Tentamen
Datastrukturer (DAT036)

- Datum och tid för tentamen: 2013-04-05, 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: Ett A4-blad med 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 kods tidskomplexitet, uttryckt i `n`:

   ```
   for (int i = 0; i < n; i++) {
       for (int j = 0; j < i; j++) {
           t.insert(n);
       }
   }
   ```

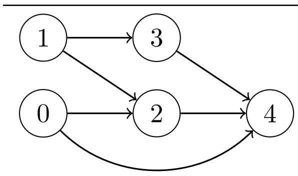
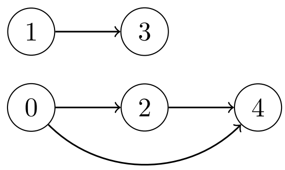
   Använd kursens uniforma kostnadsmodell, och gör följande antaganden:

   - Att `n` är ett ickenegativt heltal, och att typen `int` kan representera alla heltal.
   - Att `t` är ett AVL-träd som till att börja med är tomt.
   - Att den vanliga ordningen för heltal ($\ldots < -1 < 0 < 1 < 2 < \ldots$) används vid insättning i trädet.
   - Att om samma element sätts in två gånger i trädet så skrivs den tidigare förekomsten över.

   Onödigt oprecisa analyser kan underkännas; använd gärna Θ-notation.

2. Man kan sortera riktade acykliska grafer (DAGs) topologiskt genom att göra en djupet först-sökning och pusha noderna på en stack i postordning. Implementera den här algoritmen. Skriv en procedur som tar en graf och ger tillbaka en lista med grafens noder i topologisk ordning.

   Exempel:

Graf	Möjligt resultat
	1, 3, 0, 2, 4
	1, 3, 0, 2, 4

   Beskriv tydligt vilken grafrepresentation som används i lösningen. Djupet först-sökningen måste beskrivas detaljerat; pseudokod får användas, men den får inte utelämna för många detaljer. (Det räcker inte att skriva något i stil med "besök varje nod i djupet först-ordning och …", det ska vara enkelt – vad nu det betyder – att översätta pseudokoden till konkret kod i ett språk som Java.)

   Tips: Testa din kod, så kanske du undviker onödiga fel.

3. *För trea:* Beskriv en effektiv algoritm som, givet en array med n distinkta naturliga tal och ett naturligt tal $k \leq n$, beräknar summan av de k största talen i arrayen. Exempelvis ska svaret bli 12 för arrayen $\{3, 7, 5, 4\}$ då $k = 2$. Du kan anta att inget tal i arrayen är större än N. Analysera algoritmens tidskomplexitet, uttryckt i n, k och N.

   För fyra: Som för trea, men tidskomplexiteten måste vara $O(n \log k)$.

4. Uppgiften är att konstruera en datastruktur för en mängd-ADT med följande operationer:

   `new Set()` Konstruerar en tom mängd.

   `insert(`i`)` Lägger till heltalet i till mängden.

   `delete(`i`)` Tar bort heltalet i från mängden. (Lämnar mängden oförändrad om i inte finns i mängden.)

   `member(`i`)` Avgör om heltalet i finns i mängden.

   `delete-odd()` Tar bort alla *udda* heltal från mängden.

   Exempel: Följande kod, skriven i ett Javaliknande språk, ska ge `true` som svar:

```
Set s = new Set();
s.insert(0);
s.insert(1);
s.insert(2);
boolean b = s.member(0) && s.member(1) && s.member(2);
s.delete(2);
b = b && s.member(0) && s.member(1) && (! s.member(2));
s.delete-odd();
b = b && s.member(0) && (! s.member(1)) && (! s.member(2));
return b;
```

   Operationerna måste ha följande tidskomplexiteter (där n är antalet element i mängden):

   - *För trea:* `new`: $O(1)$, `insert`, `delete`, `member`: $O(\log n)$, `delete-odd`: $O(n \log n)$.
   - *För fyra:* `new`: $O(1)$, `insert`, `delete`, `member`: $O(\log n)$, `delete-odd`: $O(1)$.

   Visa att så är fallet. Implementationen av datastrukturen behöver inte beskrivas med detaljerad kod, men lösningen måste innehålla så mycket detaljer att tidskomplexiteten kan analyseras.

   Tips: Konstruera om möjligt inte datastrukturen från grunden, bygg den hellre m h a standarddatastrukturer. Testa din algoritm, så kanske du undviker onödiga fel.

5. Anta att en dynamisk array har implementerats på följande sätt:
   - Det finns två operationer, insättning på den första lediga positionen (eventuellt efter förstoring av arrayen) och borttagning av det sista elementet (om arrayen inte är tom).
   - Arrayen är till att börja med tom, med längden 1.
   - Arrayens längd fördubblas vid insättning i en full array.
   - Arrayens längd halveras efter borttagning av ett element om arrayen, efter borttagningen, är som mest halvfull, och dess längd är minst 2.

   Visa att värstafallstidskomplexiteten för en sekvens av n operationer är $\Omega(n^2)$.

   Tips: För varje (tillräckligt stort) n, konstruera en sekvens s_n bestående av n operationer. Visa sedan att det finns en positiv konstant c så att, för varje tillräckligt stort n, tidskomplexiteten för s_n är minst cn^2. (Använd gärna Ω-notation i beviset.)

6. Betrakta följande Javaklass, som representerar enkellänkade listor:

```
public class List<A> {
  private class ListNode {
    A        contents;  // Innehåll.
    ListNode next;      // Nästa nod; null om det inte
                        // finns fler noder.

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

  // Pekar på första listnoden; null om listan är tom.
  private ListNode head;

  // Diverse metoder (som ej får användas i lösningen).
  ...
}
```

   Lägg till en metod `public void reverse()` till klassen. Metoden ska reversera listan. *Exempel:* $[\,]$ ska transformeras till $[\,]$, och $[0, 1, 2]$ till $[2, 1, 0]$.

   Endast detaljerad kod (inte nödvändigtvis Java) godkänns. Du får inte anropa några andra metoder/procedurer (förutom `List`- och `ListNode`-konstruerarna), om du inte implementerar dem själv.

   Metoden måste vara linjär i listans längd ($O(n)$, där n är längden). Visa att så är fallet.

   Tips: Testa din kod, så kanske du undviker onödiga fel.