

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 $j = 0, \dots, n - 1$.

```
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 ritat 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 ritat den slutgiltiga skip-listan som innehåller 11 element. Eftersom insättningsalgoritmen för skip-listor behöver slumpal för att avgöra om ett visst element ska sättas in i “omkörningsfilen, omomkörningsfilen, ...” får du här en serie slumpal:

0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, . . .

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 insättningsortering (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)

Lösningar till tentamen i
datastrukturer för D2
TDA 131

lördagen den 22 december 2001

1. Kom ihåg att $f(n)$ är $O(g(n))$ om det finns positiva konstanter c, n_0 så att $f(n) \leq cg(n)$ för alla $n \geq n_0$.
- (a) i. Funktionen $f(n) = n^2 + n + 1$ är $O(n^2)$ eftersom $n^2 + n + 1 \leq 3n^2$ för alla $n \geq 1$. Konstanterna i definitionen ovan kan alltså väljas till $c = 3$ och $n_0 = 1$.
- ii. Funktionen $f(n) = n^2$ är $O(n^2 + n + 1)$ eftersom $n^2 \leq n^2 + n + 1$ för alla $n \geq 0$. Här är $c = 1$ och $n_0 = 0$.
- iii. Funktionen $f(n) = n \log n$ är *inte* $O(n + \log n)$. Vi behöver visa att för alla positiva c, n_0 finns det $n \geq n_0$ så att $n \log n > c(n + \log n)$. Men eftersom $n + \log n < 2n$ räcker det med att hitta $n \geq n_0$ så att $n \log n > 2cn$. Detta gäller om $\log n > 2c$ vilket är fallet om $n > 2^{2c}$.
- iv. Funktionen $f(n) = n + \log n$ är $O(n \log n)$ eftersom $n + \log n < 2n \leq n \log n$ om $2 \leq \log n$, dvs om $n \geq 4$. Här är $c = 1$ och $n_0 = 4$.
- v. Funktionen $f(n) = \sqrt{n}$ är *inte* $O(\log n)$. Vi behöver visa att för alla positiva c, n_0 finns det $n \geq n_0$ så att $\sqrt{n} > c \log n$. Men om vi sätter $k = \sqrt{n}$ ser vi att detta är ekvivalent med att finna $k \geq k_0 = \sqrt{n_0}$ så att $k > c \log k^2 = 2c \log k$. Men detta kan vi göra eftersom k inte är $O(\log k)$.
- vi. Funktionen $f(n) = \log n$ är $O(\sqrt{n})$ eftersom $\log n = 2 \log \sqrt{n} < 2\sqrt{n}$ eftersom $\log k < k$. Välj alltså $c = 2$ och $n_0 = 0$ t.ex.
- (b) Den yttre for-loopen exekveras n gånger och den inre exekveras

$$n + (n - 1) + \dots + 2 + 1 = \frac{n(n + 1)}{2}$$

gånger. Antag att det tar k_0 tidsenheter att terminera den yttre loopen, maximalt k_1 tidsenheter att exekvera varje iteration i den yttre loopen (inklusive den tid det tar att terminera den inre loopen), samt k_2 tidsenheter att exekvera satserna i den inre loopen där k_0, k_1 och k_2 är konstanter oberoende av n . Alltså är sammanlagda tidsåtgången

$$f(n) \leq k_0 + k_1 n + k_2 \frac{n(n + 1)}{2}$$

Men $f(n)$ är $O(n^2)$ eftersom

$$k_0 + k_1 n + k_2 \frac{n(n + 1)}{2} \leq (k_0 + k_1)n^2 + \frac{k_2}{2}(4n^2) = (k_0 + k_1 + 2k_2)n^2$$

om $n \geq 1$.

```

2. (a) import java.util.NoSuchElementException;

public class Link {
    public Object elem;
    public Link next;

    public Link (Object a, Link l) {
        elem = a;
        next = l;
    }
}

public class SinglyLinked implements List {
    private Link first;
    private Link last;

    public void nil () {
        first = last = null;
    }

    public void push (Object a) {
        first = new Link(a,first);
        if (last == null) last = first;
    }

    public Object top () {
        if (isEmpty()) throw new NoSuchElementException();
        return first.elem;
    }

    public void pop () {
        if (isEmpty()) throw new NoSuchElementException();
        first = first.next;
        if (first == null) last = null;
    }

