Tentamen
Datastrukturer D
DAT 035/INN960
(med kortfattade lösningar)

19 december 2008

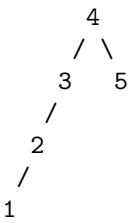
- Tid: 8.30 - 12.30
- Ansvarig: Peter Dybjer, 0736-341480 (innan 10.30). Håkan Burden, 0730-77287777, och Staffan Björensjö (after 10.30).
- Max poäng på tentamen: 60.
- Betygsgränser, CTH: 3 = 24 p, 4 = 36 p, 5 = 48 p, GU: G = 24 p, VG = 48 p.
- Hjälpmedel: *handskrivna* anteckningar på *ett* A4-blad. 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. Antag att du sätter in elementen 4, 3, 2, 1, 5 ett efter ett (i denna ordning) i följande datastrukturer (mha standardalgoritmerna för insättning)

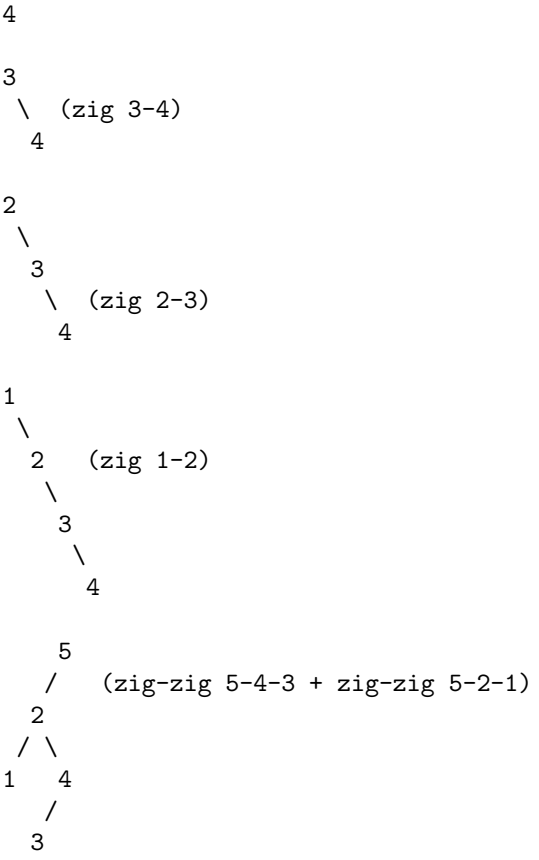
   (a) Vanligt binärt sökträd, utan balansering. (1p)

   Svar:



   (b) Splayträd. (3p)

   Svar:



   (c) (2,4)-träd. Kom ihåg att i dessa träd har alla inre noder mellan 2 och 4 barn. Alla löv ligger på samma djup så trädet är perfekt balanserat. Varje

nod lagrar mellan 1 och 3 element, och trädet har en sökträdsegenskap som generaliserar det binära sökträdets. (3p)

Svar: Först sätter man in de tre första elementen i rotnoden. När man sedan sätter in det fjärde elementet får man overflow i rotnoden:

```
1,2,3,4
```

Alltså splittrar man den och får

```
  2
 / \
1  3,4
```

(Man kan också lägga 3 i rotnoden, osv.) Slutligen får man

```
  2
 / \
1  3,4,5
```

(d) Heap (en vanlig binär "min-heap" som lagrar minsta elementet i roten). (3p)

Svar:

```
4

  3
 /    (bubbla upp 3)
4

  2
 / \   (bubbla upp 2)
4   3

    2
   / \   (5 hamnar på rätt plats från början)
  4   3
 /
5
```

Rita hur träden byggs upp steg för steg efter varje insättning och förklara vad som händer! I (a) räcker det dock att du ritar det slutgiltiga binära sökträdet.

2. Avgör om vart och ett av följande påståenden är sant eller falskt! Endast svar med korrekt motivering godtages.

   (a) Om en algoritm har den asymptotiska komplexiteten $O(1)$ så tar den alltid lika lång tid att exekvera oavsett indata. (2p)

   Svar. Nej, $O(1)$ betyder bara att det finns en konstant C så att $T(n) \leq C$ för alla $n < n_0$. Av detta följer inte att $T(n) = C$ för alla n för någon konstant C.

