Tentamen
Datastrukturer D
DAT 035/INN960

22 december 2006

- 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!
- Skriv tydligt och disponera papperet på ett lämpligt sätt.
- Börja varje ny uppgift på nytt blad.
- Skriv endast på en sida av papperet.
- **Kom ihåg:** alla svar ska motiveras väl!
- Poängavdrag kan ges för onödigt långa, komplicerade eller ostrukturerade lösningar.
- Lycka till!

1. 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. När O-komplexiteter för relevanta operationer är betydelsefull för frågans svar ska de anges.

   (a) Borttagning av *minsta* elementet i en heap tar $O(1)$ i värsta fall. (2p)
   Svar. Nej, det tar $O(\log n)$.

   (b) Borttagning av *minsta* elementet i ett AVL-träd tar $O(n)$ i värsta fall. (2p)
   Svar. Nej, det tar $O(\log n)$ att söka upp det och ta bort det.

   (c) Splayträd är en mycket effektiv datastruktur för lagring av prioritetsköer. (2p)
   Svar. Nej, en mycket viktig prioritetsköoperation är att söka upp minsta elementet och denna operation måste vara effektiv. För datastrukturer som är särskilt lämpade för prioritetsköer, som heapar, kan detta göras på tiden $O(1)$. Att hitta minsta elementet i ett splayträd kan inte göras på konstant tid.

   (d) Det är alltid bättre att använda hashtabeller än balanserade sökträd (t ex AVL-träd, rödsvarta träd, B-träd) som lagringsmedium för avbildningar ("maps"). (4p)
   Svar. Nej, inte i alltid, men i allmänhet. I hashtabeller tar sökning, insättning och borttagning i praktiken $O(1)$ om man har en bra hashkodning. I vissa situationer, t ex vid lagring av data på externminne eller om man dessutom ofta söker efter minsta elementet kan dock balanserade sökträd av olika slag vara att föredra.

2. Grannmatriser kan användas för att representera grafer. De kan implementeras antingen med hjälp av *vanliga fält* som har en viss storlek eller med hjälp av *dynamiska fält* som kan utvidgas vid behov.

   (a) När är det lämpligt att använda den ena implementeringsmetoden och när är det lämpligt att använda den andra? (2p)
   Svar. Man bör använda dynamiska fält om man inte på förhand vet hur stora grafer man behöver hantera. Annars kan man använda vanliga fält. Om man använder dynamiska fält blir insättning av ny nod $O(n)$ i värsta fall medan för fixa fält blir den $O(1)$. Notera dock att insättning för fixa fält kan resultera i "overflow".

   (b) Förklara varför det är intressant att diskutera den *amorterade* komplexiteten i det här sammanhanget! (3p)
   Svar. Av samma anledning som den amorterade komplexiteten är intressant för insättning i dynamiska fält, dvs att även om komplexiteten för en insättning är $O(n)$ i värsta fallet, så tar *en följd* av n insättningar $O(1)$ i medeltal även i värsta fallet.

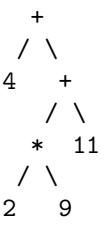
3. Antag att du vill lagra epostmeddelanden i en hashtabell, så att man snabbt kan leta upp ett meddelande från en viss avsändare en given dag. Konstruera en bra hashfunktion för uppgiften!

Svar. Använd polynomiell hashkodning av avsändarens namn (en sträng av tecken). Detta ger ett heltal. Addera sedan datum betraktat som ett sexsiffrigt heltal. (5p)

4. Betrakta aritmetiska uttryck som består av positiva heltal, operationerna + och * och parenteser. Mer precist säger vi att antingen är ett aritmetiskt uttryck ett positivt heltal n eller så har det någon av formerna $(e + e')$ eller $(e * e')$, där e och e' redan är aritmetiska uttryck. (Notera att vi alltid sätter ut parenteser kring en summa eller produkt.) Exempel på aritmetiska uttryck är $7, 12, (3 + 4), (5 * 8), ((2 * 9) + 11)$, osv.

   (a) Aritmetiska uttryck representeras lämpligen i datorn som äkta binära träd med operationerna + och * i de inre noderna och tal i löven. Rita det träd som representerar uttrycket $(4 + ((2 * 9) + 11))$! (2p)

   Svar.



