Tentamen
Datastrukturer D
DAT 036/INN960

18 december 2009

- Tid: 8.30 - 12.30
- Ansvarig: Peter Dybjer, tel 7721035 eller 405836
- Max poäng på tentamen: 60.
- Betygsgränser, CTH: 3 = 24 p, 4 = 36 p, 5 = 48 p, GU: G = 24 p, VG = 48 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.
- **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? Motivera!

   (a) Stackar kan implementeras effektivt med länkade listor. (2p)

   Svar: Ja. De viktigaste stackoperationerna manipulerar bara toppen på stacken. Alla dessa är $O(1)$ när de implementeras med länkade listor.

   (b) En algoritm med komplexiteten $O(n^2)$ är alltid långsammare än en algoritm med komplexiteten $O(n \log n)$. (2p)

   Svar: Nej. Eftersom den exakta komplexiteten både beror på konstanten och på indatastorleken, så kan en algoritm med komplexiteten $O(n^2)$ vara snabbare än en algoritm med komplexiteten $O(n \log n)$. T ex är insertionsort snabbare än heapsort för sorterade indata och små indata.

   (c) Om ett fält nästan är sorterat är det bättre att använda heapsort än insertionsort. (2p)

   Svar: Nej. Insertionsort är $O(n)$ om fältet är helt sorterat och är då snabbare än heapsort och denna fördel gäller också om fältet är nästan sorterat.

   (d) Komplexiteten hos quicksort beror på ordningen mellan elementen i fältet som ska sorteras. (2p)

   Svar: Ja. Partitionsoperationen byter plats på element som "ligger fel" m a p pivotelementet. Har man tur behöver man inte byta plats på några element. Har man otur behöver man byta plats på många. Alltså påverkas den exakta tidsåtgången.

   (e) Hashtabeller är effektivare än balanserade träd om man ska söka efter data på externminne. (2p)

   Svar: När man söker på externminne går den mesta tiden åt till den tidsödande operationen att flytta data från externminnet till primärminnet. Så även om hashtabellsoperationerna teoretiskt har komplexiteten $O(1)$ (med perfekt hashfuntkion) är det inte säkert att detta blir utslagsgivande om de inte är optimerade för att minimera antalet sådana dataförflyttningar. Det finns dock vissa balanserade träd, t ex B-träd, som är optimerade för detta ändamål. B-trädets noder kan lagra ett helt minnesblock med element, och sökningen sker genom att överföra ett sådant block i taget till primärminnet. Eftersom B-trädets noder lagrar många element blir dess höjd liten och därmed blir antalet blockförflyttningar också litet.

2. Bestäm O-komplexiteten hos följande kodfragment:

(a)
```
for (int count = 0; count < n; count++)
{
   /* instruktionsföljd med tidskomplexitet O(1) */
}
```
(2p)

Svar: Komplexiteten är $O(n)$. Instruktionsföljden tar $\leq C$ tidsenheter för något C och genomlöps n gånger. Den totala tiden är alltså $\leq Cn$.

(b)
```
for (int count = 0; count < n; count ++)
{
   for (int count2 = 1; count2 < n; count2 = count2 * 2)
      {
         /* instruktionsföljd med tidskomplexitet O(n) */
      }
}
```
(3p)

Svar: Komplexiteten är $O(n^2 \log n)$. Instruktionsföljden exekveras $n \log_2 n$ gånger.

(c)
```
for (int i = 0; i < n; ++i)
{
   for (int j = i; j < n; ++j)
      {
         /* instruktionsföljd med tidskomplexitet O(n) */
      }
}
```
(3p)

Svar: Komplexiteten är $O(n^3)$. Instruktionsföljden exekveras $n + (n-1) + \cdots + 2 + 1$ gånger, dvs $O(n^2)$ gånger.

För att få full poäng räcker det inte att du ger ett korrekt O-uttryck, utan detta uttryck måste också vara så bra som möjligt! Ditt svar måste som vanligt motiveras ordentligt.

3. (a) Mergesort använder sig av operationen `merge` som sammanflätar två sorterade fält till ett sorterat fält. Gör `merge` $O(1)$ jämförelser i bästa fall? Motivera! (2p)

   Svar: Nej, i bästa fall är komplexiteten $O(\min(m, n))$ om fälten har storleken m och n. Elementen i båda fälten måste kopieras över i resultatfältet.

   (b) Har `merge` komplexiteten $O(m + n)$ i värsta fall, om längderna av indatafälten är m och n respekive? Motivera! (2p)

   Svar: Ja, det är riktigt. I värsta fall måste man göra $m+n-1$ jämförelser.