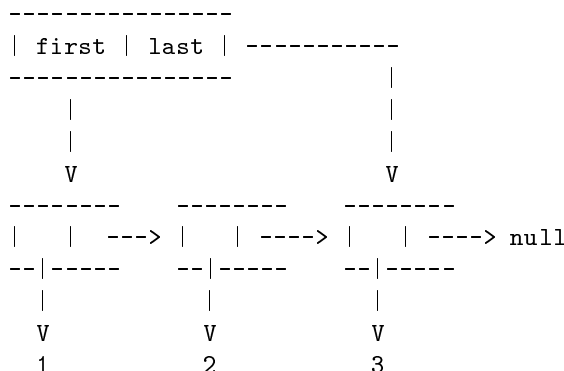
    public boolean isEmpty () {
        return first == null;
    }

    public void addLast (Object a) {
        Link aLink = new Link(a,null);
        if (isEmpty()) {first = last = aLink;}
        else {last.next = aLink; last = last.next;}
    }

    public void append (List bs) {
        while (! bs.isEmpty()) {
            addLast(bs.top());
            bs.pop();
        }
    }
}

```

(b)



- (c) Alla operationerna utom `append` i gränssnittet är $O(1)$. Varje operation har en maximal exekveringstid som är oberoende av m och n .
Exekveringstiden för `append` är $O(n)$ eftersom den gör en iteration för varje element i listan `bs`. Exekveringstiden för varje iteration är $O(1)$ under förutsättning att exekveringstiderna för `top` och `pop` på `bs` är oberoende av $O(1)$

3. (a) Vi får följande hashkoder $h(i)$ för de 11 värdena på i

i : 12 44 13 88 23 94 11 39 20 16 5

$h(i)$: 7 5 9 5 7 6 5 6 1 4 4

Om vi använder hashning med hinkar får vi alltså följande innehåll i hinkarna

```
          11
         5 88 39 23
        20      16 44 94 12      13
```

hink : 0 1 2 3 4 5 6 7 8 9 10

Hinkarna har här fyllts på nerifrån och upp.

- (b) Med öppen hashning får vi följande händelseförlopp:

12 - läggs direkt i cell 7.

44 - läggs direkt i cell 5.

13 - läggs direkt i cell 9.

88 - cell 5 är redan använd, nästa lediga är cell 6.

23 - cell 7 är redan använd, nästa lediga är cell 8.

94 - cell 6 är redan använd, nästa lediga är cell 10.

11 - cell 5 är redan använd, nästa lediga är cell 0.

39 - cell 6 är redan använd, nästa lediga är cell 1.

20 - cell 1 är redan använd, nästa lediga är cell 2.

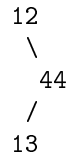
16 - läggs direkt i cell 4.

5 - cell 4 är redan använd, nästa lediga är cell 3.

- (c) När man sätter in element i ett AVL-träd använder man sig först av algoritmen för insättning i ett allmänt binärt sökträd. Om det resulterande binära sökträdet ej är höjdbalanserat genomför man trenodsomstruktureringar för att återställa balansen och få ett korrekt AVL-träd.

Vi gör successiva insättningar och visar bara de situationer då vi behöver återställa balansen.

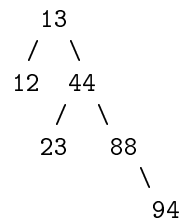
Efter insättning av 12, 44 och 13 får vi följande träd:



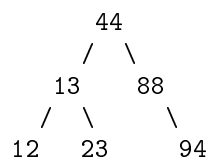
Trenodsomstrukturering (av 12-44-13) ger:



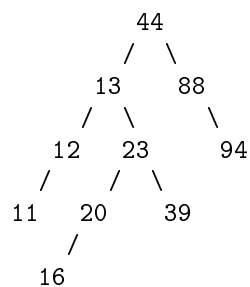
Efter insättning av 88, 23 och 94 får vi:



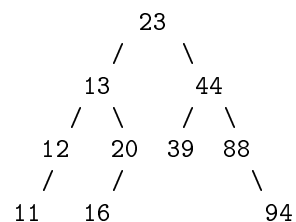
Trenodsomstrukturering av 13-44-88 ger:



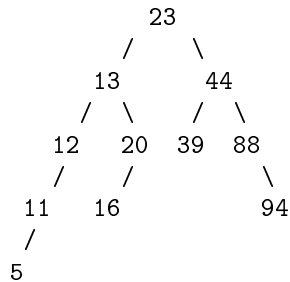
Efter insättning av 11, 39, 20 och 16 får vi:



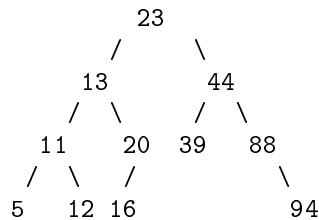
Trenodsomstrukturering av 44-13-23 ger:



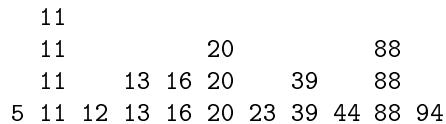
Slutligen sätter vi in 5 och får:



Trenodsomstrukturering av 12-11-5 ger:



(d) Vi får följande skip-lista



Se Budd sid 224-225 för en förklaring av pekarstrukturen som också ska finnas med i ett korrekt svar.

4. (a) En rekursiv definition ser ut så här.

Ett binärt träd har heapegenskapen omm (i) vänstra delträdet är tomt *eller annars* om roten är mindre än eller lika med sitt vänstra barn och vänstra delträdet har heapegenskapen *samt* om (ii) högra delträdet är tomt *eller annars* om roten är mindre än eller lika med sitt högra barn och högra delträdet har heapegenskapen.

Vi visar här hur man kan göra en implementering i Java av denna definition förutsatt att vi har följande gränssnitt för en abstrakt datatyp av binära träd med heltal i noderna.

