Tentamen
Datastrukturer för D2
TDA 131

lördagen den 22 december 2001

- Tid: 8.45 - 12.45
- Ansvarig: Peter Dybjer, tel 7721035 eller 405836
- Max poäng: 60
- Betygsgränser CTH: 3 = 30 p, 4 = 40 p, 5 = 50 p
- Hjälpmedel: Kursboken Timothy Budd, Classic Data Structures in Java, Addison-Wesley. (Alternativt får boken Mark Allen Weiss, Datastructures and Algorithm Analysis i Java, Addison-Wesley användas. Dock får endast *en* av dessa båda böcker användas.)
- Skriv tydligt och disponera papperet på ett lämpligt sätt.
- Börja varje ny uppgift på nytt blad.
- Skriv endast på en sida av papperet.
- Alla svar ska motiveras väl.
- Lycka till!

1. (a) Vilka av följande påståenden är sanna?

   i. Funktionen $f(n) = n^2 + n + 1$ är $O(n^2)$.
   ii. Funktionen $f(n) = n^2$ är $O(n^2 + n + 1)$.
   iii. Funktionen $f(n) = n \log n$ är $O(n + \log n)$.
   iv. Funktionen $f(n) = n + \log n$ är $O(n \log n)$.
   v. Funktionen $f(n) = \sqrt{n}$ är $O(\log n)$.
   vi. Funktionen $f(n) = \log n$ är $O(\sqrt{n})$.

   Detaljerade motiveringar krävs för full poäng – delpoäng ges dock för korrekt svar även utan motivering. (6p)

   (b) Följande kodavsnitt i Java beräknar en serie av n summor $S_j = \sum_{i=0}^{j} a_i$ för $j = 0, \ldots, n-1$.

```
for (int j = n - 1; j >= 0; --j) {
    int sum = 0;
    for (int i = 0; i <= j; ++i)
        sum += a[i];
    a[j] = sum;
}
```

   Indata är alltså ett fält `a` med n element. Programmet beräknar sedan de n summorna S_j och lagrar dem i samma fält `a`.

   Hur beror exekveringstiden av detta kodavsnitt på n uttryckt med hjälp av O-notation? Ordentliga motiveringar ska ges. För att få full poäng ska O-komplexiteten vara så bra som möjligt – det ska vara en "skarp" uppskattning. (Man använder ibland notationen $\Theta(g(n))$ för sådana skarpa uppskattningar). (6 p)

2. Din uppgift är att implementera en abstrakt datatyp för listor/stackar som man kan utföra operationen `append` på. En sådan abstrakt datatyp kan representeras av följande Java-gränssnitt:

```
public interface List {
    public void nil ();
    public boolean isEmpty ();
    public void push (Object a);
    public Object top ();
    public void pop ();
    public void append (List bs);
}
```

Ett `List`-objekt ska kunna representera en lista/stack `as` av godtycklig längd m. När man anropar metoden `nil()` ska en tom lista lagras i objektet. När man anropar metoden `push(a)` ska ett nytt element `a` läggas till i början av listan. När man anropar metoden `top()` ska första elementet i listan returneras och när man anropar `pop()` ska första elementet tas bort från listan. När man slutligen anropar metoden `append(bs)` ska den konkatenera listan `as` med listan `bs` och lagra den i objektet.

Om `as` $= 1, 2, 3$ och man gör `pop` blir `as` $= 2, 3$ och `top` returnerar elementet 1. Om man gör `push` på ett objekt som innehåller talet 4 blir `as` $= 4, 1, 2, 3$ och om man gör `append` med listan `bs` $= 4, 5$ så blir `as` $= 1, 2, 3, 4, 5$ efter konkateneringen.

(a) Din uppgift är att skriva en Java-klass

```
public class SinglyLinked implements List { ... }
```

som implementerar gränssnittet med hjälp av enkellänkade listor som har pekare både till första och sista elementet i listan (se t ex Budd sid 185 längst ned till höger). Om listan är tom ska båda pekarna vara nollpekare `null`. (8 p)

(b) Rita en figur som visar vilken enkellänkad lista enligt ovan som representerar listan/stacken $1, 2, 3$. (2 p)

(c) Ange vilken asymptotisk tidskomplexitet dina metoder har uttryckt som funktion av längden m av listan `as`, och i fallet med `append` uttryckt som funktion som eventuellt kan bero både på längden m av listan `as` och längden n av listan `bs`. Som vanligt ska du förstås använda dig av O-notation, och för att få full poäng ska dina metoder ha optimal tidskomplexitet. (3 p)

3. Antag att du vill lagra heltal så att du kan söka efter dem på ett effektivt sätt. För detta ändamål kan man t ex använda hashtabeller, AVL-träd eller skip-lists.