   (c) Diskuterar huruvida `merge` är in-place eller inte! (2p)

   Svar: Den normala implementeringen är inte in-place eftersom man allokerar ett nytt resultatfält av storleken $m + n$.

   (d) Skriv Haskellprogram

```
T_best_mergesort  : Int -> Int
T_worst_mergesort : Int -> Int
```

   som räknar ut *exakt* hur många jämförelser mergesort gör i bästa fall och värsta fall. Indata är längden av listan.

   Om du vill kan du i stället skriva programmet i Java eller detaljerad pseudokod. (4p)

   Svar:

$$
\begin{aligned}
\texttt{T\_best\_merge}\ n &= n/2 \\
\texttt{T\_worst\_merge}\ n &= n-1 \\
\texttt{T\_best\_mergesort}\ n &= \texttt{T\_best\_merge}\ n + \\
&\quad \texttt{T\_best\_mergesort}\ (n+1)/2 + \texttt{T\_best\_mergesort}\ n/2 \\
\texttt{T\_worst\_mergesort}\ n &= \texttt{T\_worst\_merge}\ n + \\
&\quad \texttt{T\_worst\_mergesort}\ (n+1)/2 + \texttt{T\_worst\_mergesort}\ n/2
\end{aligned}
$$

   Observera att $n/2$ är heltalsdivision. Vi måste ta hänsyn till att n kan vara udda. Då är bästa fallet att alla element i det korta fältet är mindre än alla i det längre fältet. Vidare kommer då mergesort att dela upp fältet i två olika långa fält av längderna $(n + 1)/2$ och $n/2$.

   (e) Vilka av sorteringsmetoderna insertion sort, selection sort, mergesort och quicksort använder sig av söndra och härska ("divide and conquer") metoden? Svaret måste motiveras utförligt! (2p)

   Svar: Mergesort och quicksort använder sig av söndra och härska tekniken eftersom båda bryter ned ett problem till två delproblem som (åtminstone i bästa fall) är ungefär lika stora och kan lösas rekursivt med samma metod. I fallet quicksort kan visserligen de båda delproblemen vara olika stora men i medelfallet leder detta trots allt till logaritmisk komplexitet hos nedbrytningsprossesen. Insertion sort och Selection sort använder sig inte av söndra och härska eftersom de bara reducerar ett problem av storlek $n + 1$ till ett problem av storlek n.

4. (a) Skriv ett program som testar om ett givet binärt träd är ett AVL-träd. Du kan använda Haskell, Java eller detaljerad Java-liknande pseudokod. (6p)

   Svar: Man måste testa om det binära trädet har (i) sökträdsegenskapen (ii) är balanserat, dvs om höjden på de två delträden till varje nod skiljer sig med högst 1. Sökträdsegenskapen kan kontrolleras genom att göra en inordertraversering tillsammans med en test att den resulterande listan är sorterad. Pseudokod för detta finns i boken.

   Balanseringen kontrolleras genom att först beräkna höjden på alla delträd och sedan jämföra höjdskillnaderna mellan delträden till varje nod.

   Här är Haskellkod.

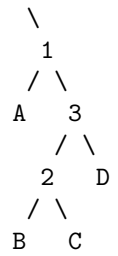
```
isBalanced Empty = True
isBalanced (Branch l a r)
= isBalanced l && isBalanced r && abs (height l - height r) < 2

height Empty = 0
height (Branch l a r) = max (height l) (height r) + 1

inOrder Empty = []
inOrder (Branch l a r) = inOrder l ++ ([a] ++ inOrder r)

isSorted [] = True
isSorted [a] = True
isSorted (a:b:as) = a <= b && isSorted b:as

isBST t = isSorted (inOrder t)

isAVL t = isBalanced t && isBST t
```

   (b) Vilken O-komplexitet har ditt program? För full poäng krävs att programmet har optimal O-komplexitet för uppgiften. (2p)

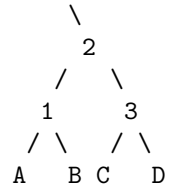
   Svar: Om trädet har n noder, så tar inordertraverseringen tillsammans med sorteringskontrollen $O(n)$. Även höjdberäkningarna tillsammans med höjdjämförelserna tar $O(n)$, eftersom både höjdberäkningen och jämförelsen görs rekursivt för varje nod, och arbetet för varje nod är $O(1)$.

(c) När man sätter in ett element från ett AVL-träd uför man två steg. Först gör man den vanliga operationen för insättning i binära sökträd. Sedan gör man en ombalansering av trädet genom en enkel- eller dubbelrotation. Hur många pekare behöver ändras i värsta fall för denna ombalansering? Motivera! (2p)

Svar: Värsta fallet är en dubbelrotation. Då behöver man ändra 5 pekare, inklusive pekaren till roten på det delträd som ombalanseras. Antag att vi har strukturen



före dubbelrotationen. Då får vi



efteråt. Alla pekare i figuren utom 1 - A och 3 - D måste alltså ändras.

