Tentamen med lösningsförslag
Datastrukturer för D2
TDA 131

18 december 2003

- Tid: 8.30 - 12.30
- 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!

Kommentar. Några allmänna rättningsanvisningar.

- Motivera alltid kort era poängavdrag. Jag måste förstå era motiv när jag granskar rättningen.
- Gör en lista på era poängsättningsprinciper, så att ni är konsekventa och jag kan kontrollera det. Jag ger vissa anvisningar nedan, men dessa kan förstås inte täcka allt.
- Ge endast hela poäng.
- Uppgiftens formulering ska betraktas som "lagtext", dvs grunden för poängsättningen ska vara hur pass korrekt svaret är på frågan som den faktiskt formulerats. Man bör också vara ganska generös med alternativa tolkningar av texten om de är rimliga och intelligenta.

- I princip kräver vi alltid motiveringar till svar för att ge poång. Ibland gör vi dock undantag från denna princip, se nedan. Motiveringar ska förstås vara korrekta och fullständiga, men i allmänhet räcker det om de är kortfattade. Tänk också på att rättningen ska vara anpassad till elevernas nivå och kräv inte alltför hög grad av stringens i svaren.

1. Vilka av följande påståenden är korrekta och vilka är felaktiga? Motivera! Svar utan korrekt motivering ger inga poäng.

   (a) Man sparar minne om man lagrar ett binärt träd i ett fält i stället för som en länkad struktur.

   Svar. Nej. Om vi lagrar det som en länkad struktur är storleken på minnet $O(n)$ medan det är $O(2^n)$ i värsta fall om vi lagrar det i ett fält. Om det binära trädet är helt obalanserat blir ju trädets höjd $O(n)$ och storleken på fältet växer exponentiellt med höjden.

   Kommentar. Varje deluppgift (a) - (e) ger 2p. Korrekt svar med kort korrekt motivering ger full poäng. Bara ja eller nej ger inga poäng.

   (b) Sökning i ett splayträd tar $O(\log n)$ i värsta fall.

   Svar. Nej. Ett splayträd kan vara obalanserat. En enstaka sökning är $O(n)$ i värsta fall. Den amorterade komplexiteten är dock $O(\log n)$.

   Kommentar. Ge 2 poäng även för svaret "Ja, den amorterade komplexiteten är $O(\log n)$".

   (c) Hashtabeller med öppen adressering ("open addressing") använder mer minne än hashning med hinkar ("hashing in buckets" eller "chaining").

   Svar. Nej. Om de två hashtabellerna har samma antal celler använder hashning med hinkar mer minne eftersom varje cell måste lagra en lista av objekt och inte bara ett enstaka objekt.

   Kommentar. Eftersom uppgiften inte fastlägger att hashtabellerna ska ha lika många celler, bör variationer på ovanstående svar accepteras om de är korrekta.

   Någon elev hävdar att öppen adressering tar mer minne eftersom den är mer känslig för hög belastningsgrad. Även om detta är en riktig och relevant observation måste ju detta vägas mot det extra uttrymme som hinkarna tar. För full poäng krävs att båda dessa aspekter diskuteras.

   (d) Grannmatriser använder mer minne än grannlistor om man har glesa grafer, dvs grafer som inte har så många bågar.

   Svar. Ja. Grannmatrisen har minneskomplexiteten $O(n^2)$ medan grannlistorna har $O(n+m)$ om grafen har n noder och m bågar. Om grafen är gles är m mycket mindre än n^2.

   (e) Om antalet noder i en riktad graf är n så är antalet bågar högst n^2 om vi förutsätter att det inte finns några parallella bågar. (Dock tillåter vi "self loops" dvs bågar med samma källa och mål.)

   Svar. Ja. Det finns högst en båge mellan varje par av noder och det finns n^2 par av noder.

   Kommentar. En del elever undrade om två bågar med samma ändpunkter men olika riktning räknas som parallella. Jag svarade "nej", men jag tycker att vi även skulle kunna acceptera korrekta svar som räknar sådana bågar som parallella också. Det krävs dock ett tydligt och korrekt resonemang kring svar som grundar sig på denna tolkning för full poäng.