   För att illustrera hur dessa datastrukturer fungerar får du nu i uppgift att lagra talen 12, 44, 13, 88, 23, 94, 11, 39, 20, 16 och 5.

   (a) Antag först att du lagrar talen i en hashtabell med 11 celler och använder hashning med hinkar ("hashing using buckets"). Din hashfunktion är

   $$h(i) = (2i + 5) \bmod 11$$

   Du ska nu rita den hashtabell som du får genom att sätta in talen ovan i ordning. (3 p)

   (b) I den andra varianten ska du också använda hashtabeller, men i stället använda öppen adressering när kollisioner uppstår. Använd samma hashfunktion som ovan och den enklaste varianten av "linear probing" där man väljer nästa lediga cell (mod 11). I det här fallet ska du också rita den slutgiltiga hashtabellen men här är det viktigt att du motiverar i detalj varför elementen ligger där de ligger. (3 p)

   (c) I den tredje varianten ska du lagra talen i ett AVL-träd. Du börjar med ett tomt AVL-träd och sätter in talen ett efter ett med hjälp av standardalgoritmen för insättning i AVL-träd. Denna algoritm har två steg: (i) använd algoritmen för insättning av ett nytt element som ett löv i ett binärt sökträd; (ii) om det resulterande trädet inte är ett AVL-träd (dvs höjdbalanserat) så utför "rotationer/trenodsomstrukturering" så att trädet ånyo blir balanserat.

   Din uppgift är att förklara hur AVL-trädet med ovanstående 11 element successivt byggs upp. Du behöver inte rita alla 11 stegen – det räcker att du ritar de steg där trädet behöver ombalanseras. Det är dock viktigt att du förklarar principen bakom ombalanseringen. (Tänk på att det är lätt att göra fel!) (5 p)

   (d) I den fjärde varianten ska du bygga en skip-lista. Här räcker det att du ritar den slutgiltiga skip-listan som innehåller 11 element. Eftersom insättningsalgoritmen för skip-listor behöver slumptal för att avgöra om ett visst element ska sättas in i "omkörningsfilen, omomkörningsfilen, ..." får du här en serie slumptal:

   $$0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, \ldots.$$

   Här betyder 1 att elementet ska vara med i omkörningsfilen, etc, och 0 betyder att det inte ska det. (3 p)

Som alltid är det viktigt att du motiverar dina lösningar, inte minst för att det gör det möjligt att ge delpoäng för ej helt korrekta lösningar!

4. (a) Skriv en algoritm som avgör om ett givet binärt träd har heap-egenskapen, dvs att varje förälder är mindre än sina barn enligt en viss given ordning! Det räcker här om du skriver "pseudokod", som dock måste vara tillräckligt detaljerad för att kodningen i Java ska bli rutinmässig. (7 p)

   (b) Vilken tidskomplexitet har ditt program? Motivera! För full poäng på uppgiften ska den asymptotiska komplexiteten vara så bra som möjligt. (4 p)

5. Antag att du exekverar de fyra sorteringsalgoritmerna insättningssortering (insertion sort), mergesort, quicksort och countingsort (bucketsort) på

   (a) fält med identiska element, t ex $1, 1, 1, 1, 1, 1$;

   (b) fält med alternerande element, t ex $1, 2, 1, 2, 1, 2$.

   Hur beror exekveringstiden på indatastorleken n uttryckt med hjälp av O-notation?

   Analysen av quicksort beror på implementeringsdetaljer - hur man väljer pivotelement och vad man gör när man jämför två lika element. I den här uppgiften ska du anta att du har en naiv variant av quicksort där man alltid väljer första elementet i ett osorterat delfält som pivotelement p, och sedan lagrar alla element $< p$ före p och alla element $\geq p$ efter p. (10 p)