Lösningsförslag till tentamen i Algortitmer och datastrukturer DV1 1999-03-19

1.
```
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) );
}

void insert(int key, link *r) {
  assert(r);
  if ( *r == 0 )
    *r = cons( key, NULL, NULL );
  else if ( key < (*r)->item )
    insert( key, &((*r)->left) );
  else if ( key > (*r)->item )
    insert( key, &((*r)->right) );
  else
    /* Redan inlagt. Gör ingenting. */;
}
```

2. Vi använder hashfunktionen $h(k) = k \bmod 13$ samt stegfunktionen $g(k) = (k \bmod 11) + 1$

Nycklar k	8	13	28	2	22	5	19	21	15	4
$h(k)$	8	0	2	2	9	5	6	8	2	4
$g(k)$	9	3	7	3	1	6	9	11	5	5

Om dessa läggs in i tabellen med dubbelhashning erhålles följande tabell

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
Innehåll	13	4	28		21	2	19		8	22	15	5	
Antal försök	1	3	1		3	2	1		1	1	2	2	

Totala antalet försök vid uppbyggnaden är 17 dvs det krävs i genomsnitt $17/10 = 1.7$ försök för att lokalisera en existerande nyckel.

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.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
Innehåll	13	4	28		8	~~2~~ 5	19	15	~~8~~ 21	22		2	
Antal försök	1	3	1		2	1	1	2	1	1		4	

Även i denna tabell kräver lyckad sökning $17/10 = 1.7$ försök i genomsnitt. 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.

3. (a) O(1) betyder *konstant* tid oberoende av n. Det skall således ta 1 sekund.

(b) $\Theta(\log n)$ betyder att tiden växer som $T(n) = c\log(n)$. Om 10-logaritmen används så

$T(1000) = c\log(1000) = 1$ ger $c = 1/3$

$T(10^6) = c\log(10^6) = 1/3 \cdot 6 = 2$ sekunder

(c) $\Theta(n)$ står för linjär tillväxt dvs $T(n) = cn$. Om n ökar med en faktor 1000 så kommer tiden också att öka med en faktor 1000. Svaret är således 1000 sekunder.

(d) $\Theta(n\log n)$ innebär att tiden växer som $T(n) = cn\log(n)$

$T(1000) = c \cdot 1000 \cdot \log(1000) = 1$ ger $c = 1/3000$

$T(10^6) = c \cdot 10^6 \cdot \log(10^6) = \frac{6 \cdot 10^6}{3000} = 2000$ sekunder

(e) $\Theta(n^2)$ innebär att tiden växer som $T(n) = cn^2$

$T(1000) = c \cdot 1000^2 = 1$ ger $c = 10^{-6}$

$T(10^6) = c \cdot (10^6)^2 = 10^{-6} \cdot 10^{12} = 10^6$ sekunder dvs drygt 11 dygn.

4. (a) Den *interna väglängden* är summan av alla varje nods väglängd där roten har väglängd 1, rotens barn väglängd 2, rotens barbarn väglängd 3 osv.

Den *externa väglängden* är väglängden till alla externa noder ("tomma" träd).

(b) Vi använder ett induktionsbevis. Påståendet $E = I + 2n + 1$ gäller för det tomma trädet eftersom $E = 1$ och $I + 2n + 1 = 0 + 2 \cdot 0 + 1 = 1$

Påståendet är ekvivalent med $E - I = 2n + 1$. Antag att påståendet gäller för ett träd med n (interna) noder. Om vi byter ut en av de externa noderna på nivå k mot en intern nod följt av två externa noder. Den interna väglängden ökar då med k medan den externa väglängden minskar med k och ökar med $2 \cdot (k + 1)$ Om vi betecknar de nya väglängderna med E' och I'

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

Eftersom det nya trädet har $n+1$ noder så gäller således sambandet även för detta.

(c) Den största väglängden erhålles med en nod på varje nivå:

$$I = 1 + 2 + 3 + \ldots + n = \frac{n(n + 1)}{2} \tag{1}$$

(d) Den minsta möjliga väglängden i ett komplett träd med h nivåer:

$$I = 1 + 2 + 2 + 3 + 3 + 3 + 3 + \ldots + 2^{h-1} \cdot h$$

Denna summa där varje term är en produkt av en termerna från en

aritmetisk och en geometrisk summa kan beräknas på följande sätt:

$$
\begin{array}{rrrrrrrl}
I = 1 & +2 \cdot 2^1 + & 3 \cdot 2^2 + & 4 \cdot 2^3 + & \ldots & +h \cdot 2^{h-1} & & \\
= 1 & +2^1 + & 2^2 + & 2^3 + & \ldots & +2^{h-1} & + & \\
& +2^1 + & 2^2 + & 2^3 + & \ldots & +2^{h-1} & + & \\
& + & 2^2 + & 2^3 + & \ldots & +2^{h-1} & + & \\
& & + & 2^3 + & \ldots & +2^{h-1} & + & \\
& & & & \ldots & & & \\
& & & & & 2^{h-1} & &
\end{array}
$$

Varje rad i ovanstående utgör en summa av en geometrisk serie med två som kvot. Dessa kan summeras var för sig.