2. (a) Vad beräknar följande två Java-metoder?

```
int f(int[] a){
   int s = a[0];
   for (int iter = 1; iter < a.length; iter++) {
      s = s + a[iter];
   }
   return s;
}

int g(int[] a, int k){
   for (int iter = 0; iter < a.length; iter++) {
      if (a[iter] == k) {
         return iter;
      }
   }
   return -1;
}
```

Svar. `f` beräknar summan av elementen i fältet `a`

`g` letar efter en nyckel `k` i `a` och returnerar minsta index `i` där `k` ligger lagrad. Om `k` inte finns i `a` returneras -1.

Kommentar. 2p. 1p för varje funktion.

(b) Låt $T_{\mathtt{f}}(\mathtt{n})$ och $T_{\mathtt{g}}(\mathtt{n})$ vara tiderna det tar att exekvera `f` och `g` på ett fält `a` med längden `n` i *värsta* fall. Ange lämpliga O-komplexitetsklasser för $T_{\mathtt{f}}(\mathtt{n})$ och $T_{\mathtt{g}}(\mathtt{n})$! För full poäng ska komplexitetsklasserna vara optimala, dvs så små som möjligt.

Svar. Både $T_{\mathtt{f}}(\mathtt{n})$ och $T_g(\mathtt{n})$ är $O(\mathtt{n})$ i värsta fallet. Båda genomlöper en for-loop `n` gånger i värsta fallet och varje iteration tar konstant tid.

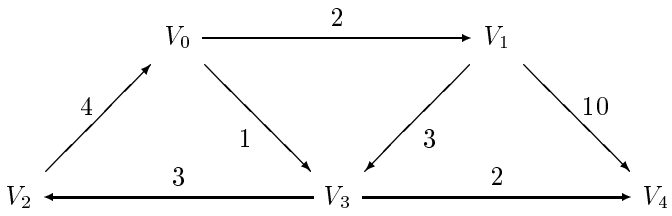
Kommentar. 4p. 2p för varje funktion. För full poäng krävs korrekt motivering.

(c) Låt $T'_{\mathtt{f}}(\mathtt{n})$ och $T'_{\mathtt{g}}(\mathtt{n})$ vara exekveringstiderna för `f` och `g` i *bästa* fall.

- Är $T_{\mathtt{f}}(\mathtt{n}) = T'_{\mathtt{f}}(\mathtt{n})$?
  Svar. Ja, om `a` har längden `n` så kommer exekveringen av metoden att genomlöpa samma instruktioner (och genomlöpa for-loopen lika många gånger) oberoende av hur `a` ser ut. Alltså tar exekveringen lika lång tid i alla fall.
- Har $T_{\mathtt{f}}(\mathtt{n})$ och $T'_{\mathtt{f}}(\mathtt{n})$ samma O-komplexitet?
  Svar. Ja. Eftersom $T_f(\mathtt{n}) = T'_f(\mathtt{n})$ har de förstås samma O-komplexitet också.
- Är $T_{\mathtt{g}}(\mathtt{n}) = T'_{\mathtt{g}}(\mathtt{n})$?
  Svar. Nej, de är olika eftersom man i bästa fall hittar nyckeln omedelbart och i värsta fall måste man söka genom hela fältet.
- Har $T_{\mathtt{g}}(\mathtt{n})$ och $T'_{\mathtt{g}}(\mathtt{n})$ samma O-komplexitet?
  Svar. Nej. $T_g(\mathtt{n})$ är $O(1)$ och $T'_g(\mathtt{n})$ är $O(\mathtt{n})$ och alltså olika.

Kommentar. 4p. 1p för varje delfråga. För poäng krävs korrekt motivering. Eftersom den andra frågan följer direkt från den första tycker jag dock att vi ger poäng utan motivering om svaret på den första frågan är korrekt. Samma princip för tredje frågan vars svar följer direkt från den fjärde.

3. Betrakta följande graf:



(a) Visa hur man representerar denna graf med hjälp av en grannmatris!

Svar. Se avsnitt 12.2.2 i kursboken

Kommentar. 2p. Här räcker det om de ritar en matris med vikter för bågarna och någonting som markerar frånvaro av båge.

