra varje tal med för att se om det ska in bland dom tio bästa. Med normal tur blir det inte så ofta man ska byta ut tionde talet, så antalet jämförelser blir bara drygt tiotusen.

5. Lönar sig sortering? (*Tildatenta 970404*)

> *En miljon dumbolotter säljs var månad. För varje lott sparas lottnumret och köparen i ett objekt. En lista med en miljon objekt finns alltså i datorn vid dragningen, då tusen vinstnummer slumpas fram, ett efter ett.*
>
> *För varje nummer måste hela listan letas igenom, eftersom den är osorterad. Hur många jämförelser får man räkna med totalt? Lönar det sej att först sortera listan, en gång för alla?*

Ja. I en osorterad lista krävs cirka en halv miljon jämförelser för varje sökning, dvs totalt en halv miljard ($0.5 \cdot 1\,000\,000$ sökningar $\cdot$ $1\,000$ vinstnummer).

Sortering med quicksort kräver cirka $n \log n$ jämförelser, dvs cirka 20 miljoner ($1\,000\,000 \cdot \log_2 1\,000\,000$)

Sedan tar varje binärsökning ($O(\log n)$) bara tjugo jämförelser, dvs tjugotusen totalt (20 jämförelse/vinstnummer $\cdot$ $1\,000$ vinstnummer).

6. Billig standard selection sort (*Tildatenta 960303*)

> *Tilda och Totte skrev var sin sorteringsprocedur. Tilda valde en utsökt merge sort medan Totte tog en standard selection sort. När dom provkörde med tusen objekt gick ändå Tottes program lika fort, eftersom han har superdator. Men med tiotusen objekt vann Tilda. Med hur mycket?*

Selection sort (urvalssortering) är $O(n^2)$, dvs för någon konstant k är tiden t:

$$t = k \cdot n^2$$

$$\left.\begin{array}{rclcrcl} n & = & 10\,000 & \Rightarrow & t_2 & = & k \cdot 10\,000^2 \\ n & = & 1\,000 & \Rightarrow & t_1 & = & k \cdot 1\,000^2 \end{array}\right\} \Rightarrow \frac{t_2}{t_1} = 100$$

Tottes $10\,000$-objektssortering tar alltså 100 gånger så lång tid som $1\,000$-objektssorteringen.

Merge sort är $O(n \cdot \log n)$, dvs $t = k \cdot n \cdot log_2 n$.

$$\left.\begin{array}{rclcrcl} n & = & 10\,000 & \Rightarrow & t_2 & = & k \cdot 10\,000 \cdot \log_2 10\,000 \\ n & = & 1\,000 & \Rightarrow & t_1 & = & k \cdot 1\,000 \cdot \log_2 1\,000 \end{array}\right\} \Rightarrow \frac{t_2}{t_1} \approx \frac{10 \cdot 13}{1 \cdot 10} \approx 13 \qquad (13.333\ldots)$$

$2^{13} = 8\,192 \rightarrow \log_2 10\,000 \approx 13$ (exakt $13.287\ldots$)
$2^{10} = 1\,024 \rightarrow \log_2 1\,000 \approx 10$ (exakt $9.965\ldots$)

Lilla Matterutan

Tildas $10\,000$-objekts sortering tar alltså 13 gånger så lång tid som $1\,000$-objekts sorteringen.

Tildas $10\,000$-objekts sortering är därför $100/13 = 8$ gånger så snabb som Tottes trots den långsammare datorn!

7. Hoppfull sortering

Höjdhoppsfederationens databas över världens alla höjdhoppstävlingsresultat består av objekt med bland annat fälten datum, plats, höjd (cm), hoppare och rivit/klarat. På skivminnet ligger objekten i datumordning, men man vill sortera om dom i resultatordning, nämligen klarade hopp före rivna och höga hopp före låga.

Vilken sorteringsmetod är bäst? Motivera utförligt.

De hopp som finns är grovt sett $100 - 300$ cm, dvs det finns bara några hundra olika höjdvärden (distributioner) i hoppfilen. Antalet registrerade hopp är väldigt många fler än antalet höjdvärden (distributioner) och då är distributionsräkning bästa sorteringsalgoritmen. Tar vi hänsyn till rivit/klarat får vi dubbelt så många distributioner.

Algoritm: Läs igenom filen två gånger, första gången för att räkna hur många hopp det finns av varje rivit/klarat plus höjd. Sedan avsätter man lagom stort segment av listan för varje rivit/klarat plus höjd och vid andra genomläsningen av filen kan varje hopp sättas in på rätt ställe i listan.