   (b) Skriv en Javaklass som lagrar äkta binära träd som representerar aritmetiska uttryck enligt ovan. (Det räcker med att ange tillståndsvariablerna i de klasser du använder.) (4p)

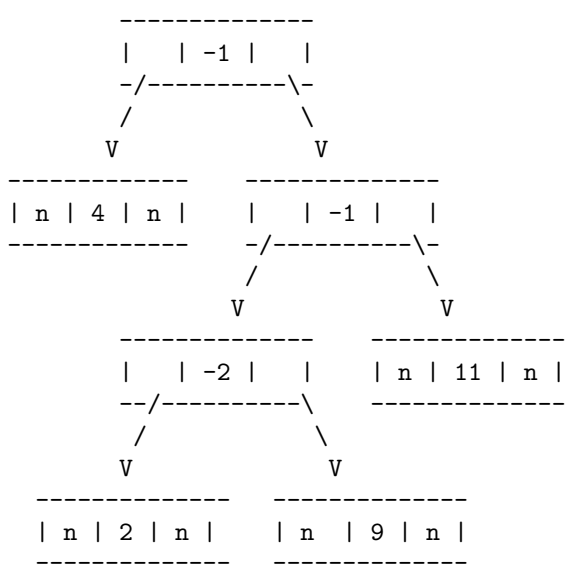
   Svar. Låt -1 representera + och -2 representera *. Sedan kan du använda ett vanligt binärt träd med heltal i noderna för att representera uttrycket. T ex:

```
public class ExpNode {
   int PLUS = -1;
   int TIMES = -2;
   ExpNode left;
   int c;
   ExpNode right;
}

public class ExpTree {
   ExpNode root;
}
```

(c) Rita sedan hur uttrycket i (a) lagras i minnet som ett objekt som tillhör *din* Javaklass i (b). Du ska rita ut *alla* minnesceller som används och även deras innehåll. (2p)

Svar. Vi ritar en pekarstruktur där `n` står för en nollpekare.



(d) Skriv en metod

```
public int valueOf()
```

som returnerar värdet av trädet som ligger lagrat i din Javaklass! Om `valueOf()` anropas från det objekt som lagrar trädet $(4+((2*9)+11))$ ska det alltså returnera 33. (Avdrag kommer inte att ges för smärre syntaxfel i Java. Om du skriver i Java-liknande pseudokod är det dock viktigt att den är så detaljerad och lik Java som möjligt.) (4p)

Svar. Här är ett en enkel metod i klassen `ExpNode` som räknar ut värdet av delträdet som har denna nod som rot.

```
int valueOf() {
    if (c > 0) return c;
    if (c == PLUS) return left.valueOf() + right.valueOf();
    if (c == TIMES) return right.valueOf() * right.valueOf();
    }
```

Ett riktigt program bör förstås göra felhantering också, men vi krävde inte detta uttryckligen i uppgiften. Man bör kontrollera att noder med + och * verkligen har barn med väldefinierade värden.

5. I kursen använde vi skiplistor för att implementera avbildningar och lexika ("maps" och "dictionaries").

(a) Man kan också använda skiplistor för att sortera. Ge pseudokod för en sorteringsalgoritm som använder sig av skiplistor! (Pseudokoden behöver inte vara detaljerad men de viktiga stegen i algoritmen måste framgå klart.) Algoritmen ska ta ett fält med heltal som indata och returnera ett sorterat fält som utdata.

Vilken O-komplexitet har din algoritm i värsta fallet och medelfallet? (5p)

Svar. Om antalet element i indatalistan lista är n, så gör man helt enkelt först n insättningar av elementen i skiplistan. Eftersom skiplistans element är sorterade kan man avslutningsvis bara läsa av elementen ett efter ett i skiplistan.

Varje insättning i skiplistan tar $O(\log n)$ och totalt tar alltså alla insöttningarna $O(n \log n)$. Utplockningen av elementen tar $O(n)$. Alltså tar algoritmen totalt $O(n \log n)$ i medeltal. I värsta fall tar en insättning $O(n)$ och alltså tar hela sorteringen $O(n^2)$ i värsta fall.

(b) Ytterligare en tänkbar användning av skiplistor är för att implementera mängder. Hur gör man om man vill implementera unionen av två mängder med skiplistor? Vilken komplexitet har din algoritm? Är det en bra eller dålig implementering i jämförelse med de alternativ som finns? (5p)

Svar. Man kan sammanfläta skiplistorna med "merge"-operationen på i princip samma sätt som man gör när man tar unionen av vanliga sorterade listor genom att sammanfläta dem. Mergeoperationen är dock något mer komplicerad för en skiplista eftersom vi måste hålla reda på flera kopior av ett element. Om ett element finns i båda skiplistorna måste vi bestämma hur många kopior av detta som ska finnas i unionskiplistan. Det enklaste är här att slumpmässigt välja samma multiplicitet som det i den ena eller den andra skiplistan som vi började med.

Om den ena skiplistan innehåller m element och den andra listan innehåller n element så är komplexiteten för mergeoperationen $O(m + n)$ både för skiplistor och vanliga listor. Anledningen är att i medeltal har ett element i en skiplista en kopia och alltså ökar bara tidsåtgången med en konstant faktor.

Det är bra att använda skiplistor för att implementera mängder. De har bättre komplexitet än vanliga sorterade listor för sökning, insättning och borttagning och samma $O(m + n)$-komplexitet för union, snitt mm.

