Tentamen
Datastrukturer för D2
TDA 131
(med lösningsförslag)

20 december 2003

- Tid: 8.45 - 12.45
- Ansvarig: Peter Dybjer, tel 7721035 eller 405836
- Max poäng: 60. Vart och ett av de sex problemen ger maximalt 10 p.
- Betygsgränser: 3 = 30 p, 4 = 40 p, 5 = 50 p
- Inga hjälpmedel.
- 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. Följande grannmatris representerar en viktad graf.

	A	B	C	D	E	F
A	-	1	4	5	-	-
B	1	-	2	-	2	-
C	4	2	-	5	6	4
D	5	-	5	-	-	2
E	-	2	6	-	-	6
F	-	-	4	2	6	-

Talen i matrisen representerar bågarnas vikter. Ett streck (-) betyder att bågen saknas.

(a) Rita grafen! Sätt ut vikterna på bågarna.

(b) Man kan använda Dijkstras algoritm för att finna kortaste vägen mellan två noder i en graf. Som i lab 4 vill man ofta både veta vilka noder som ligger på den kortaste vägen och hur lång vägen är (dvs summan av de ingående bågarnas vikter). Beskriv i ord hur Dijkstras algoritm gör för att beräkna dessa två saker. (Du får delpoäng om du bara kan beräkna längden men inte nodlistan.) Du behöver inte ge fullständig pseudokod - det räcker om du förklarar vilka datastrukturer du använder och förklarar noggrannt vilken roll de spelar under beräkningen.

Svar: På sid 587 i G & T finns pseudokod för en version av Dijkstra som bara beräknar längden av kortaste vägen från en given nod v till en godtycklig annan nod u. (Ett fullgott svar på tentan kräver att de viktigaste idérna bakom denna pseudokod beskrivs klart.) Observera i synnerhet att algoritmen använder sig av en prioritetskö Q som bestämmer i vilken ordning algoritmen "besöker" de olika noderna i grafen och ett avstånd $D[u]$ av den kortaste vägen från den givna startnoden v till u.
Om man dessutom vill returnera en lista med noderna på den kortaste vägen från v till u lagrar man för varje nod u inte bara längden $D[u]$ av den kortaste vägen från v till u utan även den nod $P[u]$ som kommer innan u på denna kortaste väg. Från början sätter man $D[u]$ till oändligheten och $P[u]$ till en nollpekare. Sedan uppdaterar man både $D[u]$ och $P[u]$ med allt kortare avstånd och vägar efterhand som man besöker nya noder. Dijkstars algoritm som den beskrivs i boken terminerar när man besökt alla noder. Då kommer $D[u]$ att vara den korrekta längden av den kortaste vägen och $P[u]$ att vara föregångaren till u på denna väg.

När Dijkstras algoritm används för att beräkna kortaste vägen mellan två givna noder v och u kan man terminera algoritmen efter det att u besökts. Man kan då ge som utdata längden $D[u]$ och vägen $u, P[u], P[P[u]], \ldots$ i omvänd ordning (denna väg kan förstås dessutom reverseras om så önskas).

(c) Dijkstras algoritm arbetar stegvis. Visa hur algoritmen fungerar genom att visa vilka mellanresultat (tillstånd hos datastrukturerna) som Dijkstras algoritm (enligt din beskrivning ovan) lagrar för att kunna returnera den kortaste vägen och denna vägs längd mellan A och F i grafen ovan.

Svar: Vi visar efter varje besök av en ny nod tillståndet hos prioritetskön och värdena på $D[u]$ och $P[u]$ för alla noder u i prioritetskön.

A	$(B,1,A),(C,4,A),(D,5,A)$
B	$(C,3,B),(E,3,B),(D,5,A)$
C	$(E,3,B),(D,5,A),(F,7,C)$
E	$(D,5,A),(F,7,C)$
D	$(F,7,C)$
F	

Första raden betyder alltså att när vi har besökt A så ligger B först i Q med $D[B] = 1$ och $U[B] = A$, osv.

(d) Är det i allmänhet lämpligt att använda en grannmatris för att representera en graf när man implementerar Dijkstra! Varför eller varför inte? Motivera med hjälp av O-komplexiteter.