Även en grannmatris utan vikter (bara nollor och ettor) accepteras eftersom uppgiften inte uttryckligen frågar efter representationen av vikter.

(b) Visa hur man representerar denna graf med hjälp av en grannlista!

Svar. Se avsnitt 12.2.3 i kursboken

Kommentar. 3p. Här kommer vi nog att få stor variation i svaren. Ett perfekt svar bör rita ut alla minnesceller och pekare som implementeringen använder. Eftersom boken ofta använder särskild notation för abstrakta listor måste vi dock även acceptera svar som visar en abstrakt nodlista med tillhörande abstrakta listor av bågar, och vi måste nog även vara liberala med olika sätt att rita dessa.

(c) Dijkstras algoritm kan användas för att bestämma kortaste vägen mellan två noder. I figuren ovan är V_0 startnod och V_4 målnod. Din uppgift är att förklara hur algoritmen fungerar genom att visa steg för steg hur algoritmen arbetar på grafen ovan och hur de kortaste vägarna successivt bestäms. Du ska alltså rita en följd av grafer som visar vilka mellanresultat som beräknats efter varje iteration. Som vanligt ska du också ge motiveringar; det räcker inte att bara rita figurer.

Svar. Se avsnitt 12.6.1 i kursboken. Figur 12.15-16 visar en följd av grafer som illustrerar hur Dijkstras algoritm beräknas steg för steg.

Kommentar. 5p. För full poäng krävs både en följd av grafer och en kort beskrivning. Vi bör också acceptera svar där de på annat sätt visar steg för steg hur algoritmen lagrar information om vilka noder som besökts och hur prioritetskön ändrar sig.

Vi kan dock inte kräva att de explicit diskuterar prioritestköns implementering eftersom detta kan göras på två olika sätt.

4. Analysera effektiviteten hos insertion sort och quicksort för små och stora listor!

   (a) Antag först att vi vill sortera en lista med precis 4 element. Hur många jämförelser gör de båda algoritmerna i *värsta* fall och i *bästa* fall? Motivera ditt svar på följande sätt:

   - Ge exempel på en indatalista med 4 element som ger *maximalt* antal jämförelser för *insertion sort* och förklara vilka jämförelser som görs.
   - Ge exempel på en indatalista med 4 element som ger *minimalt* antal jämförelser för *insertion sort* och förklara vilka jämförelser som görs.
   - Ge exempel på en indatalista med 4 element som ger *maximalt* antal jämförelser för *quicksort* och förklara vilka jämförelser som görs.
   - Ge exempel på en indatalista med 4 element som ger *minimalt* antal jämförelser för *quicksort* och förklara vilka jämförelser som görs.

   För att få full poäng ska du förutsätta att algoritmerna är implementerade på ett optimalt sätt. (Du behöver dock inte visa hur de är implementerade.)

   Svar.

   - För insertion sort är värsta fallet när elementen ligger i omvänd ordning. Då är antalet jämförelser 1 + 2 + 3. Följande indatalista ger maximalt antal jämförelser:

     4|321

     Det vertikala strecket markerar skiljelinjen mellan det sorterade initialsegmentet och den osorterade resten av listan. Först sätter vi in 3 på rätt plats genom att jämföra med 4 (1 jämförelse):

     34|21

     Sedan sätter vi in 2 på rätt plats genom att jämföra med 4 och 3 (2 jämförelser):

     234|1

     Sedan sätter vi in 1 på rätt plats genom att jämföra med 4, 3 och 2 (3 jämförelser):

     1234|

   - För insertion sort är bästa fallet när fältet redan är sorterat. Då är antalet jämförelser 1 + 1 + 1 och en sorterad indatalista ör:

     1|234

     Först sätter vi in 2 på rätt plats genom att jämföra med 1 (1 jämförelse):

     12|34

     Sedan sätter vi in 3 på rätt plats genom att jämföra med 2 (1 jämförelse). Obs att vi börjar att jämföra från höger, annars får vi inte optimal komplexitet i bästa fallet!