$$
\begin{aligned}
I &= \sum_{i=0}^{h-1} 2^i + \sum_{i=1}^{h-1} 2^i + \sum_{i=2}^{h-1} 2^i + \ldots + \sum_{i=h-1}^{h-1} 2^i \\
&= 2^0 \sum_{i=0}^{h-1} 2^i + 2^1 \sum_{i=0}^{h-2} 2^i + 2^2 \sum_{i=0}^{h-3} 2^i + \ldots + 2^{h-1} \sum_{i=0}^{0} 2^i \\
&= 2^0(2^h - 1) + 2^1(2^{h-1} - 1) + 2^2(2^{h-2} - 1) + \ldots + 2^{h-1}(2^1 - 1) \\
&= h2^h - \sum_{i=0}^{h-1} 2^i = h2^h - (2^h - 1)
\end{aligned}
$$

Antal noder n i ett träd med h nivåer är

$$n = 1 + 2 + 4 + \ldots + 2^{h-1} = 2^h - 1$$

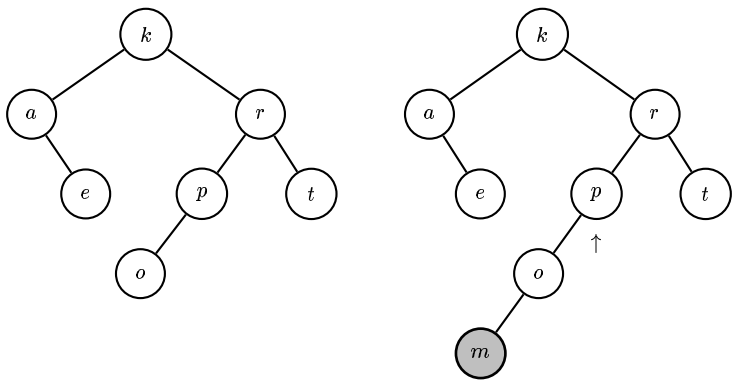
vilket innebär att

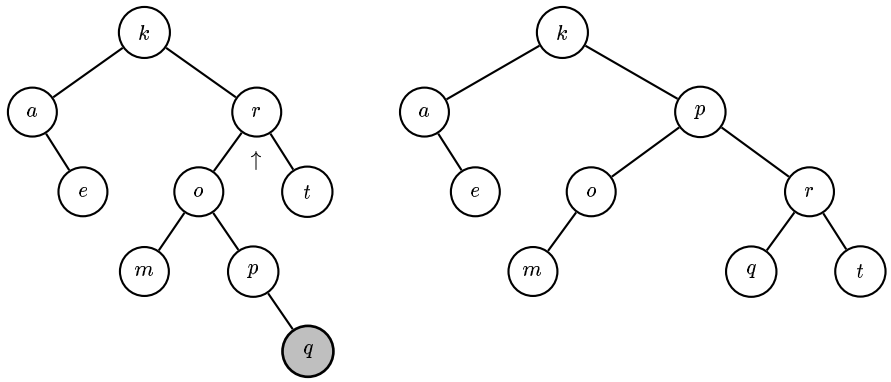
$$
\begin{aligned}
2^h - 1 &= n \\
2^h &= n + 1 \\
h &= \log_2(n + 1)
\end{aligned}
$$

Om detta användes i uttrycket för I ovan så erhålles slutligen

$$I = (n + 1)\log_2(n + 1) - n$$

5. Inläggning av m förorsakar en enkel högerrotation med p som pivotnode och inläggning av q medför en dubbel högerrotation kring r:





6. Ett sätt att lösa problemet att bestämma medianen är att lagra talen i en array (med n element), sortera denna och se vilket element som kommer på plats $n/2$. Med de vanliga, snabba sorteringsmetoderna som quicksort eller mergesort tar detta dock $\Theta(n \log n)$ tid vilket alltså inte uppfyller kravet i uppgiften. Ett vanligt algoritm för att beräkna medianen är att modifiera quicksort så att den i rekursionen bara går vidare men den del som medianen kommer att finnas i. Denna metod är $O(n)$ i *genomsnitt* vilket inte heller uppfyller kravet i uppgiften (som skulle garantera högst linjärt tidsberoende). (Det är dessutom inte helt enkelt att visa att algoritmen är $O(n)$.)

   För att lösa problemet skall vi i stället utnyttja den kunskap om nycklarna som omnämns i uppgiften: heltal i intervallet [1,1000]. Det betyder att det finns högst 1000 olika nycklar. Vi använder en array med 1000 platser (indicerad med talen i intervallet [1,1000]) där vi på plats i lagrar hur många nycklar med värdet i som finns. Detta är i själva verket en sorteringsmetod som endast kräver $O(n)$ operationer. Slutligen måste vi räkna oss fram till var den mittersta nyckelm hamnar genom att succesivt summera antalet antalet nycklar i position 1, 2, ... tills halva antalet nycklar uppnåtts.

   Antag att nycklarna är lagrade (osorterat) i en array `keys[n]`

   Algoritm:

```
for i = 1 to 1000 do
    freq[i] = 0
for i = 1 to n do
    freq[keys[i]] = freq[keys[i]] + 1
sum = 0
i   = 1
while ( sum < n/2 )
   sum = sum + freq[i]
   i   = i + 1
return i  /* Medianen */
```

   Den första for-iteration kräver ett konstant antal operationer (oberoende av n). Den andra utförs exakt n gånger och kräver således $\Theta(n)$ operationer. Den sista kan högst utföras 1000 gånger och är således oberoende av n. Totalt krävs således $\Theta(n)$ operationer.