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

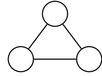
   Visa att det går att implementera *inc* på ett sådant sätt att tidskomplexiteten för att beräkna

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

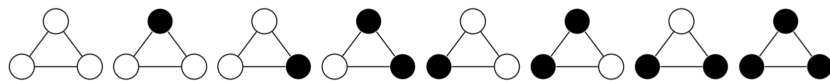
   med n förekomster av *inc*, är $O(n)$. (Tips: Med en lämplig representation av listor blir ekvationerna ovan detaljerad pseudokod.)

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.

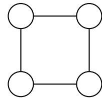
   Exempel: Om algoritmen appliceras på grafen

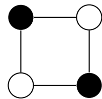
   så ska svaret vara negativt, eftersom det inte går att färga grafen på det önskade sättet:

   För grafen

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

Kortfattade lösningsförslag för tentamen i Datastrukturer (DAT036) från 2012-04-13

Nils Anders Danielsson

1. Första loopen: $O(|\texttt{X}| \log |\texttt{X}|)$.[1] Andra loopen: $O(|\texttt{Y}| \log |\texttt{X}|)$. Övrigt: $O(1)$. Totalt: $O((|\texttt{X}| + |\texttt{Y}|) \log |\texttt{X}|)$.

2. Representera avbildningarna som AVL-träd som mappar nycklar till (enkellänkade) *listor* av värden. I ett språk som liknar Java:

```
class multi-map<K,V> {
  Map<K,List<V>> map;

  multi-map() {
    map = new AVLTree<K,List<V>>();
  }

  void insert(K k, V v) {
    List<V> vs = map.lookup(k);
    if (vs == null) {
      vs = new LinkedList<V>();
    }
    map.insert(k, vs.insert(v));
  }

  V delete(K k) {
    List<V> vs = map.lookup(k);
    // Notera att vi kan anta att vs är icketom.
    V v = vs.head();
    vs  = vs.tail();
    if (vs.empty()) {
      map.delete(k);
    } else {
      map.insert(k,vs);  // Tar bort den gamla bindningen.
    }
    return v;
  }
}
```

[1] Notera att $\sum_{i=1}^{|\texttt{X}|} \log i = \log\left(\prod_{i=1}^{|\texttt{X}|} i\right) = \log(|\texttt{X}|!\,) = \theta(|\texttt{X}| \log |\texttt{X}|)$.

Notera att om det finns en bindning i trädet för en nyckel k så är motsvarande lista vs icketom. Alltså är trädoperationerna som värst logaritmiska i antalet nycklar (det skulle de inte vara om raden `map.delete(k);` togs bort). Övriga operationer tar konstant tid.

3. Listrepresentation: Dubbellänkade listor med ”vaktposter” i början och slutet samt pekare till vaktposterna. I ett språk som liknar Java:

```
class List<A> {
  class ListNode<A> {
    A            contents;
    ListNode<A> prev, next;

    ListNode(A contents, ListNode<A> prev, ListNode<A> next) {
        this.contents = contents;
        this.prev     = prev;
        this.next     = next;
    }
  }

  ListNode<A> head;  // Pekar på första vaktposten.
  ListNode<A> tail;  // Pekar på sista vaktposten.

  // Skapar en tom lista.
  List() {
    head      = new ListNode<A>(null, null, null);
    tail      = new ListNode<A>(null, head, null);
    head.next = tail;
  }

  // Lägger till elementen från ys sist i listan, gör om ys till
  // en tom lista.
  public void append(List<A> ys) {
    tail.prev.next    = ys.head.next;
    ys.head.next.prev = tail.prev;
    tail              = ys.tail;

    ys.tail           = new ListNode<A>(null, ys.head, null);
    ys.head.next      = ys.tail;
  }
}
```

   Notera att `append` tar konstant tid.

4. Enkel lösning: Sortera arrayen, multiplicera de k första elementen. Värstafallstidskomplexitet med t ex merge sort eller heap sort: $O(n \log n) + O(k) = O(n \log n)$.

