

Tentamen  
Datastrukturer D  
DAT 035/INN960  
(med *mycket* kortfattade lösningsförslag)

21 december 2007

- Tid: 8.30 - 12.30
- Ansvarig: Peter Dybjer, tel 7721035 eller 405836
- Max poäng på tentamen: 60. (Bonuspoäng från övningarna tillkommer.)
- Betygsgränser, CTH: 3 = 30 p, 4 = 40 p, 5 = 50 p, GU: G = 30 p, VG = 50 p.
- Hjälpmedel: *handskrivna* anteckningar på *ett* A4-blad. Man får skriva på båda sidorna och texten måste kunna läsas utan förstoringsglas. Anteckningar som inte uppfyller detta krav kommer att beslagtas!  
Föreläsningsanteckningar om datastrukturer i Haskell, av Bror Bjerner
- Skriv tydligt och disponera papperet på ett lämpligt sätt.
- Börja varje ny uppgift på nytt blad.
- Skriv endast på en sida av papperet.
- Läs noga genom en uppgift innan du svarar på den!
- **Kom ihåg:** alla svar ska motiveras väl!
- Poängavdrag kan ges för onödigt långa, komplicerade eller ostrukturerade lösningar.
- Lycka till!

1. Betrakta det lilla lexikonet  $\{(C, 1), (A, 2), (A, 3)(B, 4)\}$  som har tecken  $A, B, C, \dots$  som söknycklar och heltal som värden. I denna uppgift ska du sätta in elementen i ett antal olika datastrukturer och rita en bild på den resulterande strukturen. Elementen ska sättas in ett efter ett (i den ovan angivna ordningen) med respektive standardalgoritmer för insättning.

- (a) Rita en bild på detta lexikon lagrat som en enkellänkad sorterad lista! Tecknen sorteras i bokstavsordning. (2p)

**Svar.** Se bild på enkellänkad lista i boken. Notera att listan ska vara sorterad samt måste innehålla båda kopiorna  $(A, 2), (A, 3)$ .

- (b) Rita en bild på det binära sökträd du får om du sätter in elementen ett efter ett med hjälp av standardalgoritmen för insättning i binära sökträd. Implementeringen av sökträdet ska vara som en *länkad* struktur (alltså inte som ett fält). (2p)

**Svar.** Se boken eller föreläsningssanteckningarna för bild på hur man implementerar binära träd och hur standardalgoritmen för insättning i binära sökträd fungerar. Notera att även här måste båda kopiorna  $(A, 2), (A, 3)$  lagras.

- (c) Rita en bild på den skipplista du får om du använder standardalgoritmen för insättning i en skipplista! Skipplistan ska inte ha någon extra kopia av  $(C, 1)$  och  $(A, 2)$  men en extra kopia av  $(A, 3)$  och  $(B, 4)$ . (2p)

**Svar.** Se boken. Även här ska dubletterna lagras.

- (d) Rita en bild på den hashtabell du får om du använder standardalgoritmen för insättning i en hashtabell som använder "open addressing with linear probing". Hashfunktionen avbildar  $A$  på 1,  $B$  på 2,  $C$  på 3, osv. Hashtabellen är ett fält med storleken 5. (2p)

**Svar.** Notera att man lagrar *pekare* till objekt innehållande paren  $\{(C, 1), (A, 2), (A, 3)(B, 4)\}$  i fältet som implementerar hashtabellen. Notera igen att dubletter ska lagras och att linear probing metoden för att finna nästa lediga cell ska användas.

För att få full poäng ska du rita ALLA minnesceller och pekare korrekt! I vissa av strukturerna ovan har du en viss frihet att göra olika implementeringsval. Förklara vilket val du gjort!

2. Vilka av följande påståenden är korrekta och vilka är felaktiga? Diskutera! Det kan vara viktigare att belysa frågan på ett allsidigt sätt än att ge rätt svar.

- (a) Om  $n > 7$  så gäller att  $n > 4 \log_2 n$ . (2p)

**Svar.** Nej, om  $n = 8$  så är  $4 \log_2 n = 12$ .

- (b) Antag att du använder en hashtabell med hinkar ("hashing in buckets" eller "chaining"). Hashtabellen innehåller  $n$  element som lagras i  $N$  hinkar (dvs hashtabellen implementeras som ett fält med storleken  $N$ ). Operationen att sätta in ett element i hashtabellen tar då  $O(n)$  i värsta fall. (2p)

**Svar.** Nej, den tar  $O(1)$  eftersom man bara sätter in elementet i början av på den länkade lista som implementerar hinken.