   (b) Höjden på ett binärt sökträd med n noder är alltid $O(\log n)$. (2p)

   Svar. Nej, höjden på ett binärt sökträd kan i värsta fall vara $O(n)$ om det är helt obalanserat.

   (c) Det gäller alltid att $O(\log_2 n) = O(\log_4 n)$. (2p)

   Svar. Ja, eftersom $\log_2 n = 2 \log_4 n$ så gäller att om $T(n) \leq C \log_2 n$ så $T(n) \leq 2C \log_4 n$ och om $T(n) \leq C \log_4 n$ så $T(n) \leq (C/2) \log_2 n$.

   (d) Höjden på ett (2,4)-träd som lagrar n element är alltid $O(\log n)$. (2p)

   Svar. Ja. Värsta fallet är när varje nod bara lagrar ett element och alltså är ett binärt fullständigt balanserat träd. Ett träd med höjden h då lagra $n = 2^{h+1} - 1$ element. Alltså är $2^h \leq n$, dvs $h \leq \log n$. Så h är $O(\log n)$.

   (e) Man kan sortera ett fält på tiden $O(n \log n)$ i medeltal genom att först sätta in elementen i fältet ett efter ett i en skipplista, och sedan plocka ut dem i ordning från understa raden i skipplistan. (2p)

   Svar. Ja, den understa raden i en skipplista är sorterad. Varje insättning tar $O(\log n)$ i medeltal och varje utplockning tar $O(1)$. Insättningarna kommer alltså att dominera tidsåtgången vilken alltså blir $O(n \log n)$ i medeltal.

3. (a) Följande program har som indata ett fält `A` av heltal.

```
int f(int[] A) {
  int n = A.length;
  int s = 0;
  for (int i = 0; i < n; i++) {
    s = s + A[0];
    for (int j = 1; j <= i; j++) s = s + A[j]
  }
  return s
```

Vilken är O-komplexiteten hos programmet uttryckt som funktion av `n`, längden hos indatafältet `A`? Motivera! För full poäng ska O-komplexiteten vara en "skarp" uppskattning. (3p)

Svar. Den yttre loopen körs `n` gånger och den inre loopen körs $0 + 1 + \cdots + (n-1) = n(n-1)/2$ gånger. Eftersom varje tilldelning till s tar $O(1)$ kommer den totala exekveringstiden av den inre loopen att dominera och vara $O(n^2)$.

(b) Följande program har som indata ett fält `A` av heltal $<$ `N`.

```
void f(int[] A, int N){
  int n = A.length;
  int[] B = new int[N];
  for (int j = 0; j < N; j++) B[j] = 0;
  for (int i = 0; i < n; i++) B[A[i]]++;
  for (int j = 0; j < N; j++) {
     int i = 0;
     while(B[j] > 0){
        A[i++] = j;
        B[j]--;
     }
  }
  return;
}
```

Vilken är algoritmens O-komplexitet uttryckt som funktion av fältets storlek `n` och `N`? Motivera! För full poäng ska O-komplexiteten vara en "skarp" uppskattning. (3p)

Svar. Den första for-slingan tar $O(\mathtt{N})$, den andra $O(\mathtt{n})$ och den tredje $O(\mathtt{N} + \mathtt{n})$, eftersom den körs `N` gånger och den inre while-slingan tar $O(\mathtt{n})$ totalt. Alltså blir den totala tidsåtgången $O(\mathtt{N} + \mathtt{n})$.

(c) Är det korrekt att programmet sorterar `A`? Om ja, motivera! Om nej, lokalisera felet! (4p)

Svar. Programmet ifråga är nästan "counting sort", en variant av "bucket sort" där man sorterar heltal genom att räkna antalet förekomster i indatafältet. Det har dock smugit sig in ett fel, genom att variabeln `i` nollställs varje gång i den tredje slingan. På grund av detta kommer t ex

`A[0]` att skrivas över varje gång while-slingan påbörjas. Om indata fältet exempelvis är `1,2` kommer det att bli `2,2` efter exekveringen. I stället ska `i` initialiseras utanför while-slingan.

