Løsningsforslag for utvalgte oppgaver fra kapittel 8

Innhold

8.1–1 Lastfaktorer

Formel: $\alpha = \frac{\text{elementer}}{\text{størrelse}}$

a) $\alpha = \frac{133}{45\,000} \approx 0,003$ b) $\alpha = \frac{35\,000}{45\,000} \approx 0,78$ c) $\alpha = \frac{83\,000}{45\,000} \approx 1,84$

8.1–2 Kompleksitet i kompilatorer

n: Antall *ulike* variabelnavn i koden. (Antall deklarasjoner)
r: Det totale antall variabler som nevnes i koden.

a) Usortert tabell, lineære søk: Nye variabelnavn kan legges inn med $\Theta(1)$ kompleksitet, det er n slike tilfeller. Et lineært søk har $O(n)$ kompleksitet, det er r slike tilfeller. Total kompleksitet: $O(n+nr)$. Her vet vi at $r \geqslant n$ så r kan ikke være null. Dermed kan vi forenkle til $O(nr)$.

b) Sortert tabell og binærsøk: Å legge inn de n variablene får en samlet kompleksitet på $O(n^2)$, da dette tilsvarer én omgang med innsettingssortering. Et oppslag med binærsøk i en tabell med størrelse n har kompleksiteten $O(\log n)$, og det er r slike oppslag. Total kompleksitet blir $O(n^2 + r\log n)$. Dette uttrykket kan ikke forenkles videre, det er usikkert hvilket ledd som har høyest orden.

c) Hashtabell: Å legge inn ett variabelnavn i tabellen har kompleksiteten $\Theta(1)$, og det er n slike tilfeller. Et oppslag har også kompleksiteten $\Theta(1)$, og det er r oppslag. Total kompleksitet blir $\Theta(n+r)$. Siden vi vet at $r \geqslant n$, kan vi forenkle dette til $\Theta(r)$.

Store programmer kan ha tusener av variabler som refereres hundretusener av ganger, så kompleksitet for søke- og innsettingsfunksjonene har en del å si for kjøretiden til kompilatoren.

8.2–1 Beregne hashfunksjoner

Formel: $h(k) = k \bmod 13$

$h(12\,300) = 2$ $h(3) = 3$ $h(75\,776) = 12$ $h(224) = 3$ $h(13) = 0$

8.2–4 Gode og dårlige hashfunksjoner

k & 65 535	Ikke av de beste. Her bruker vi bare 16 bit av nøkkelen, og kaster vekk resten. Hvis nøkkelen stort sett bare varierer i de 16 øverste bitene får vi veldig mye kollisjoner her.
(k>>16)^(k & 65535)	God hashfunksjon som utnytter alle bitene. Vær imidlertid oppmerksom på at xor-funksjonen i blant kan være problematisk.
k>>16	Samme problem som den første hashfunksjonen, bare at her er det bare de øverste bitene fra nøkkelen som brukes. Hvis variasjonen stort sett er i de nederste, får vi et problem.
$k/65\,536$	Dette er akkurat samme hashfunksjon som den forrige, med de samme problemene.
k mod 65521	Primtalldivisjon fungerer generelt bra, men her ligger primtallet farlig nær toerpotensen 65 536. Dermed får vi mye av de samme problemene som med den første hashfunksjonen. Dessuten får vi aldri høyere verdier enn 65 520, så de 16 siste tabellposisjonene blir ikke brukt.

8.2–5 Nøkler som kolliderer

Vær oppmerksom på at det fins mange riktige svar her.

a) $k = 1$ og $k = 65537$

b) $k = 0$ og $k = 65537$

c) Alle $k < 65536$ gir $h(k) = 0 \ldots$

d) Samme som c)

e) $k = 0$ og $k = 65521$

8–1 Gode hashfunksjoner

a) 2048 er en toerpotens, nærmere bestemt 2^{11}. Jeg velger derfor å bruke en hashfunksjon basert på multiplikasjon; $h_1(k) = \lfloor 2048\,(kA - \lfloor kA \rfloor) \rfloor$, og en oddetallsfunksjon for $h_2(k)$. Hvis vi bruker A-verdien vi har sett på i boka, og implementerer dette på en vanlig 32-bits maskin, får vi $h_1(k) = \left\lfloor 2\,654\,435\,769k/2^{32-11} \right\rfloor$ og $h_2(k) = (2|k| + 1) \bmod 2048$

b) 88 741 er et primtall. Da passer det godt å velge en h_1 basert på restdivisjon. F.eks. $h_1(k) = k \bmod 88\,741$. Siden tabellstørrelsen er et primtall kan vi unngå felles faktorer med $h_2(k) = k \bmod (88\,740) + 1$.

c) Også 65 536 er en toerpotens, 2^{16}. Ved å følge samme oppskrift som i oppgave a), får vi $h_1(k) = \left\lfloor 2\,654\,435\,769k/2^{32-16} \right\rfloor = \lfloor 2\,654\,435\,769k/65\,536 \rfloor$ og $h_2(k) = (2|k| + 1) \bmod 65\,536$.

d) 729 er hverken primtall eller toerpotens. Det er dermed ingen funksjoner som peker seg ut. Det er flere muligheter for å velge en brukbar h_1.

