Lösningar till tentamen i datastrukturer för D2 TDA 131

lördagen den 22 december 2001

1. Kom ihåg att $f(n)$ är $O(g(n))$ omm 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

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

   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

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

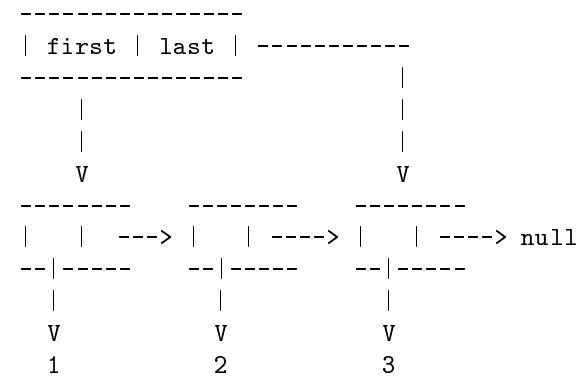
   Men $f(n)$ är $O(n^2)$ eftersom

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

   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
```

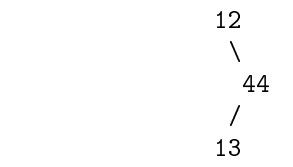
Hinkarna har här fyllts på nerifrån och upp.

(b) Med öppen hashning får vi följande händelseförlopp:

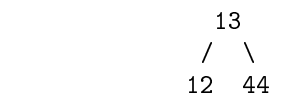
12 - läggs direkt i cell 7.
44 - läggs direkt i cell 5.
13 - läggs direkt i cell 9.
88 - cell 5 är redan använd, nästa lediga är cell 6.
23 - cell 7 är redan använd, nästa lediga är cell 8.
94 - cell 6 är redan använd, nästa lediga är cell 10.
11 - cell 5 är redan använd, nästa lediga är cell 0.
39 - 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.

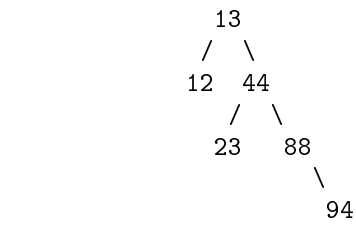
Vi gör successiva insättningar och visar bara de situationer då vi behöver återställa balansen.
Efter insättning av 12, 44 och 13 får vi följande träd:



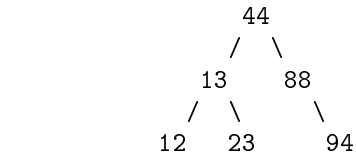
Trenodsomstrukturing (av 12-44-13) ger:



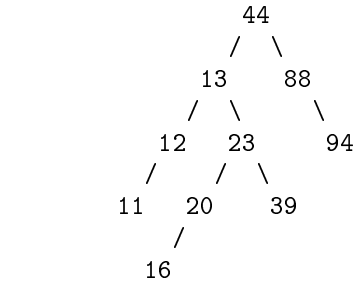
Efter insättning av 88, 23 och 94 får vi:



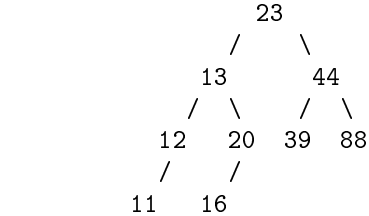
Trenodsomstrukturering av 13-44-88 ger:



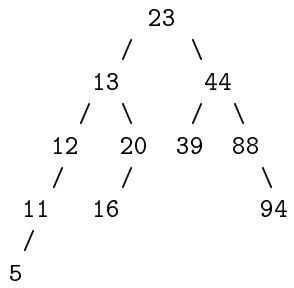
Efter insättning av 11, 39, 20 och 16 får vi:



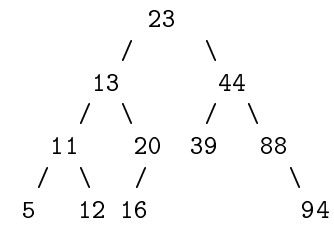
Trenodsomstrukturering av 44-13-23 ger:



Slutligen sätter vi in 5 och får:



Trenodsomstrukturering av 12-11-5 ger:



(d) Vi får följande skip-lista

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

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 `isHeap(t)` är $O(n)$ om n är antalet noder i trädet. Man ser att tiden $f(n)$ för att beräkna `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 vär 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

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

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 partiotioneringarna 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 $O(n)$.