Svar: Nej, det är i allmänhet lämpligare att använda grannlistor. Varje gång en nod u besöks går man genom dess grannar och uppdaterar $D[u]$ och $P[u]$. Om man använder en grannmatris måste man gå genom alla noder i grafen för att finna dessa grannar. Om man däremot använder grannlistor finns us grannar direkt åtkomliga. Totalt behöver man alltså under exekveringen inspektera grannmatrisen $O(n^2)$ gånger om det är n noder i grafen men bara inspektera grannlistorna $O(m+n)$ om det är m bågar i grafen. Om inte grafen är "tät" eller liten är grannlistan alltså klart bättre än grannmatrisen. Ett annat men oftast mindre viktigt argument är att grannlistornas utrymmeskomplexitet $O(m+n)$ också är bättre än grannmatrisens $O(n^2)$ om inte grafen är tät.

2. Följande tre algoritmer utför samma uppgift. Givet ett fält med heltal (som kan vara både positiva och negativa) räknar de ut den maximala summan av elementen i en delsekvens (utan gap). Din uppgift är att tala om vilken asymptotisk tidskomplexitet de tre algoritmerna har. Du ska ange bästa O-komplexitetsklass för exekveringstiden uttryckt som funktion av fältets storlek n (`a.length` i koden nedan). Kom ihåg att alla svar måste motiveras ordentligt!

   (a)
   ```
   public static int maxSubsequenceSum(int [] a) {
     int maxsum = 0; int thisSum = 0;
     for(int i = 0, j = 0; j < a.length; j++) {
       thisSum += a[j];
       if(thisSum > maxSum)
         {maxSum = thisSum; seqStart = i; seqEnd = j}
       else if(thisSum < 0) {
         i = j + 1;
         thisSum = 0
         }
       }
     return maxSum;
   }
   ```

   Svar: Algoritmen är $O(n)$. `for`-loopen exekveras n gånger och varje exekvering är $O(1)$. Initialisering och avslutning av programmet tar också $O(1)$.

   (b)
   ```
   public static int maxSubsequenceSum(int [] a) {
     int maxsum = 0;
     for(int i = 0; i < a.length; i++) {
       int thisSum = 0;
       for(int j = i; j < a.length; j++) {
         thisSum += a[j];
         if(thisSum > maxSum)
           {maxSum = thisSum; seqStart = i; seqEnd = j}
         }
       }
     return maxSum;
   }
   ```

   Svar: Algoritmen är $O(n^2)$. Den yttre `for`-loopen exekveras n gånger och den inre `for`-loopen exekveras $n + (n-1) + \cdots + 2 + 1$ gånger. Alternativt kan man direkt se att den inre `for`-loopen exekveras maximalt n gånger för varje exekvering av den yttre loopen dvs maximalt n^2 gånger. Som ovan är varje exekvering av loopen $O(1)$ liksom initialisering och avslutning av de bägge looparna.

   (c)
   ```
   public static int maxSubsequenceSum(int [] a) {
     int maxsum = 0;
     for(int i = 0; i < a.length; i++) {
       for(int j = i; j < a.length; j++) {
   ```

```
      int thisSum = 0;
      for(int k = i; k <= j; k++) thisSum += a[k];
      if(thisSum > maxSum)
        {maxSum = thisSum; seqStart = i; seqEnd = j}
      }
    }
  return maxSum;
}
```

Svar: Algoritmen är $O(n^3)$. Den yttersta `for`-loopen exekveras n gånger, den mittersta `for`-loopen exekveras maximalt n gånger för varje exekvering av den yttersta loopen dvs maximalt n^2 gånger, och den innersta loopen maximalt n gånger för varje exekvering av den mittersta loopen, dvs maximalt n^3 gånger. Även här är varje exekvering av loopen $O(1)$ liksom initialisering och avslutning av de tre looparna.