6. (a) I spelprogram representeras ofta spel som träd eller grafer. Varje ställning representeras som en nod, och varje drag representeras som en riktad båge från ställningen innan draget till ställningen efter draget. En del spel har oändligt många ställningar och kommer på så sätt att representeras av *oändliga* grafer. Ett exempel är *luffarschack* som spelas på ett obegränsat stort rutnät. (Spelarna turas om att markera rutor med "o" respektive "x". Den spelare vinner som först får fem rutor i rad, vågrätt, lodrätt, eller diagonalt.)

Djupet-först och bredden-först är två populära metoder för att söka i grafer. Djupet-först är oftast den mer effektiva metoden av de två. Den har

dock en viktig nackdel när den används på oändliga grafer vars noder har ändlig grad (ändligt många grannar). I så fall finns situationer då djupet-först sökning inte kommer att genomsöka alla de noder som bredden-först sökning gör. Ge ett exempel på en sådan situation! (Om du vill kan du använda den oändliga grafen i deluppgift (b), men du får gärna konstruera ett eget exempel.) (3p)

Svar. Betrakta grafen i (b). Djupet-först sökning kommer att besöka noderna 1,2,4,8,... men inte nod 3, 5, 6, 7. Bredden-först sökning kommer att besöka alla noderna.

(b) *Iterativ fördjupning* är en intressant metod som försöker kombinera fördelarna hos djupet-först och bredden-först sökning. Iterativ fördjupning består i att göra upprepade djupet-först sökningar. Första gången söker man till djupet 1, andra gången till djupet 2, osv. (Med djup menar vi här avstånd från startnoden.) På så sätt kommer metoden att finna samma noder som bredden-först sökning.

Betrakta den oändliga graf du får genom att ha en nod för varje positivt heltal och bågar mellan talen n och $2n$ och mellan n och $2n+1$ för alla n. Räkna upp de *sju* första noder som besöks för djupet-först sökning, bredden-först sökning, och iterativ fördjupning om du börjar med noden med talet 1! (Observera att iterativ fördjupning besöker samma nod flera gånger och att du ska räkna upp den varje gång den besöks.) (3p)

Svar. Djupet-först: 1,2,4,8,16.32,64.
Bredden-först: 1,2,3,4,5,6,7.
Iterativ fördjupning: 1,2,3,1,2,4,5.

(c) Slutligen ska du analysera den asymptotiska komplexiteten hos iterativ fördjupning i *ändliga* grafer. Gäller det alltid att iterativ fördjupning är $O(m+n)$ i en graf med n noder och m bågar? Motivera! (2p)

Svar. Nej, om man har en graf med noderna $0, 1, 2, \ldots, n-1$ och bågar mellan konsekutiva tal, så kommer iterativ fördjupning att besöka $2+3+\cdots+(n-1)$ noder. Alltså är komplexiteten $O(n^2)$.

7. Din uppgift är att skriva effektiva algoritmer för att räkna röster i ett val. Antag att det finns n väljare och k kandidater. Din uppgift är att skriva en algoritm som returnerar den kandidat som fått flest röster. Vilken algoritm (och datastruktur) som är bäst att använda beror på förhållandet mellan k och n.

   (a) I vanliga val är k mycket mindre än n. Vilken algoritm och datastruktur är då lämplig att använda? Du behöver inte ge pseudokod, men måste förklara tydligt de olika stegen i algoritmen. Du ska även ange O-komplexitet för din algoritm som funktion av n och k. (5p)

   Svar. Man tilldelar varje kandidat ett nummer $0, \ldots, k-1$. Sedan använder man helt enkelt ett heltalsfält med k celler för att räkna rösterna: när man räknar en röst på kandidat i ökar man cellen med index i. När rösterna är färdigräknade löper man genom fältet på jakt efter den kandidat som fått flest röster.

Rösträkningen tar $O(n)$ och att finna största elementet i fältet tar $O(k)$ så algoritmen har komplexiteten $O(k+n)$.

(b) Om k är mycket större än n är det dock bättre att använda en annan metod än i (a). Föreslå även här en lämplig algoritm och datastruktur, samt ange O-komplexitet för din algoritm som funktion av n och k. (5p)
Svar. Vi antar att rösterna ligger lagrade i ett indatafält.

Om k är mycket större än n kommer inte alla kandidaterna att få röster och det är onödigt att använda ett fält med storlek k. I stället kan man löpa genom indatafältet med röster och tilldela ett nummer bara till de k_0 kandidater som fått någon röst. Detta tar $O(n)$. Sedan gör man som i (a). Man löper genom indatafältet en andra gång och räknar rösterna. Detta tar $O(n)$. Slutligen löper man genom rösträkningsfältet på jakt efter den kandidat som fått flest röster. Detta tar $O(k_0)$ vilket också är $O(n)$. Alltså är den totala komplexiten för algoritmen $O(n)$, dvs den är oberoende av k.

Poängsättningen kommer att bero både på hur pass effektiv din algoritm är och hur bra din komplexitetsanalys är.