Tentamen
Datastrukturer (DAT036)

- Datum och tid för tentamen: 2012-04-13, 8:30–12:30.
- Ansvarig: Nils Anders Danielsson. Nås på 0700 620 602 eller anknytning 1680. Besöker tentamenssalarna ca 9:30 och ca 11:30.
- Godkända hjälpmedel: Kursboken (Data Structures and Algorithm Analysis in Java, Weiss, valfri upplaga), handskrivna anteckningar.
- För att få betyget n (3, 4 eller 5) måste du lösa minst n uppgifter. Om en viss uppgift har deluppgifter med olika gradering så räknas uppgiften som löst om du har löst *alla* deluppgifter med gradering n eller mindre.
- För att en uppgift ska räknas som "löst" så måste en i princip helt korrekt lösning lämnas in. Enstaka mindre allvarliga misstag kan *eventuellt* godkännas (kanske efter en helhetsbedömning av tentan; för högre betyg kan kraven vara hårdare).
- Lämna inte in lösningar för flera uppgifter på samma blad.
- Skriv din tentakod på varje blad.
- Lösningar kan underkännas om de är svårlästa, ostrukturerade eller dåligt motiverade. Pseudokod får gärna användas, men den får inte utelämna för många detaljer.
- Om inget annat anges så kan du använda kursens uniforma kostnadsmodell när du analyserar tidskomplexitet (så länge resultaten inte blir uppenbart orimliga).
- Om inget annat anges behöver du inte förklara standarddatastrukturer och -algoritmer från kursen, men däremot motivera deras användning.

1. Analysera nedanstående algoritms värstafallstidskomplexitet, givet att `X` och `Y` är länkade listor av längd |`X`| och |`Y`|, och att det tar konstant tid att jämföra två element från de här listorna.

```
disjoint(X,Y):
  S = new empty AVL tree
  for every element x in X
    S.insert(x)
  for every element y in Y
    if S.member(y) then
      return false
  return true
```

2. *För trea:* Uppgiften är att konstruera en datastruktur som implementerar en variant av avbildnings-ADTn ("map-ADTn").

   Anta att nycklar är totalt ordnade och att det finns en komparator som på konstant tid avgör hur två nycklar k_1 och k_2 är relaterade ($k_1 < k_2$, $k_1 = k_2$ eller $k_1 > k_2$). ADTn ska ha följande operationer:

   `multi-map`: Skapar en tom avbildning.

   `insert(`k`,`v`)`: Lägger till en bindning $k \mapsto v$ till avbildningen. Tidigare bindningar tas *inte* bort.

   `delete(`k`)`: Den här operationen får endast anropas om det finns minst en bindning för nyckeln k. En sådan bindning $k \mapsto v$ tas bort, och motsvarande värde v lämnas som svar.

   Exempel (om vi antar att `multi-map` är en konstruktor): Resultatet av

```
example1():
  m = new multi-map
  m.insert(1,'a')
  m.insert(1,'b')
  return m.delete(1)
```

   ska vara `'a'` eller `'b'`. Resultatet av

```
example2():
  m = new multi-map
  m.insert(1,'a')
  m.insert(1,'b')
  m.insert(2,'c')
  m.delete(1)
  return m.delete(1)
```

   ska också vara `'a'` eller `'b'`.

   För fyra: Som ovan, men tidskomplexiteterna för `insert` och `delete` måste vara $O(\log n)$, där n är antalet *nycklar* (inte bindningar) i avbildningen.

3. Implementera en operation som lägger till en lista i slutet av en annan lista. I värsta fallet ska operationen ta konstant tid. Beskriv noggrant hur listor representeras (gärna med figurer). Exempel: Om man lägger till $[3,4,5]$ i slutet av $[1,2]$ ska resultatet bli $[1,2,3,4,5]$.

4. *För trea:* Beskriv en effektiv algoritm som, givet en array med n distinkta heltal och ett naturligt tal $k \leq n$, beräknar produkten av de k minsta talen i arrayen. Exempelvis ska svaret bli 12 för arrayen $\{3,7,5,4\}$ då $k = 2$. Analysera algoritmens tidskomplexitet, uttryckt i n och k.

   För fyra: Som för trea, men tidskomplexiteten måste vara "bättre" än $\theta(n \log n)$.

5. Låt oss representera naturliga tal n (0, 1, 2, ...) som listor av bitar $b_0 b_1 ... b_k$, där $k \geq 0$,

$$n = \sum_{i=0}^{k} b_i 2^{k-i},$$

   och $b_0 = 0$ om och endast om $n = 0$. Exempel:

Tal (n)	Representation
0	0
3	11
8	1000

   Följande ekvationer definierar en funktion *inc* som, givet representationen av talet n, beräknar representationen av talet $n+1$:

$$\begin{aligned} inc(b_0 b_1 ... b_{k-1} 0) &= b_0 b_1 ... b_{k-1} 1, \quad k \geq 0 \\ inc(b_0 b_1 ... b_{k-1} 1) &= \begin{cases} inc(b_0 b_1 ... b_{k-1})0, & k \geq 1 \\ 10, & k = 0 \end{cases} \end{aligned}$$

   Exempel: $inc(0) = 1$, $inc(1) = 10$, $inc(111) = 1000$.

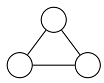
   Visa att det går att implementera *inc* på ett sådant sätt att tidskomplexiteten för att beräkna

$$\overbrace{inc(inc(...(inc}^{n}(0))...)),$$

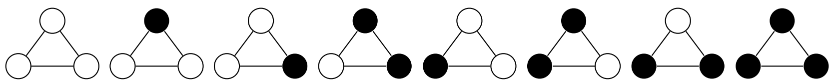
   med n förekomster av *inc*, är $O(n)$. (Tips: Med en lämplig representation av listor blir ekvationerna ovan detaljerad pseudokod.)

6. Beskriv en effektiv algoritm som avgör om noderna i en oriktad, sammanhängande graf kan färgas svarta och vita på ett sådant sätt att angränsande noder har olika färger, och visa att algoritmen verkligen är effektiv.

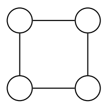
   Exempel: Om algoritmen appliceras på grafen

   så ska svaret vara negativt, eftersom det inte går att färga grafen på det önskade sättet:

   För grafen

   så ska svaret emellertid vara positivt, eftersom grafen kan färgas på följande sätt: