Lösningsförslag och kommentarer till tentamen i Algortitmer och datastrukturer DV1 2000-03-17

1. Ordo, Omega och Theta är matematiska begrepp som uttrycker (matematiska) funktioners asymptotiska beteenden dvs deras väsentliga uppförande för stora värden på argumentet. Begreppen har således i sig ingenting med algoritmer eller tidsåtgång att göra. Däremot använder man dem i algoritmanalys för att beskriva de funktioner som anger hur tiden eller utrymmet beror av indata.

   Definitioner:

   En funktion $f(n)$ sägs vara $O(g(n))$ om det existerar två konstanter n_0 och c som är sådana att $f(n) \leq c \cdot g(n)$ för alla $n \geq n_0$.

   En funktion $f(n)$ sägs vara $\Omega(g(n))$ om det existerar två konstanter n_0 och $c > 0$ som är sådana att $f(n) \geq c \cdot g(n)$ för alla $n \geq n_0$.

   En funktion $f(n)$ sägs vara $\Theta(g(n))$ om det existerar tre konstanter n_0, c_1 och $c_2 > 0$ som är sådana att $c_1 \cdot g(n)$ $gef(n) \geq c_2 \cdot g(n)$ för alla $n \geq n_0$.

   Observera att det är *existensen* av konstanterna som är viktig!

   Lösligt uttryckt kan man säga att Ordo är en övre och Omega en undre gräns medan Theta är likhet. Observera också att t.ex. Ordo i sig inte har något med "värsta" fall att göra! Man kan mycket väl ge en Ordo-uppskattning för tidsåtgången för bästa eller genomsnittsfallet!

2. Tiden för trädsortering kan, för slumptal, förväntas växa som $T(n) = cn \log n$. Eftersom $T(1000) = 1s$ så kan konstanten beräknas som $c = 1/(1000 \cdot \log 1000) = 1/3000$) (Använd 10-logaritmen!) $T(10^6) = 6 \cdot 10^6/3000 = 2000s$)

3.
```
struct node {
  int data;
  struct *left, *right;
}

int max(int x, int y) {
  return x>y?x:y;
}

int height(link r) {
  if ( r==NULL )
    return 0;
  else
    return 1 + max( height(r->left),
                    height(r->right) );
}
```

   Körtiden kan enklast uppskattas genom att konstatera att varje gång vi besöker en viss nivå måste vi passera alla noder i nivåerna ovanför. I ett komplett träd innehåller nivå k exakt 2^k noder (roten definieras som nivå 0) och antalet noder i nivåerna $0 \ldots k-1$ är $2^k - 1$ dvs vi kommer ungefär dubbelt så många noder som behövs vilket innebär att tiden växer linjärt med antalet noder.

4. Den interna väglängden definieras som summan av varje nods väglängd där rotens väglängd definieras som 1, rotens barns som 2 osv. Maximal väglängd:

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

Den minimala väglängden:

Att det räcker att beakta $n = 2^i - 1$ innebär att det räcker att analysera ett komplett träd. Enklast är att gå via den externa väglängden för ett sådant träd. Ett komplett träd med h nivåer har 2^h externa noder på nivå $h + 1$ dvs

$$E_{min} = (h+1)2^h$$

Sambandet $n = 2^h - 1$ ger att $2^h = n + 1$ och $h = \log(n+1)$. Om vi använder detta och sambandet mellan inter och extern väglängd erhålles

$$\begin{aligned} I(n) &= E(n) - 2n - 1 = (\log(n+1) + 1)(n+1) - 2n - 1 \\ &= n\log(n+1) - n + \log(n+1) \end{aligned}$$

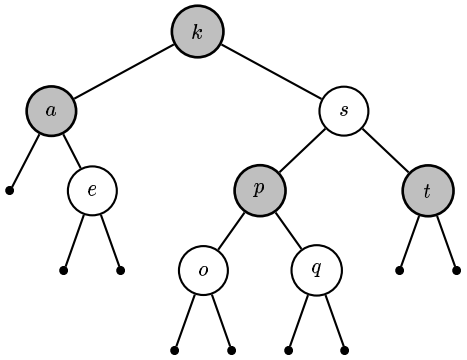
(Det kan vara klokt att kontrollera att detta stämmer för t ex $n = 1$ och $n = 3$)

5. (a) I ett rödsvart träd skall varje nod vara antingen röd eller svart. Alla externa noder (de tomma noderna som varje misslyckad sökning slutar i) skall vara (betraktas som) *svarta*. Vidare skall

   - ingen röd nod ha en röd förälder och
   - alla vägar från roten till en extern (tom) nod innehålla lika många svarta noder.

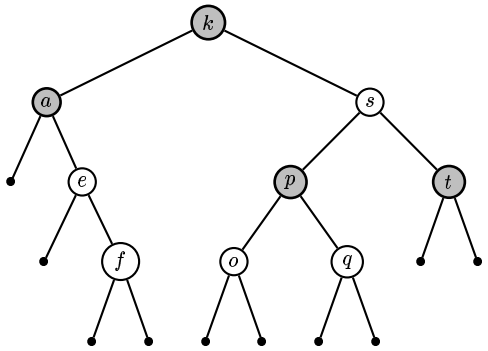
   Att visa att ett träd är ett rödsvart träd görs genom att visa att det går att måla noderna så att ovanstående villkor är uppfyllda.

   Vi kompletterar trädet med de externa noderna (svarta) och visar, med skuggning, vilka övriga noder som skall vara svarta. Trädets svarthöjd är 3.



   När en ny nod läggs in målas den alltid röd för att bevara svartvillkoret. Därefter undersöks rödvillkoret. Inlägg av f i trädet

(b)



(c) Övning :-)

(d) Om noden märkt med e tas bort är trädet fortfarande ett rödsvart träd men ej ett AVL-träd eftersom vänstersubträdet till k har höjd två medan högersubträdet har höjd fyra.

6. (a) Vi använder hashfunktionen $h(k) = k \bmod 7$.

Nycklar k	11	21	3	42	17	2	19	12	26	15	1
$h(k)$	4	0	3	0	3	2	5	5	5	1	1

Om dessa läggs in i tabellen med dubbelhashning erhålles följande tabell (listorna går uppifrån och nedåt):

Index	0	1	2	3	4	5	6
Innehåll i pos 1	21	15	2	3	11	19	
Innehåll i pos 2	42	1		17		12	
Innehåll i pos 3						26	

(b) Vi använder hashfunktionen $h(k) = k \bmod 13$ (tabellen innehåller också värden för stegfunktionen som används i nästa deluppgift):

Nycklar k	11	21	3	42	17	2	19	12	26	15	1
$h(k)$	11	8	3	3	4	2	6	12	0	2	1
$g(k)$	1	11	4	10	7	3	9	2	5	5	2

Linjär kollisionshantering innebär att man, vid kollision, provar nästa plats och, om man kommer till slutet av tabellen, börjar om på plats 0. Tabellen blir då:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
Innehåll	26	1	2	3	42	17	19	15	21			11	12
Antal försök	1	2	1	1	2	2	1	6	1			1	1

För att räkna ut hur många försök i genomsnitt en misslyckad sökning i just denna tabell antar man att varje position är lika sannolik. Om vi kommer till position 0 krävs 10 försök (vi provar 0, 1, 2, ..., 9), om vi kommer till position 1 krävs 8 försök osv. Position 9 och kräver vardera 1 försök medan position 11 och 12 kräver 12 respektive 11 försök. Sammanlagt: $12 + 11 + 10 + \cdots + 2 + 1 + 1 = (12 + 1) \cdot 12/2 + 1 = 79$ dvs genomsnittet blir $79/13 = 6.08$

(c) Vid dubbel hashning använder man ett individuellt steg för varje nyckel som beräknas med funktionen $g(k) = (k \bmod 11) + 1$. Hashtabellen blir då:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
Innehåll	42	1	2	3	17	26	19	15	21			11	12
Antal försök	2	1	1	1	1	2	1	2	1			1	1

Lyckad sökning i denna tabella kräver $(2 + 1 + 1 + 1 + 1 + 2 + 1 + 2 + 1 + 1 + 1)/11 = 14/11 \approx 1.3$

(d) Vid *ordnad hashning* låter man en större nyckel ha prioritet framför en mindre till platserna. Om man vid inläggning av en nyckel k påträffar en nycklel l som är mindre en k lagrar man k på denna plats och fortsätter inlägget med l, naturligtvis med använde av $g(l)$ som steg.

Fördelen med *ordnad hashning* är att sökning kan avbrytas när man kommit till en plats med en mindre nyckel än den man söker vilket medför att det går fortare att upptäcka att en given nyckel inte finns med i tabellen.

(e) Se utdelat papper om hashning.

7. (a) En *heap* kan definieras som en talföljd: $h_1, h_2, h_3, \ldots, h_n$ som uppfyller relationerna $h_i \leq h_{2i}$ och $h_i \leq h_{2i+1}$. Vanligen placeras talföljden i en array med början i position 1. (Alternativt kan man säga att en heap är ett vänsterkomplett binärt träd där varje nod är mindre än sina barn)

   (b) Det finns ingen entydig ordning — det som krävs är att heap-villkoren är uppfyllda. Naturligtvis uppfyller en sorterad array dessa villkor men även t ex nedanstående:

Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Innehåll	1	2	12	11	3	42	26	17	15	21	19		

   Inlägg tillgår så att det nya talet placeras på den första lediga platsen vilket i detta fall är position 12. Därefter kontrolleras relationen med position $12/2 = 6$. Om fel ordning så byt och kontrollera vidare med position $6/2 = 3$. Inlägg av 7 resulterar i detta fall i följande heap:

Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Innehåll	1	2	7	11	3	12	26	17	15	21	19	42	

   Vid urtag så hämtas det minsta elementet från position 1 varefter det sista flyttas till position 1 och jämförs med barnen (position 2 och 3). Om minsta barnet mindre görs ett byte och kontroll görs etc. Ett urtag i denna heap ger:

Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Innehåll	1	2	7	11	3	12	26	17	15	21	19	42	
Nytt innehåll	2	3			21					42		nil	