   En annan lösning: Om $k = 0$, ge 0 som svar. Annars, använd quickselect för att hitta det k-te minsta talet i, multiplicera sedan alla tal $\leq i$ (det finns

exakt k stycken eftersom talen är distinkta). Medelfallstidskomplexitet (med lämplig partitioneringsstrategi): $O(n) + O(n) = O(n)$.

5. Låt oss representera bitlistor som enkellänkade listor av bitar, där den *minst signifikanta* biten är *först* i listan. Då är det lätt att implementera *inc* m h a de rekursiva ekvationerna i uppgiftsformuleringen.

   Låt oss sedan använda potentialmetoden för att visa att operationen *inc* har den amorterade tidskomplexiteten $O(1)$ när den används som i uppgiften.

   Vi kan låta potentialen av en bitlista bs vara $\Psi(bs) = k|bs|$, där $|bs|$ står för antalet ettor i bs, och k är en positiv konstant. Notera att potentialfunktionen är OK: beräkningen börjar med bitlistan 0, och $\Psi(bs) \geq 0 = \Psi(0)$ för alla bitlistor bs.

   Låt oss nu bevisa att en beräkning $inc(b_0 b_1 ... b_{k-1} b_k)$ som kräver i anrop till *inc* ökar potentialen med som mest $(2 - i)k$. Eftersom konstant arbete utförs för varje anrop så betyder det att den amorterade tidskomplexiteten är $\leq O(i) + (2 - i)k$, vilket är $O(1)$ om k väljs tillräckligt stor.

   Vi kan bevisa det här påståendet med induktion över bitlistans längd. Det finns tre fall att analysera:

   - $bs = b_0 b_1 ... b_{k-1} 0$: Här har vi $i = 1$, och kan därmed dra slutsatsen att

     $$\Psi(b_0 b_1 ... b_{k-1} 1) - \Psi(b_0 b_1 ... b_{k-1} 0) = k = (2 - 1)k = (2 - i)k.$$

   - $bs = 1$: Här har vi också $i = 1$, och får

     $$\Psi(10) - \Psi(1) = 0 \leq k = (2 - 1)k = (2 - i)k.$$

   - $bs = b_0 b_1 ... b_{k-1} 1$ och $k \geq 1$: Här har vi

     $$\begin{aligned} &\Psi(inc(b_0 b_1 ... b_{k-1})0) - \Psi(b_0 b_1 ... b_{k-1} 1) = \\ &\Psi(inc(b_0 b_1 ... b_{k-1})) - \Psi(b_0 b_1 ... b_{k-1}) - k \leq \\ &(2 - (i - 1))k - k = \\ &(2 - i)k. \end{aligned}$$

     I det näst sista steget användes induktionshypotesen.

6. Låt oss använda termen ”korrekt” för en färgning med den önskade egenskapen.

   Antag att det finns en korrekt färgning. I så fall är också den ”motsatta” färgningen korrekt. Vidare, om en viss nod har en viss färg så måste alla dess grannar ha motsatt färg.

   De här observationerna leder till följande algoritm, som färgar en nod i taget på ett sådant sätt att om det finns en korrekt färgning så är den partiella färgningen alltid en ”delfärgning” av någon korrekt färgning:

- Om grafen är tom, dra slutsatsen att den kan färgas korrekt.
- Annars, välj en godtycklig nod och ge den en godtycklig färg, t ex svart.
- Gör sedan en bredden först-sökning (eller djupet först-sökning) med start i den noden, och gör följande när en nod besöks första gången:
  - Noden har då redan färgats. Benämn färgen f.
  - Om någon av nodens grannar också har färgats med färgen f, dra slutsatsen att det inte finns någon korrekt färgning.
  - Annars, färga alla grannarna med motsatt färg (inte f).

Om alla noder kan färgas så har vi ett konstruktivt bevis för att det finns en korrekt färgning.

Algoritmen är effektiv: Om vi representerar grafen med grannlistor så utförs i värsta fallet konstant arbete för varje nod och konstant arbete för varje kant, så algoritmen är linjär i grafens storlek.