4. Följande grannmatris representerar en viktad oriktad 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. (1p)

(b) Man kan använda Dijkstras algoritm för att finna kortaste vägen mellan två noder i en graf. Som i lab 3 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. (4p)

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. (3p)

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. (2p)

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.

5. Klassen `ArrayList` i Java Collections kan lagra godtyckligt stora listor. Här är ett utdrag ur beskrivningen av klassen

> Resizable-array implementation of the `List` interface. ... The `size, isEmpty, get, set, iterator,` and `listIterator` operations run in constant time. The add operation runs in amortized constant time, ... Each `ArrayList` instance has a capacity. The capacity is the size of the array used to store the elements in the list. It is always at least as large as the list size. As elements are added to an `ArrayList`, its capacity grows automatically.

(a) Förklara betydelsen av "The add operation runs in amortized constant time"! (2p)

Svar. Det betyder att tiden det tar att utföra n add-operationer i följd efter varandra är $O(n)$, dvs i mdeletal tar var och en av dessa operationer $O(1)$.

(b) Visa hur man lämpligen kan implementera `ArrayList` genom att skriva en klass i Java eller i detaljerad pseudokod. Det räcker om du visar vilka tillståndsvariabler ("fields") man behöver samt hur man implementerar följande operationer beskrivna i "constructor summary" och "method summary":

i. `ArrayList()` - Constructs an empty list with an initial capacity of ten.

ii. `E get(int index)` - Returns the element at the specified position in this list.

iii. `void add(E o)` - Appends the specified element to the end of this list.

iv. `void add(int index, E element)` - Inserts the specified element at the specified position in this list.

(8p)

Svar.

```
public class ArrayList <E> {
        // Elements are stored in es, with current
        // capacity available as es.length
        private E[] es;
        // The number of occupied entries in the es array.
        private int size;

        public ArrayList(){
                // The initial capacity is 10, as specified
                es = (E[])new Object[10];
                size = 0;
        }
```

```
        public E get(int index) {
                // Check that the index is valid
                if (index >= size || index < 0)
                   throw new IndexOutOfBoundsException();
                // Get the element directly for the array
                return es[index];
        }

        public void add(E e) {
                // The position size is after the last object
                add(size, e);
        }

        public void add(int index, E e) {
                // Check that the index is valid
                if (index > size || index < 0)
                        throw new IndexOutOfBoundsException();

                // Double the capacity if no more room
                if (es.length == size) {
                        E[] news = (E[])new Object[2*size];
                        for (int i = 0; i < size; i++)
                                news[i] = es[i];
                        es = news;
                }

                // Shift elements to make room for the new element
                for (int i = size; i > index; i--)
                        es[i] = es[i-1];

                // Insert the new element and increase the size
                es[index] = e;
                size++;
        }
}
```

6. En min-max-heap är en datastruktur med effektiva metoder både för att ta ut minsta och största elementet i en datasamling. Precis som en vanlig heap är den ett fullständigt binärt träd ("complete binary tree"), men elementen i en min-max-heap är ordnade på ett annat sätt. Alla noder som ligger på jämnt djup (exempelvis roten) har nycklar som är mindre än (eller lika med) sina ättlingar, medan alla noder som ligger på udda djup har söknycklar som är större än (eller lika med) sina ättlingar. (En ättling till noden n är en nod som finns i delträdet med roten n.)

   (a) Rita en min-max-heap som innehåller elementen 1,2,3,4,5,6,7,8,9,10! Observera att det finns många sådana min-max-heapar. (2p)

   (b) Hur implementeras operationen `max` som returnerar det största elementet i min-max-heapen (utan att ta bort det)? Vilken är den asymptotiska tidskomplexiteten hos `max`. (3p)

   (c) Hur implementeras operationen `deleteMin` som tar bort det minsta elementet i en min-max-heap? Visa resultatet av att utföra `deleteMin` på min-max-heapen i (a). Asymptotisk komplexitet hos `deleteMin`? (5p)

   Du ska beskriva implementeringarna här med hjälp av pseudokod. (Javakod accepteras också.)

   Observera att i t.ex. G & T är heapar en datastruktur för lexika där man lagrar par av söknycklar och element. Men man kan förstås även lagra datasamlingar där söknyckeln och elementet är samma sak.