- (c) Antag att du i stället använder en hashtabell med öppen adressering ("open addressing with linear probing"). Hashtabellen innehåller även här  $n$  element och implementeras som ett fält med storleken  $N$ . Då tar det  $O(n)$  i värsta fall att hitta det *minsta* elementet. (2p)

**Svar.** Nej, man måste leta genom hela hashtabellen. Detta tar  $O(N + n)$  om storleken på hashtabellen är  $N$ .

- (d) Antag att du har en graf som är implementerad som en grannmatris. Operationen att lägga till en ny bäge tar  $O(n)$  i värsta fall om grafen har  $n$  noder. (2p)

**Svar.** Nej, den tar  $O(1)$ .

- (e) Antag att du har en riktad graf som är implementerad som grannlistor (en länkad lista av länkade listor). Operationen att lägga till en ny bäge tar  $O(n)$  i värsta fall om grafen har  $n$  noder. (2p)

**Svar.** Ja, i värsta fall ligger källnoden sist.

3. (a) Antag att du använder quicksort-algoritmen för att sortera ett fält med 4 element. Hur många jämförelser mellan elementen kommer att utföras i värsta fall och i bästa fall? Motivera. (3p)

**Svar.** Quicksort gör i bästa fall  $3 + 1 = 4$  och i värsta fall  $3 + 2 + 1 = 6$  jämförelser. Värsta fallet uppstår om man varje gång väljer största eller minsta elementet som pivotelement.

- (b) Samma fråga för mergesort! (3p)

**Svar.** Mergesort gör alltid 1 jämförelse vid merge av listor med ett element. Vid merge av två listor med två element gör den i bästa fall 2 jämförelser och i värsta fall 3 jämförelser. Totalt blir det alltså  $1+1+2 = 4$  jämförelser i bästa fall och  $1 + 1 + 3 = 5$  jämförelser i värsta fall.

4. Antag att du har följande gränssnitt för stackar i Java:

```
interface Stack<E> {  
    void push(E elem);  
    E pop();  
    E top();  
    boolean isEmpty();  
}
```

- (a) Vilken tidskomplexitet får de olika operationerna om du implementerar stacken med en enkellänkad lista? Du ska uttrycka komplexiteten med hjälp av  $O$ -notation och som funktion av antalet element  $n$  som ligger i stacken. (2p)

**Svar.** Alla operationerna är  $O(1)$ ,

- (b) Implementera en generisk metod i Java som beräknar hur många element stacken innehåller:

```
<E>int size(Stack<E> s){ ... }
```

Observera att du bara har tillgång till metoderna i gränssnittet, inte till någon klass som implementerar dem! (4p)

**Svar.**

```
int counter = 0;
while (!s.isEmpty()) {
    counter++;
    s.pop();
}
return counter;
```

Obs att denna metod förstör stacken!

- (c) Vilken  $O$ -komplexitet har den **size**-metod du skrev i (b), under förutsättning att stacken är implementerad som en enkellänkad lista enligt (a)? Du ska uttrycka  $O$ -komplexiteten som funktion av antalet element  $n$  i stacken. (2p)

**Svar.**  $O(n)$ .

- (d) Ofta innehåller ett gränssnitt för stackar även en metod

```
int size();
```

som implementeras så att dess komplexitet blir  $O(1)$ . Är det alltid en fördel att ha en sådan primitiv **size**-metod? Enligt (b) ovan kan man ju ändå alltid räkna ut storleken på stacken med hjälp av de andra stack-operationerna. Diskutera effektivitetsmässiga för- och nackdelar i olika situationer. (2p)

**Svar.** Nej, om man har en primitiv **size**-metod som har komplexitet  $O(1)$  måste man hela tiden uppdatera en tillståndsvariabel som håller reda på stackens storlek. Denna kostnad måste vägas mot fördelen av att ha en effektiv **size**-metod.

5. Om  $G$  är en riktad graf är  $G^*$  dess *transitiva hölje*.  $G^*$  har

- samma *noder* som  $G$ ;
- det finns en *båge* från  $u$  till  $v$  i  $G^*$  om och endast om det finns en *väg* från  $u$  till  $v$  i  $G$ .

Betrakta följande Java-metod

```
void f(boolean[][] G) {
    int n = G.length;
    for(int i = 0; i < n; i++){
        for(int j = 0; j < n; j++){
            for(int k = 0; k < n; k++){
                G[i][j] = G[i][j] || (G[i][k] && G[k][j]);
            }
        }
    }
}
```

```

    }
    return;
}

```

Metodens argument  $G$  är en grannmatris som representerar en riktad graf,

- (a) Vilken  $O$ -komplexitet har metoden  $f$  uttryckt som funktion av antalet noder  $n$  i grafen? (3p)

**Svar.** Komplexiteten är  $O(n^3)$ . Satserna

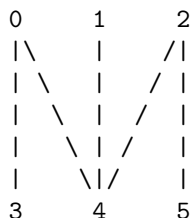
$$G[i][j] = G[i][j] || (G[i][k] \&\& G[k][j]); G(i)[j]$$

och  $k++$  tar  $O(1)$  och utförs  $n^3$  gånger. Satsen  $k < n$  utförs  $n^2(n+1)$  gånger och tar också  $O(1)$ .