Vi kan finne et primtall i nærheten av 729, fortrinnsvis litt større for å nytte hele tabellen, og bruke restdivisjon med primtallet fulgt av restdivisjon med tabellstørrelsen. Den siste restdivisjonen hindrer da at nøkkelen blir større enn tabellstørrelsen. Dette slipper vi hvis vi velger et primtall mindre enn 729, men denne løsningen fører til at noen tabellposisjoner aldri brukes direkte. De kan imidlertid fylles opp av kollisjonstilfeller.

En annen mulighet er en hashfunksjon basert på multiplikasjon. Vi kan ikke bruke den «superraske» implementasjonen som forutsetter at tabellstørrelsen er en toerpotens, fordi 729 ikke er en toerpotens. Hashfunksjoner basert på multiplikasjon fungerer imidlertid fint med alle tabellstørrelser, vi blir bare nødt til å gjøre beregningen med desimaltall.

Forslag: $h_1(k) = \lfloor 729\,(kA - \lfloor kA \rfloor) \rfloor$, der $A = \frac{\sqrt{5}-1}{2}$.

Når det gjelder h_2, må denne aldri bli 0, den kan ikke ha felles faktorer med 729, og bør dessuten variere anderledes enn h_1. Hvis vi faktoriserer 729, ser vi at $729 = 3^6$. Dette er altså en treerpotens. Det er lett nok å lage en funksjon som produserer

verdier som ikke kan deles på 3, det kan gjøres etter samme mal som funksjonen som produserer tall som ikke kan deles på 2. F.eks. $h_2(k) = (3|k|+1) \bmod 729$.

Generelt kan vi bruke denne oppskriften på tabellstørrelser $m = z^a$, der a er et heltall. (Eks.: Hvis størrelsen er $625 = 5 \cdot 5 \cdot 5 \cdot 5$ får vi $z = 5$.) 0, z, $2z$, $3z$, ... er tall vi må unngå. Vi kan da bruke $h_2(k) = (z|k|+1) \bmod m$. En slik hashfunksjon vil alltid virke, men *god* er den bare hvis z viser seg å være tilstrekkelig lav. $\frac{m}{z}$ gir oss hvor mange *ulike* verdier denne hashfunksjonen kan anta, og den bør helst ha mange muligheter for å unngå degenerering til lineær probing. Det er verre hvis z er et høyt primtall.

Denne generelle h_2-funksjonen kan forbedres i tilfeller med høy z. De eneste verdiene vi må unngå er multiplum av z, så langt har vi gjort det ved å legge én til et multiplum av z. Hvis vi i stedet legger til et tall mellom 1 og $z-1$ får vi en hashfunksjon med god spredning også for høye verdier av z. En funksjon som gir oss verdier mellom 1 og $z-1$ er $k \bmod (z-1)+1$. En god generell hashfunksjon blir da $h_2(k) = (z|k|+k \bmod (z-1)+1) \bmod m$. Legg merke til at hvis m er en toerpotens degenererer denne funksjonen til den anbefalte funksjonen $(2|k|+1) \bmod m$, og hvis $z = m$ degenererer den til den anbefalte funksjonen $k \bmod (m-1)+1$. Den passer altså i alle slike tilfeller. Bruker vi denne oppskriften på tilfellet $m = 729 = 3^6$ får vi $h_2(k) = (3|k|+k \bmod 2+1) \bmod 729$.

8–2 Tabellstørrelse med åpen adressering

En hashtabell basert på åpen adressering kan med fordel lages litt større enn det er behov for, for å unngå store mengder kollisjoner i det tabellen fylles helt opp.

Hvis vi setter inn det aller siste elementet det er plass til, vil vi gjennomsnittlig oppleve $\frac{m}{2}$ kollisjoner. Det kan jo bli mye, hvis m er stor. En hashtabell som lages litt større, slik at den bare blir 90% full, vil i gjennomsnitt oppleve $\frac{1}{1-\alpha} = \frac{1}{1-\frac{90}{100}} = 10$ kollisjoner når det siste elementet settes inn, uansett hvor stor m måtte være. Hashtabeller brukes ofte for å spare tid, og da vil det ofte lønne seg å ofre litt ekstra plass for å få opp farten.