     123|4

Sedan sätter vi in 4 på rätt plats genom att jämföra med 3 (1 jämförelse):

$$1234|$$

- Quicksort. I värsta fallet väljer man t ex 1 som pivotelement från början. Då delas listan upp i en tom lista och en 3-elementslista:

  $$[], [2, 3, 4]$$

  Den tomma listan är redan sorterad. För att sortera den andra listan väljer vi ett nytt pivotelement, i värsta fall t ex 2. Med hjälp av 2 jämförelser delar vi upp denna lista i två dellistor

  $$[], [3, 4]$$

  Återstår att sortera listan [3,4]. Detta tar 1 jämförelse.
- I bästa fall delar pivotelementet listan i två (nästan) lika delar. Säg att vi väljer pivotelementet 2 till att börja med. Då behöver vi 3 jämförelser för att dela in listan i de två dellistorna

  $$[1], [3, 4]$$

  Listan med bara ett element är redan sorterad, medan den andra listan sorteras med 1 jämförelse. (Man väljer det ena elementet som pivotelement och jämför det med det andra.)

Sammanfattning. Insertion sort behöver göra 3 jämförelser i bästa fall och 6 jämförelser i värsta fall. Quicksort behöver göra 4 jämförelser i bästa fall och 6 jämförelser i värsta fall.

Kommentar. 6p. Ge 3p för insertion sort och 3p för quicksort. För full poäng krävs att de ger exempel på indatalistor som ger upphov till bästa och sämsta fallet och beskriver vilka jämförelser som görs i desssa fall.

(b) Är insertion sort eller quicksort effektivast om man vill sortera 1000 element? Analysera både värsta fallet och bästa fallet. Här kan du förstås inte beräkna den exakta tidsåtgången utan det räcker att du utgår från din kunskap om algoritmernas asymptotiska komplexitet i bästa och värsta fallet.

Svar. Insertion sort gör $O(n)$ jämförelser i bästa fallet och $O(n^2)$ jämförelser i värsta fallet. Quicksort gör $O(n \log n)$ jämförelser i bästa fallet och $O(n^2)$ i värsta fallet. Den asymptotiska komplexiteten är alltså bättre i bästa fallet för insertion sort och i värsta fallet har de två algoritmerna samma asymptotiska komplexitet.

Vad gäller då för den exakta tidskomplexiteten för att sortera $n = 1000$ element? Då gäller att $\log n$ är ca 10 och alltså är $n \log n$ är ca 10000. Dessutom har insertion sort bättre "konstant" och är alltså mer än 10 gånger snabbare än quicksort i bästa fall. I värsta fallet har vi samma asymptotiska komplexitet, men eftersom insertion sort har bättre konstant är den snabbare.

Kommentar. 4p. För full poäng krävs korrekta O-komplexiteter men här kräver vi inte motiveringar till dessa komplexiteter. För full poäng ska också något kort sägas om fallet $n = 1000$. 1p avdrag om inte något sägs om $n = 1000$ och för varje felaktig O-komplexitet.

5. Här är ett litet gränssnitt (i Java) för ändliga mängder av heltal:

```
public interface Set {
   public boolean member(int element);
   public void remove(int element);
}
```

Metodanropet `member(element)` ska returnera `true` om `element` finns i mängden och `false` annars. Metodanropet `remove(element)` ska ta bort elementet `element` från mängden.

(a) Implementera gränssnittet `Set` i detaljerad pseudokod eller i Java. Du behöver bara implementera de två metoderna och de hjälpmetoder och hjälpklasser du behöver. Du behöver heller inte implementera någon konstruerare. Du kan själv välja storleken på hashtabellen. Som hashfunktion kan du använda modulofunktionen (`k` modulo `N` skrivs `k % N` i Java). Du får också själv välja hur du implementerar hashtabellen (öppen adressering, hashning med hinkar, ...). Du måste dock förklara vilken implementeringsmetod du använt.

(Du kan få delpoäng på denna uppgift även om du inte ger en fullständig implementering i Java eller fullständig pseudokod. T ex får du avdrag om du skriver lösningen i ofullständig pseudokod eller om du förutsätter att det finns en viss hjälpmetod eller hjälpklass utan att implementera den.)