3. Din uppgift är att implementera en klass för ändliga mängder av heltal med några vanliga mängdoperationer:

```
public class SetInt{

   ... // tillståndsvariabler

   public boolean member(int n);
   public void insert(int n);
   public void delete(int n);
   public void union(SetInt s);
   public void intersect(SetInt s);
}
```

Operationerna ska förstås ha sin vanliga mängdteoretiska betydelse. T ex betyder `intersect(s)` att man ska ersätta den mängd `t` som är lagrad i objektet med snittet av `s` och `t`.

Flera olika datastrukturer kan vara lämpliga för att implementera denna klass - vilken som är bäst beror på vilka operationer som är vanligast! Du ska diskutera följande fall:

(a) `member` är vanligast;

(b) `insert` är vanligast;

(c) `union` och `intersect` är vanligast.

(d) Visa också hur man implementerar `intersect` genom att ge pseudokod eller Javakod.

I deluppgift (a), (b) och (c) ska du föreslå minst två datastrukturer som du anser vara huvudalternativ och jämföra dem med varandra. Du ska motivera varför du anser dem vara bäst utifrån den asymptotiska tidskomplexiteten hos de olika implementeringarna av operationerna! Tänk på att även om du i första hand ska ta hänsyn till komplexiteten hos de vanligaste operationerna, måste du även i andra hand ta hänsyn till komplexiteten hos de mindre vanliga.

Svar. De aktuella datastrukturerna är hashtabeller (HT), balanserade sökträd (BST) och sorterade listor (SL). (Exempel på balanserade sökträd är AVL-träd, (2,4)-träd odh rödsvarta träd. Även skiplistor har samma asymptotiska medelkomplexiter som de balanserade sökträden och kan också komma i fråga.) Nedanstående tabell visar O-komplexiteter för de olika operationerna. Vi antar att mängden som lagras i objektet har n element och mängden `s` som är argument till `union` och `intersect` har m element. Vidare har hashtabellerna storlekarna N respektive M. (För att nedanstående komplexiteter ska gälla för hashtabeller förutsätter vi dessutom som vanligt bra hashfunktion och låg

belastningsfaktor.)

	HT	BST	SL
(a)	$O(1)$	$O(\log n)$	$O(\log n)$
(b)	$O(1)$	$O(\log n)$	$O(n)$
(c)	$O(M+m)$	$O(m \log(m+n))$	$O(m+n)$

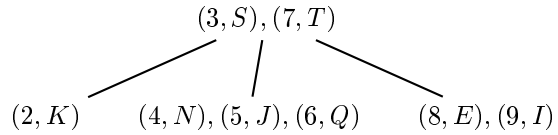
Tabellen visar att hashtabeller i allmänhet är att föredra i (a) och (b). I (a) är både binära sökträd och sorterade listor möjliga som andrahandsalternativ, vilket som är att föredra beror på övriga faktorer som frekvensen hos de andra operationerna, sökalgoritmens konstant, mm. I (b) är andrahandsalternativet binära sökträd. G & T rekommenderar sorterade listor i (c), se 10.2.1, men hashtabeller är också ett alternativ. Vilket som är allra bäst kan inte avgöras enbart utifrån de asymptotiska komplexiteterna ovan, utan beror på flera saker som relativ frekvens hos de olika operationerna, förhållandet mellan $M+m$ och $m+n$, samt de olika algoritmernas konstanter.
Javakod för deluppgift (d) finns på sidan 465-466 i G & T.

4. (2,4)-träd är ett slags balanserade sökträd som kan användas för att implementera lexika. Vi antar här att dessa lexika är "funktionella", dvs en nyckel k kan bara förekomma i högst ett par (k, v).

   Våra (2,4)-träd har följande egenskaper. (Definitionen avviker något från den i Goodrich och Tamassia eftersom vi förutsätter funktionella lexika.)

