NADA, KTH, Henrik Eriksson 2006–01–24

Formelsamling till 2D1344 Grundläggande datalogi

Urvalssortering: Gå igenom `v[1]` till `v[N]` för att finna största elementet `v[i]`, låt sedan `v[i]` och `v[N]` byta plats. Upprepa med N ersatt av $N-1$ osv. Kräver $N^2/2$ jämförelser, både i genomsnitt och i värsta fallet. Elementen flyttas högst en gång var.

Bubbelsortering: Jämför `v[1]` med `v[2]` och låt dom vid behov byta plats, jämför sedan `v[2]` med `v[3]` osv. Nästa varv räcker det att gå till $N-1$. Om inga byten skett under ett helt varv är sorteringen klar. Kräver $N^2/3$ jämförelser i genomsnitt och $N^2/2$ i värsta fallet. Om alla element ligger nästan på rätt plats går det fortare, till exempel $5N$ om inget ska flyttas mer än fyra platser.

Insättningssortering: Antag att `v[1]` till `v[k-1]` är sorterade, Nu sätts `v[k]` in på rätt plats genom att alla mindre element flyttas upp ett steg så att `v[k]` kan sättas in på den tomma platsen. Det här gör man för $k = 2, 3, \ldots, N$ och det kräver cirka $N^2/4$ jämförelser i genomsnitt, $N^2/2$ i värsta fallet. För nästan sorterade vektorer går det fort, till exempel $2N$.

Quicksort: Kalla små element för damer och stora för herrar. Var skiljevärdet ska gå kan bestämmas godtyckligt. Sätt vänster pekfinger på första herren, höger pekfinger på sista damen och byt plats på dom. Fortsätt så tills pekfingrarna korsas – då är vektorn partitionerad. Quicksortera sedan varje partition för sej. Quicksort kräver cirka $N \log N$ jämförelser i genomsnitt men N^2 i värsta fallet. För att undvika värsta fallet bör man välja skiljevärdet vid partitioneringen intelligent.

Mergesort: Dela vektorn i två halvor, mergesortera varje del för sej och samsortera delarna till en ny sorterad vektor. Kräver ungefär $N \log N$ jämförelser i genomsnitt och i värsta fallet. Det går åt extra minnesutrymme lika stort som originalvektorn.

Bottenuppsamsortering: En variant av mergesort där man först sorterar par, samsorterar till 4-grupper, samsorterar till 8-grupper etc. Samma komplexitet.

Trappsortering (heapsort): Vektorn `v` uppfattas som en trappa (se nedan) och sorteras först med hjälp av trappvillkoret. När elementen tas ut från trappan med `heap.get()` kommer dom automatiskt i ordning. Det kräver $2N \log N$ jämförelser i värsta fallet och något mindre i genomsnitt.

Distributionsräkning: Om det bara förekommer M *olika* värden (t ex $M < 1000$) kan man sortera med två genomgångar. Först räknar man bara antal förekomster av varje värde, sedan avsätter man lagom stora segment av en vektor för varje värde och i en andra genomgång sätter man in varje element på sin plats. Kräver $2N$ jämförelser.

Linjär sökning i osorterad vektor: Kräver i genomsnitt $N/2$, i värsta fallet N jämförelser.

Binär sökning i sorterad vektor: Man jämför med mittelementet `v[N/2]` och söker vidare i ena vektorhalvan. Kräver i genomsnitt $\log N - 1$, i värsta fallet $\log N$ jämförelser.

Binär sökning i sökträd: Samma som ovan om trädet är balanserat. Om trädet urartat till en tarm övergår det till linjär sökning.

Sökning i hashtabell: Hashtabellen är en vektor `v[0]..v[M-1]` av pekare, nämligen toppekare på var sin krocklista. En hashfunktion beräknar ett index för söknyckeln key och pushar elementet på den krocklistan. Sökning kräver sedan drygt en jämförelse, beroende på hur många krockar det blivit. Med femtio procents luft (dvs en hashvektor som är dubbelt så stor som antalet element som ska hashas in) blir det nästan inga krockar. Om vektorn är precis lika stor som antalet element som ska hashas in krävs i genomsnitt 1.5 jämförelser.

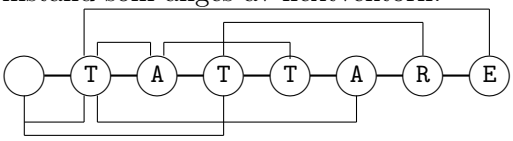
Hashfunktionen brukar vara `f(key) modulo M` där M bör vara ett primtal.

Abstrakta datastrukturer: Den uppsättning anrop som kan göras är för

- Stack: `s.push(x), x=s.pop(), if s.isempty()...` In och ut upptill..
- Kö: `q.put(x), x=q.get(), if q.isempty()...` In baktill, ut framtill.
- Sökträd: `t.put(x,key), x=t.get(key), if t.exists(key)..., t.write()`.
- Hashtabell: `tab.put(x,key), x=tab.get(key), if tab.exists(key)....`
- Trappa (prioritetskö, heap): `h.put(x,key), x=h.get(), if h.isempty()...` Bästa ut.

Trappa (heap) är konkret en vektor uppfattad som ett binärträd med `v[1]` överst, `v[2]` och `v[3]` på nivån under etc. Den understa nivån kan vara delvis fylld (från vänster). Trappvillkoret *fadern bättre än sönerna* innebär att `v[k]` jämförs med `v[2k]` och `v[2k+1]`. Nya element sätts in på understa nivån och när toppelementet tas bort ersätts det av högraste elementet på nedersta nivån. I båda fallen tvingar trappvillkoret fram viss omsortering.

Knuthautomat har ett tillstånd för varje bokstav i sökordet (och ett nolltillstånd). Man tjuvtittar på nästa bokstav i texten; om det är sökbokstaven glufsar man den och går till nästa tillstånd, annars backar man till det tillstånd som anges av nextvektorn.



i	next[i]
1	0
2	1
3	0
4	2
5	1
6	3
7	1

Syntax och rekursiv medåkning:

```
<sats>  ::= <subj><pred> . | <subj><pred> ATT <sats>
<subj>  ::= JAG | DU | ALLA
<pred>  ::= VET | TROR
```

Metoden `readSubj()` glufsar bara i sej ett ord och kollar det. Metoden `readSats()` anropar `readSubj()`, `readPred()`, tjuvtittar på nästa tecken och glufsar eventuell punkt, glufsar och kollar annars ett ord och anropar `readSats()`. Rekursiv medåkning fungerar om man med tjuvtitt kan välja rätt syntaxalternativ.

Datakomprimering: Möjlig komprimering bestäms av entropin. Huffmanträd ger vanliga tecken korta binärkoder, LZW-ordlistor upptäcker återkommande teckenföljder. Bilder komprimeras förlustfritt med GIF. Effektivare men förstörande är JPEG för bilder, MPEG för video och MP-3 för ljud.