8. Tjugondag Knut kastas julen ut (*Tildatenta 010116*)

För att kontrollera sanningen i detta talesätt har man i en fil samlat tre miljoner datum för svenska julgranars utkastning. Man vill veta mediandatum, alltså det datum då hälften av granarna slängts ut, ut, ut och hälften ännu står gröna och granna i stugan.

*Rangordna följande sex föreslagna metoder efter deras effektivitet. Binärsökning, hashning, insättningssortering, distributionsräkning, djupet-först–sökning, trappsortering (*heap sort*).*

Vi vill sortera datumen och plocka ut det mittersta.

Distributionsräkning är bäst ($O(n)$) eftersom det bara finns 365 olika datum. Trappsortering är näst bäst ($O(n \log n)$ och man kan avbryta när hälften sorterats.

Insättningssortering fungerar också ($O(n^2)$).

Hashning är nästan oanvändbart; man bör i så fall vara säker på att hashfunktionen inte kan ge krockar.

Binärsökning och djupet-först–sökning går inte att använda för att hitta medianen.

9. Skatteregistret

Riksskatteverkets databas med nio miljoner svenskar finns sorterad på efternamn. Man vill sortera om den på personnummer. Hur många jämförelser krävs med quicksort? Hur många med den bästa metoden?

Eftersom $n \approx 2^{23}$ $(= 8\ 388\ 608)$ tar quicksort $9\ 000\ 000 \cdot 23 = 207$ miljoner jämförelser.

Radixsortering (gå igenom alla, dela upp i tio buntar efter sista siffran, lägg samman, gör om med näst sista siffran etc) tar $10 \cdot 9\ 000\ 000 = 90$ miljoner. Man kan faktiskt strunta i sista siffran eftersom den är checksiffra. Det finns inte två pnr som bara skiljer sej i den siffran.

10. Båtflytt (*Tildatenta 020406*)

> *Under en seglingstävling vill varje båt hitta den snabbaste vägen till målet. Problemet är att en segelbåt inte kan segla hur som helst och att den seglar olika snabbt beroende på vindriktning och styrka. Antag att havet förenklat består av en massa jämnt fördelade punkter med information om vindstyrka, vindriktning och vilka punkter som finns runt om.*
>
> *Beskriv en algoritm som på ett så effektivt sätt som möjligt tar reda på vilka punkter som ligger utefter den snabbaste seglingsvägen givet en startpunkt och en slutpunkt.*
>
> *Båtägaren är orolig att hans miljövänliga bottenfärg ska nötas bort och vill därför istället ta den väg som är kortast (dvs minst antal steg). Förklara vad som behöver ändras i din föregående algoritm.*

Använd bästaförstsökning med en prioritetskö som prioriterar på lägsta seglade totaltiden. Låt varje nod innehålla total seglingstid samt ha en faderspekare (för rekursiv utskrift av vägen då lösning hittats). Princip för genomgång av problemträdet:

- Lägg startpunktsnoden med totaltiden noll och tom faderspekare i prioritetskön.
- Upprepa så länge kön inte är tom:
  1. Plocka ut en fadersnod ur kön. Om detta är slutpunkten, skriv ut vägen rekursivt och avsluta.
  2. Generera en son i taget genom att för varje punkt runt omkring fadersnoden skapa en son-nod med seglingstiden ökad beroende på vindstyrka, vindriktning och placering i förhållande till fadersnoden. Lägg in son-noden i prioritetskön.

Om dumbarnskoll ska utnyttjas måste det ta hänsyn till både punkten och totala seglingstiden till den punkten. Alla söner med sämre tider till samma punkt är då dumsöner. På det viset slipper algoritmen besöka samma punkt flera gånger – den blir effektivare.

Eventuellt krävs någon snabb uppslagning av punkternas information, t ex med en hashtabell som hashar på punkternas nummer eller position. Noderna kan innehålla lägsta tid som uppnåtts för att tillåta dumbarnkoll utan att kräva extra minne.

För kortaste vägen, använd istället bredden-först med en vanlig kö och blanda inte in totala seglingstiden i varje nod. I detta fall måste vi göra dumbarns-koll för att sökningen ska bli effektiv.

Datastrukturer:

- Prioritetskö / kö
- Hashtabell
- Noder med seglingsdata