- (b) Är det sant att metoden  $f$  beräknar transitiva höljet av indatagrafen? Dvs om grannmatrisen representerar  $G$  före exekveringen så representerar den  $G^*$  efter exekveringen? Om svaret är nej ska du lokalisera felet och korrigera det och om svaret är ja ska du motivera varför! (5p)

**Svar.** Detta är nästan Warshalls algoritmen, men for-slingorna är omkastade. Efter exekveringen kommer  $G[i][j]$  att innehålla en båge om det finns en väg från nod  $i$  till nod  $j$  som har längden 1 eller 2 i indatagrafen. Däremot kommer inte vägar av längd 3 att komma med. Algoritmen korrigeras lämpligen genom att flytta den innersta for-slingan så att den kommer ytterst.

6. En oriktad graf är *bipartit* om noderna kan delas in i två disjunkta delmängder  $V_0$  och  $V_1$  så att alla bågar har ena ändpunkten i  $V_0$  och den andra i  $V_1$ . Här är ett exempel på en bipartit graf där  $V_0 = \{0, 1, 2\}$  och  $V_1 = \{3, 4, 5\}$ :



- (a) Skriv en effektiv algoritm i pseudokod som tar en graf som indata och som returnerar *true* om grafen är bipartit och *false* annars. Du ska även beskriva vilken datastruktur du använder för att implementera grafen. (7p)

**Svar.** Använd grannlistor. Gör en bredden-först sökning som i boken, men där noder på jämnt djup markeras med  $V_0$  och noder på udda djup markeras med  $V_1$ . När man besöker en ny nod kollar man att alla dess grannar tillhör motsatt  $V_i$ . (Detta är samma sak som att kontrollera att det inte finns några tvärbågar/"cross edges".) Obs att grafen kan vara osammanhängande och att vi då behöver göra upprepade bredden-först sökningar.

- (b) Vilken  $O$ -komplexitet har din algoritm uttryckt i antalet noder  $n$  och antalet bågar  $m$  i grafen? (3p)

**Svar.** Algoritmen tar  $O(m + n)$ . Det är en variant av bredden-först sökning.

Poängsättningen kommer att bero på hur bra din pseudokod är, hur effektiv algoritmen är och hur väl motiverad tidsanalysen är.

7. (a) Definiera en Haskell-typ

`AdjacencyListsGraph a`

av riktade grafer representerade som grannlistor! Noderna i grafen tillhör en godtycklig typ `a` och det finns ingen information i bågarna. (2p)

**Svar.**

`type AdjacencyListsGraph a = [(a,[a])]`

- (b) Skriv sedan en Haskellfunktion

`areAdjacent :: Eq a => AdjacencyListsGraph a -> a -> a -> Bool`

så att

`areAdjacent g u v`

returnerar `True` om det finns en båge från `u` till `v` i grafen `g` och returnerar `False` annars. (4p)

**Svar.**

`areAdjacent [] source target = False`

`areAdjacent (u,us):uss source target  
 | source == u = elem target us  
 | otherwise = areAdjacent uss source target`

- (c) Vilken  $O$ -komplexitet har din funktion uttryckt i termer av antalet noder  $n$  och antalet bågar  $m$ ? (2p)

**Svar.**  $O(n + m)$ . Funktionen `areAdjacent` anropas  $O(n)$  gånger och beräkningen av `elem` är  $O(m)$ . (Om man vet maximala gradtalet  $d$  för en nod är komplexiteten av `elem` är  $O(d)$  och alltså blir den totala komplexiteten i så fall  $O(n + d)$ .)

Alternativt kan du lösa uppgiften i Java. För att få full poäng i (a) ska du i så fall implementera en Java-klass

`class AdjacencyListsGraph<V> { ... }`

och i (b) ska du implementera en metod i denna klass:

`boolean areAdjacent(V u, V v) { ... }`

Du får förutsätta att du redan har en klass med länkade listor eller dylikt och behöver inte implementera metoderna i denna. Deluppgift (c) ska förstås också göras om du löser uppgiften i Java.

Pseudokodslösningar ger dock endast delpoäng.