5. En oriktad graf kan implementeras som en riktad graf som är *symmetrisk*, dvs om det finns en båge från nod a till nod b så finns det också en båge från nod b till nod a. Vi förutsätter att det inte finns några parallella bågar i den oriktade grafen, dvs det finns högst en oriktad båge mellan varje nodpar a och b.

   (a) Antag först att din riktade graf är implementerad som en grannmatris. Skriv en algoritm i pseudokod som avgör om den riktade grafen är symmetrisk (dvs om den representerar en oriktad graf)! Analysera tidskomplexiteten hos din algoritm! Du ska ange O-komplexiteten beroende på antalet noder n och antalet bågar m. (4p)

   Svar: Här är Javakod för en metod som kontrollerar om en grannmatris `m` med n noder, är symmetrisk:

```
boolean isSymmetric(boolean[][] m) {
   int n = m.length;
   for(i = 0; i < n; i++)
      for(j = 0; j < n; j++)
         if (m[i][j] != m[j][i]) return false;
   return true;
}
```

   Den asymptotiska komplexiteten är $O(n^2)$.

   (b) Antag sedan att din riktade graf i stället är implementerad med hjälp av grannlistor och besvara samma frågor som ovan. Dvs skriv en algoritm i pseudokod som avgör om den riktade grafen är symmetrisk (dvs om den representerar en oriktad graf) och analysera O-komplexiteten beroende på antalet noder n och antalet bågar m. (6p)

   Svar: Vi antar att noderna är numrerade och att grafen är implementerad som ett fält av länkade listor. Man konverterar först grannlistorna till en grannmatris. Sedan använder man algoritmen i (a).

   Om grafen har m bågar tar konverteringen $O(m+n)$ eftersom man genomlöper alla noderna och deras grannar. Totalt tar algoritmen $O(m+n^2) = O(n^2)$.

   Om grafen är gles så kan det vara bättre att direkt leta efter inversa bågar. För varje båge från u till v söker man då upp v's grannlista och kontrollerar om u finns i den. Komplexiteten hos denna operation beror på hur grannlistorna är implementerade.

6. Den ungerske matematikern Paul Erdös (1913-96) skrev totalt 1475 matematiska uppsatser, fler än någon annan i historien. Många av dessa uppsatser skrevs tillsammans med andra: han samarbetade med totalt 511 matematiker. Man säger att dessa 511 har Erdösnummer 1. Vidare säger man att en matematiker har Erdösnummer 2 om han eller hon författat artiklar tillsammans med en matematiker med Erdösnummer 1. Mer allmänt säger man att en matematiker som författat artiklar tillsammans med en matematiker med Erdösnummer n har Erdösnumret $n+1$. (Notera att Erdösnumret är det lägsta tal som tilldelas på detta sätt. Om t ex en matematiker författat artiklar både med Erdös och med en medförfattare till Erdös har alltså matematikern Erdösnumret 1, inte 2.)

   (a) Antag att du har en databas med matematiska artiklar och information om deras författare. Hur kan man skriva ett program som räknar ut en viss matematikers Erdösnummer? Om matematikern inte har något Erdösnummer ska programmet returnera "No Erdös number". Skriv ett program i pseudokod! Om du använder algoritmer och datastrukturer som du lärt dig i kursen behöver du inte ge pseudokod för dem. (7p)

   Svar: Först genererar man en graf från artikeldatabasen. Varje författare i databasen representeras av en nod, och det finns en båge mellan författarna u och v omm det finns en artikel där både u och v är medförfattare. Sedan använder man BFS för grafer utgående från noden Erdös. Precis som på föreläsningarna tilldelar BFS en nivå (avstånd till Erdös) för varje nod och detta är Erdösnumret. Noder som inte nås av BFS med startnod Erdös saknar Erdösnummer, och för dem returnerar algoritmen "No Erdös number".

   (b) Vilken O-komplexitet har din algoritm uttryckt som funktion av antalet artiklar och antalet författare i databasen? För full poäng ska komplexiteten vara optimal för detta programmeringsproblem! (3p)

   Svar: Algoritmen BFS har komplexiteten $O(m+n)$ om n är antalet noder (matematiker) och m är antalet bågar i grafen. Notera att $O(m)$ inte är korrekt eftersom vi även måste tilldela "No Erdös number" till alla författare som inte finns i samma sammanhängande komponent som Erdös.