Tentamen
Datastrukturer (DAT036)

Det här är inte originaltesen. Uppgift 6 var felaktigt formulerad, och har rättats till.

- Datum och tid för tentamen: 2011-12-16, 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).

1. *(Lätt.)* Skriv ett effektivt program som tar en lista av strängar, beräknar varje strängs frekvens (hur många gånger den förekommer i listan), och skriver ut strängarna sorterade efter frekvens (i fallande ordning). Om två strängar har samma frekvens spelar ordningen ingen roll. Exempel:

   - Lista: `"elefant"`, `"giraff"`, `"lodjur"`, `"elefant"`, `"elefant"`
   - Utskrift:

     ```
     elefant
     lodjur
     giraff
     ```

   Förklara implementationen, och analysera programmets tidskomplexitet. Din analys ska ta hänsyn både till listans längd n, och antalet tecken w i den längsta strängen. Var tydlig med vilka antaganden du gör i din analys. Standarddatastrukturer och -algoritmer från kursen behöver inte förklaras, men däremot motiveras. Om pseudokod används måste den vara detaljerad.

2. *(Lätt.)* Betrakta följande Javaklass:

   ```
   // Trädnoder med föräldrapekare.
   //
   // Invariant: Om this.parent inte är null så gäller antingen
   //   this.parent.left  = this
   // eller
   //   this.parent.right = this.
   public class Node<A> {
       public A contents;      // Innehåll.
       public Node<A> left;    // Vänster delträd.
       public Node<A> right;   // Höger delträd.
       public Node<A> parent;  // Förälder.
   }
   ```

   Är följande implementation av enkelrotation korrekt? Motivera (gärna med figurer), och om något är fel, visa hur man kan åtgärda felet. (Eventuella problem har inget med Java att göra. Koden kompilerar utan problem.)

```
// Roterar upp noden child förbi sin förälder.
//
// Noden child samt dess förälder och förälderns förälder
// måste vara icke-null.
public static <A> void rotateUp(Node<A> child) {
    // Förfäder.
    Node<A> parent      = child.parent;
    Node<A> grandparent = parent.parent;

    // Uppdatera föräldrapekare.
    child.parent  = grandparent;
    parent.parent = child;

    // Uppdatera barnpekare.
    if (parent.left == child) {
        parent.left = child.right;
        child.right = parent;
    } else {
        parent.right = child.left;
        child.left   = parent;
    }

    // Uppdatera pekaren från "farföräldern".
    if (grandparent.left == parent) {
        grandparent.left  = child;
    } else {
        grandparent.right = child;
    }
}
```

3. 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 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 har följande operationer:

   `empty`**:** Skapar en tom avbildning.

   `insert`(k, v)**:** Lägger till en bindning $k \mapsto v$ till avbildningen. Om en bindning $k \mapsto v'$ redan finns i avbildningen så tas den bort.

   `lookup`(k)**:** Om en bindning $k \mapsto v$ finns i avbildningen så ger den här operationen v som svar, och annars meddelas på något lämpligt sätt att en sådan bindning saknas.

   `remove-smaller`(k)**:** Tar bort alla bindningar $k' \mapsto v$ där $k' < k$. Övriga bindningar påverkas inte.

   Den här uppgiften har graderade deluppgifter:

   För trea: Implementera ovanstående ADT, förklara hur din implementation fungerar, och analysera operationernas tidskomplexitet. Standarddatastrukturer och -algoritmer från kursen behöver inte beskrivas i detalj. För att bli godkänd måste du se till att operationerna har följande tidskomplexiteter (antingen i värsta fall, eller amorterat):

   `empty`**:** $O(1)$.

   `insert`**:** $O(\log n)$.

   `lookup`**:** $O(\log n)$.

   `remove-smaller`**:** $O(n \log n)$.

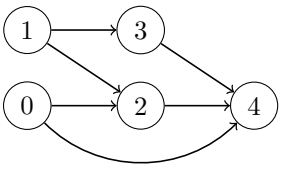
   Här är n antalet bindningar i avbildningen. Du kan anta att element kan jämföras på konstant tid.

   För fyra: Som för trea, men `remove-smaller` ska ha tidskomplexiteten $O(\log n)$.

4. Man kan sortera riktade acykliska grafer (DAGs) topologiskt genom att göra en djupet först-sökning och pusha noderna på en stack i postordning.

   För trea: Implementera den här algoritmen. Skriv en funktion (metod) som tar en graf och ger tillbaka en lista med grafens noder i topologisk ordning. Exempel:

   - Graf:

   - Möjligt resultat: 0, 1, 2, 3, 4.

   Var tydlig med hur du representerar grafer, och hur djupet först-sökningen utförs. Du behöver inte beskriva implementationer av stackar och listor i detalj.

   För fyra: Analysera din implementations tidskomplexitet (noggrant, det räcker inte att skriva "djupet först-sökning tar si och så lång tid", du måste analysera din egen implementation).

5. *(Svår.)* Betrakta följande variant av list-ADTn:

   `empty`: Skapar en tom lista.

   `insert`(x)**:** Lägger till x sist i listan.

   `delete-last`: Tar bort och ger tillbaka listans sista element. Krav: Listan får inte vara tom.

   Man kan implementera den här ADTn med en dynamisk array på följande sätt:[1]

   `empty`: Allokera en array med längd ett.

   `insert`(x)**:** Om arrayen är full: fördubbla arrayens storlek. Stoppa in x på den första lediga positionen.

   `delete-last`: Ta bort det sista elementet från arrayen. Om arrayens längd c är minst 4, och den innehåller $c/4$ element, halvera arrayens storlek. Ge tillbaka det sista elementet.

   Notera att arrayens längd hela tiden är 2^k för något $k \in \mathbb{N}$.

   Din uppgift är att bevisa att alla operationer tar amorterat konstant tid.

[1] Notera att den här pseudokoden *inte* är detaljerad.

6. *(Svår.)* Handelsresandeproblemet kan formuleras så här: Givet en komplett,[2] viktad, oriktad graf med minst två noder, hitta en kortaste hamiltonsk cykel. Med andra ord, hitta en väg som besöker alla noder exakt en gång, förutom att den börjar och slutar i samma nod, och är minst lika kort som alla andra sådana vägar.

   Lös följande variant av handelsresandeproblemet *effektivt*:

   - Kanternas vikter är ickenegativa. Du kan anta att de är naturliga tal.
   - Grafen uppfyller triangelolikheten: Låt $d(u, v)$ stå för vikten av kanten mellan noderna u och v. I så fall gäller för alla noder u, v och w att $d(u, w) \leq d(u, v) + d(v, w)$.
   - Du behöver inte hitta en kortaste hamiltonsk cykel, det räcker att hitta en hamiltonsk cykel som är max dubbelt så lång som de kortaste.

   Standarddatastrukturer och -algoritmer från kursen behöver inte förklaras. Motivera utförligt varför din lösning är korrekt och effektiv.

   Tips: Använd ett minsta uppspännande träd.

[2] I en komplett graf finns det en kant mellan noderna u och v om och endast om $u \neq v$.