   - Varje nod lagrar ett, två eller tre par (k, v) av nycklar k och element v. Vi antar här att nycklarna är heltal.
   - En intern nod som lagrar n par av nycklar och element har $n + 1$ barn.
   - Följande sökträdsegenskap gäller:
     - om en nod lagrar ett par (k_1, v_1) och paret (k, v) ligger i det första (vänstra) delträdet till noden så gäller $k < k_1$. Om (k, v) ligger i det andra (högra) delträdet så gäller $k_1 < k$.
     - om en nod lagrar två par $(k_1, v_1), (k_2, v_2)$ och (k, v) ligger i det första delträdet så gäller $k < k_1$. Om (k, v) ligger i det andra delträdet så gäller $k_1 < k < k_2$. Om (k, v) ligger i det tredje delträdet så gäller $k_2 < k$.
     - om en nod lagrar tre par $(k_1, v_1), (k_2, v_2), (k_3, v_3)$ och (k, v) ligger i det första delträdet så gäller $k < k_1$. Om (k, v) ligger i det andra delträdet så gäller $k_1 < k < k_2$. Om (k, v) ligger i det tredje delträdet så gäller $k_2 < k < k_3$. Om (k, v) ligger i det fjärde delträdet så gäller $k_3 < k$.
   - Alla lövnoder har samma djup.

   Här är en bild på ett (2,4)-träd:



   (a) Definiera en klass i Java som implementerar (2,4)-träd. Klassen ska bara ha *en* metod. Denna metod är en sökmetod som tar en nyckel (ett heltal) k som indata och returnerar ett element v om (k, v) finns lagrat i någon nod i (2,4)-trädet. Annars ska metoden returnera `NO_SUCH_KEY`.

   (b) Rita två bilder som visar hur

      i. ovanstående (2,4)-träd,
      ii. det tomma (2,4)-trädet

      representeras i minnet!

      Tänk på att rita alla relevanta pekare och/eller fält som representationerna använder.

   (c) Definiera också en konstruerarmetod som skapar ett tomt (2,4)-träd!

Eftersom du inte har någon Java-bok tillgänglig kommer vi att ha överseende med smärre syntaxfel. Logiska fel ger dock poängavdrag.

Svar: Det enklaste sättet att implementera (2,4)-träd är att representera dem som rödsvarta träd. Sambandet beskrivs i G & T 9.5 med implementering av rödsvarta träd i 9.5.2. Sökning i rödsvarta träd fungerar precis som sökning i binära sökträd, se t ex 9.1.3.

5. (a) Skriv en algoritm i pseudokod som givet en oriktad graf beräknar antalet sammanhängande komponenter i grafen! En sammanhängande komponent är en maximal sammanhängande delgraf. En delgraf är sammanhängande om varje par av noder i delgrafen kan förbindas med en väg.

   (b) Vilken O-tidskomplexitet har din algoritm uttryckt som en funktion av antalet noder n och antalet bågar m som finns i grafen? Motivera utifrån pseudokoden.

   (c) Skriv en algoritm i pseudokod som givet en riktad graf avgör om grafen saknar cykler, dvs om grafen är en DAG ("Directed Acyclic Graph").

   (d) Ange även här O-tidskomplexiteten! Motivera!

   För att få maximal poäng på uppgiften måste dina algoritmer ha så god asymptotisk tidskomplexitet som möjligt.

   Svar: Man kan använda djupet-först sökning både i (a) och (c), se avsnitt 12.3.1 i G & T särskilt Proposition 12.13 där också tidskomplexiteten $O(m+n)$ för båda uppgifterna anges. Vi antar här att n är antalet noder i grafen och m är antalet bågar.

6. Du har fått i uppgift att skriva en artikel för ett uppslagsverk om hashning. Artikeln riktar sig till en person som har elementära kunskaper om programmering, men inte vet något ytterligare om datalogi. Den ska innehålla 150 - 300 ord. Du ska alltså kort och klart sammanfatta det viktigaste du vet om hashning på ett populärvetenskapligt sätt. Försök ge läsaren svar på frågor som "Vad är hashning?", "Vad används det till?", "Vad har metoden för för- och nackdelar?".

   Poängsättningen grundas både på hur pass välskriven artikeln är, om den är "på rätt nivå", och förstås, om den gör en bra avvägning av vad som är viktigaste att veta!

   Svar: Hashtabeller beskrivs i 8.3 i G & T. Ett fullgott svar bör innehålla en lättförståelig beskrivning av hur hashtabeller fungerar inklusive något om kollisionshantering. Vidare bör något sägas om hashtabellens effektivitet som är huvudskälet till att den är en mycket viktig datastruktur.