Tentamen i Algortitmer och datastrukturer DV1 2001-03-16

Skrivtid: 0800 – 1300
Hjälpmedel: Räknedosa, *Några användbara matematiska samband*
Varje delfråga kan, om inget annat anges, ge högst 1 poäng.
Svar skall motiveras om inte annat anges.

Lycka till!

Uppgifter

1. Nedan följer ett antal påståenden. Ange, utan motivering, för vart och ett av dessa huruvida det *alltid* är korrekt eller ej. Rätt svar ger 0.5 poäng, fel svar -0.5 poäng och uteblivet svar 0 poäng. Totalpoängen på uppgiften kan dock aldrig bli negativ.

   (a) Tiden för heapsort är $\Theta(n \log n)$

   (b) Att söka en nyckel med binär sökning i en sorterad array kräver $O(\log n)$ operationer.

   (c) Den största möjliga interna väglängden i ett binärt träd med 1000 noder är ungefär 5 000 000.

   (d) Den minsta möjliga interna väglängden i ett binärt träd med 1000 noder är ungefär 12 000.

2. Följande deklarationer är givna och skall användas för att representera listor sorterade på `key`-komponenten:

```
typedef
struct listElem {
  int key;
  int count;
  struct listElem *next;
} listElem, *link;

link cons(int key, link next) {
  link l = (link) malloc(sizeof(listElem));
  l->key   = key;  // Nyckel
  l->count = 1;    // Frekvensräknare
  l->next  = next; // Pekare till nästa
  return l;
}
```

   (a) Skriv en funktion `int sum(link first)` som summerar frekvensräknarna (`count`-komponenterna) i den lista vars första element pekas ut av `first`. Summan skall returneras som funktionsvärde.

   (b) Skriv en funktion `void insert(int key, link *fp)` för inläggning av en ny nyckel `key`. Om denna nyckel redan finns i listan så skall dess

frekvensräknare ökas med 1 i annat fall skall ett nytt element skapas och läggas in så att sorteringen bevaras. Parametern `*fp` pekar till första elementet i listan. (2p)

(c) När nedanstående program testkördes tog det ca 1 sekund.

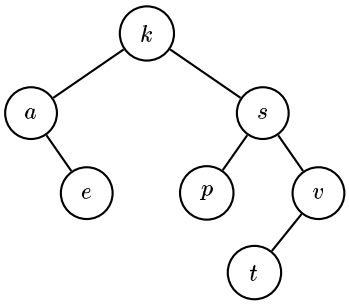
```
main() {
  link l=0;
  int k, i, n=1000;
  for ( i=1; i<=n; i++ ) {
    k = lrand48();
    insert(k,&l);
  }
}
```

Funktionen `lrand48()` returnerar ett slumptal (heltal) i intervallet $[0, 2^{31}]$. Uppskatta hur lång tid programmet skulle ta om $n = 10^6$ dvs hur lång tid det skulle ta att sortera 10^6 tal med denna metod.

(d) Antag att vi i ovanstående program endast lagrar tresiffriga nycklar t ex genom att sätta `k = lrand48() % 1000` och att det fortfarande tar 1 sekund för $n = 1000$. Uppskatta då hur långt tid det kommer att ta med $n = 10^6$.

3. Vid testkörning av ett program som implementerar treesort på slumptal uppmättes tiden 10 sekund för att sortera 10^6 tal. Uppskatta hur lång tid som krävs för att sortera 10^9 tal.

4. Givet följande binära sökträd med bokstäver som nycklar:



(a) Hur många försök krävs i genomsnitt för en misslyckad sökning i detta träd?

(b) Vad menas med ett *rödsvart* träd? Visa att ovanstående träd är ett sådant.

(c) Lägg in nycklarna u och x så att trädet förblir ett rödsvart träd. Om du använder annat än standardalgoriterna måste du redovisa dessa. (Alternativt kan du behandla detta som ett AVL-träd och göra AVL-trädsinlägg.)

(d) Bevisa att höjden i ett rödsvart träd är $O(\log n)$ där n är antalet noder. (Alternativt samma sak för ett AVL-träd.) (2p)

5. Använd nycklarna 23, 12, 7, 13, 5, 2, 19, 26, 15, 28, 4 (i denna ordning) för att bygga upp hashtabeller med m platser enligt nedan. Som hashfunktion skall

$$h(k) = k \bmod m$$

och, i förekommande fall, stegfunktionen

$$g(k) = k \bmod (m-2) + 1$$

användas.

   (a) En *länkad* hashtabell. Tabellens storlek m skall vara 7.

   (b) En öppen hashtabell (alla nycklar i tabellen) med *linjär* kollisionshantering och med tabellstorleken $m = 13$. Hur många försök kräver en *lyckad* respektive en *misslyckad* sökning i genomsnitt i just denna tabell?

   (c) En öppen hashtabell med *ordnad dubbel hashning* med tabellstorleken $m = 13$. Det räcker att redovisa tabellens utseende — inte hur du kommit fram till den. Hur många försök kräver en *lyckad* sökning i genomsnitt i just denna tabell?

   (d) Antag att vi lagrar n element i en hashtabell med m platser och använder *sorterade* listor i kollisionshanteringen. Härled formler för hur det genomsnittliga antalet försök för lyckad respektive misslyckad sökning i detta fall beror av n och m. (2p)

6. (a) Definiera begreppet *heap* (i datastruktur- och algoritmsammanhang).

   (b) Skriv en funktion `void heapInsert(int h[], int *n, int k)`. Parametern `h` än en array som innehåller en *heap* med `*n` element (men det finns plats för fler). Funktionen skall lägga in `k` så att heapegenskaperna bevaras. Funktionen skall också uppdatera antalet lagrade element `*n`.