Svar. Det är enklast att implementera hashing in buckets. En implementering för HashMaps i pseudokod finns i kursboken på sidan 367. Koden för HashSet är väsentligen densamma. För full poäng ska man också implementera hinkarna. Det enklaste är att använda en enkellänkad lista, se sidan 359 i boken. Även här behöver man göra en smärre anpassning av koden från Map till Set.

Kommentar. 8p. 2p för korrekt representation av själva hashtabellen, och 3p för varje metod. Om man implementerar hashning i hinkar så måste man även implementera hinkarna. Om man förusätter att sådan implementering finns blir det 3p avdrag. Annars 2p avdrag för varje allvarligt fel, och 1p avdrag får mindre fel. Inget avdrag alls för syntaxfel och andra smärre ofullständigheter som man kan acceptera i pseudokod.

(b) Visa sedan hur du har tänkt dig att lagra följande mängd av element om hashtabellens storlek är $N = 11$!

$$19, 53, 30, 64, 31$$

Hur ser din datastruktur ut om du sedan tar bort elementet 19 från mändgen? (Du kan få poäng på denna uppgift även om du inte gjort en fullständig implementering i (a).)

Kommentar. 2p. För att få full poäng ska bilden överensstämma med implementeringen i (a). De som inte gjort någon implementering i (a) får dock poäng ändå, om implementeringen är korrekt.

6. (a) Heapar brukar lagras i fält. Skriv en algoritm i detaljerad pseudokod eller i Java som har som indata ett fält (med storleken n) med heltal och returnerar `true` om fältet representerar en heap med storleken n och `false` annars. Även en mer skissartad beskrivning av algoritmen kan ge delpoäng om den är tillräckligt klar.

Svar. Fältet ska representera ett fullständigt binärt träd, där roten är lagrad i cell 0 och barnen till en nod som är lagrad i cell n ligger i cell $2 * n + 1$ och $2 * n + 2$. Att det binära trädet är fullständigt betyder att alla de n cellerna i fältet representerar noder.

Vi skriver en rekursiv Javametod som kontrollerar att heapegenskapen är uppfylld för alla noder. Metoden `isSubHeap(a,n)` kontrollerar att det binära delträdet med roten lagrad i cell `n` verkligen är en delheap, genom att kontrollera att noden i cell `n`'s nyckel är mindre än eller lika med barnens (om de finns) och att barnen dessutom är rötter till delheapar. Metoden `isHeap(a)` anropar `isSubHeap(a,0)` som alltså kontrollerar att hela det binära trädet är en heap.

```
boolean isHeap(int[] a){return isSubHeap(a,0);}

boolean isSubHeap(int[] a,int n){
   int size = a.length;
   int firstChild = 2*n + 1;
   int secondChild = 2*n + 2;
   boolean firstRight = firstChild >= size ||
      (a[n] <= a[firstChild] && isSubHeap(a,firstChild));
   boolean secondRight = secondChild >= size ||
      (a[n] <= a[secondChild] && isSubHeap(a,secondChild));
   return firstRight && secondRight;
}
```

Kommentar. 7p. Om algoritmen har sämre asymptotisk komplexitet än linjär blir det 3p avdrag. Annars samma princip för poängsättningen som ovan, dvs 2p avdrag för allvarliga programmeringsfel och 1p för mindre allvarliga. Inga avdrag för syntaxfel och andra smärre ofullständigheter som kan accepteras i pseudokod.

(b) Vilken tidskomplexitet har ditt program uttryckt som funktion av antalet element i heapen? Motivera!

Svar. $O(n)$. Algoritmen använder $O(1)$ tid för att behandla varje nod. Den kontrollerar först om ett visst barn finns och om så är fallet om förälderns nyckel är mindre än eller lika med barnets. Förutom det görs ett visst initialiseringsarbete som också tar $O(1)$.

Kommentar. 3p. Bedömingen grundar sig på om de gjort en korrekt analys av den algoritm de skrivit i (a). För full poäng krävs motivering. Rätt svar utan motivering ger 1p.

För full poäng på uppgiften ska algoritmen ha korrekt angiven optimal asymptotisk komplexitet.