```

interface BinTree {
    public int root ();
    public BinTree left ();
    public BinTree right ();
    public boolean isEmpty ();
}

```

Då kan vi skriva isHeap-programmet på följande sätt

```

public class isHeap {
    public boolean isHeap (BinTree t) {
        if (t.isEmpty()) return true;
        BinTree tleft = t.left();
        BinTree tright = t.right();
        boolean leftOk = true;
        boolean rightOk = true;
    }
}

```

```

    if (! tleft.isEmpty())
        leftOk = t.root() <= tleft.root() && isHeap(tleft);
    if (! tright.isEmpty())
        rightOk = t.root() <= tright.root() && isHeap(tright);
    return leftOk && rightOk;
}
}

```

(Koden testar som sig bör heap-egenskapen med avseende på \leq i stället för $<$ som det står i uppgiften.)

Svar i pseudokod som t ex approximerar ovanstående kod väl godkänns också.

- (b) Exekveringstiden för $\text{isHeap}(t)$ är $O(n)$ om n är antalet noder i trädet. Man ser att tiden $f(n)$ för att beräkna $\text{isHeap}(t)$ är summan av den tid det tar att utföra de två rekursiva anropen och den maximala tid k (oberoende av n) det tar att exekvera övriga instruktioner. Vi har här förutsatt att alla operationerna i gränssnittet är implementerade så att deras exekveringstid är $O(1)$. Därför är $f(n) \leq kn$.

5. (a) Listor med identiska element.

Insertion sort är $O(n)$ eftersom listan redan är sorterad och vi behöver därför bara utföra en jämförelse vid varje insättning, dvs en iteration av den inre loopen.

Merge sort är $O(n \log n)$. Analysen av mergesort är likadan som om indata är en godtycklig lista. (Se sid 163-165 i Budd.)

Quicksort är $O(n^2)$. Pivotelementet är 1 och alla andra element kommer att hamna i den högra partitionen ≥ 1 . Därför reduceras problemet att sortera n element bara till att sortera $n - 1$ element. Tidskomplexiteten ges därför av en aritmetisk serie liknande den i uppgift 1(b) och är alltså $O(n^2)$.

Counting sort är $O(m+n)$ (där m var antalet värden som kan förekomma i indata) oberoende av hur indata ser ut. Om vi dessutom antar att m är en konstant ≥ 1 så kan vi förenkla svaret till $O(n)$.

- (b) Listor med n alternerande element. Vi antar att n är ett jämt tal.

Insertion sort är $O(n^2)$. Vi utför här

$$1 + 1 + 2 + 1 + 3 + \dots + 1 + \frac{n}{2} = \frac{n}{2} + \sum_{i=1}^{\frac{n}{2}} i$$

jämförelser (iterationer av den inre loopen). Antalet jämförelser kan alltså bestämmas till $O(n^2)$ efter ett liknande resonemang som i 1(b).

Merge sort är $O(n \log n)$. Analysen av mergesort är likadan som om indata är en godtycklig lista.

Quicksort är $O(n^2)$. Här kan det vara till hjälp att illustrera med ett fall, t ex $n = 8$. Vi visar de två första mellanresultaten efter partitioneringarna och markerar pivotelementen med p:

```

1p 2  1  2  1  2  1  2
1  1  1  1 2p 2  2  2

```

Vi har här alltså reducerat problemet till att sortera två listor med längden 3 med identiska element. I allmänhet reducerar alltså denna version av quicksort problemet att sortera en lista med n alternerande

element till problemet att sortera två listor med vardera $\frac{n}{2} - 1$ identiska element. Eftersom varje sådan sortering är $O(n^2)$ enligt (a) blir svaret $O(n^2)$.

Counting sort är $O(m + n)$. Om vi dessutom antar att m är en konstant ≥ 2 så kan vi förenkla svaret till $O(n)$.