Tentamen
Datastrukturer (DAT036)

- Datum och tid för tentamen: 2012-12-18, 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++) {
       t.insert(i);
   }
   ```

   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 obalanserat binärt sökträd som till att börja med är tomt.
   - Att den vanliga ordningen för heltal $(... < -1 < 0 < 1 < 2 < ...)$ används vid insättning i sökträdet.

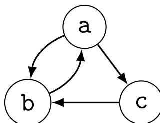
   Onödigt oprecisa analyser kan underkännas; använd gärna Θ-notation.

2. (a) *För trea:* Implementera en algoritm som, givet en riktad, oviktad graf, byter riktning på grafens alla kanter, och analysera algoritmens tidskomplexitet. Använd följande grafrepresentation:

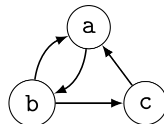
   ```
   public class Graph<A> {
       // Antal noder.
       // Noderna numreras från 0 till nodes - 1.
       private int nodes;

       // Array med nodinnehåll.
       private A[] contents;

       // Array med grannlistor.
       private List<Integer>[] adjacent;

       ...

       // Din uppgift.
       public void reverse() {
           ...
       }
   }
   ```

   Endast detaljerad kod (inte nödvändigtvis Java) godkänns. Du får inte anropa några andra metoder/procedurer, om du inte implementerar dem själv. *Rättelse (inte med i den ursprungliga tesen): Standardmetoder och -konstruerare för listor får användas.*

   Exempel: Om algoritmen appliceras på grafen

så ska resultatet vara följande graf:



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

(b) *För fyra:* Beskriv en grafrepresentation som möjliggör en $O(1)$-implementation av `reverse`. (Svaret måste motiveras ordentligt.)

3. Beskriv en algoritm som, givet en lista med antalet hårstrån på huvudet hos olika personer, avgör om minst två av de här personerna har samma antal hårstrån på huvudet. Algoritmen måste vara linjär (eller bättre) i antalet personer. Visa att så är fallet.

   Du får anta att det finns en övre gräns för antalet hårstrån på ett huvud (oberoende av algoritmens indata), t ex 200 000. Om du använder antagandet, var i så fall tydlig med hur det används.

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

   `new Map()` Konstruerar en tom avbildning.

   `insert(`k**`, `**v**`)`** Lägger till paret (k, v) till avbildningen. Om det redan finns ett par $p = (k, v')$ så tas p bort.

   `member(`k**`)`** Avgör om det finns något par (k, v) i avbildningen.

   `nth-smallest(`i**`)`** Får endast köras om i är ett positivt heltal och avbildningen innehåller minst i element. Om avbildningen innehåller paren $(k_1, v_1), (k_2, v_2), ...$ där $k_1 < k_2 < ...$ så ska operationens resultat vara v_i. Operationen ska alltså ta fram värdet v_i som svarar mot den i-te minsta nyckeln k_i.

   Du kan anta att alla nycklar och värden är heltal.

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

```
Map m = new Map();
m.insert(5, 5);
m.insert(5, 3);
m.insert(6, 0);
return m.member(5) && (! m.member(1)) &&
       m.nth-smallest(1) == 3 && m.nth-smallest(2) == 0;
```

   Operationerna måste ha följande tidskomplexiteter (där n är antalet par i avbildningen):

   - *För trea:* `new`: $O(1)$, `insert`, `member`: $O(\log n)$, `nth-smallest`: $O(n)$.

- *För fyra:* `new`: $O(1)$, `insert`, `member`, `nth-smallest`: $O(\log n)$.

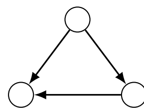
Visa att så är fallet. Implementationen av datastrukturen behöver inte beskrivas med detaljerad kod, men lösningen måste innehålla så mycket detaljer att tidskomplexiteten kan analyseras.

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

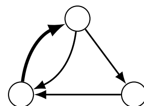
5. Visa att tidskomplexiteten för insättning av n element i en tom dynamisk array är $O(n)$. Du kan anta att varje element sätts in sist i arrayen, att arrayens längd till att börja med är 1, och att arrayens längd fördubblas vid insättning i en full array.

6. Beskriv en effektiv algoritm som avgör om en riktad graf kan göras starkt sammanhängande[1] genom att lägga till (som mest) en enda kant, och analysera dess tidskomplexitet.

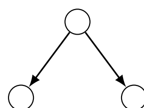
   Exempel: Grafen

   är inte starkt sammanhängande, men blir starkt sammanhängande om vi lägger till en kant:

   Grafen

   är inte heller starkt sammanhängande, och kan inte göras starkt sammanhängande genom att lägga till en enda kant.

   Tips: Börja med att lösa problemet för riktade acykliska grafer.

[1] En graf är starkt sammanhängande om det finns vägar från varje nod till varje annan nod.

Lösningsförslag för tentamen i Datastrukturer (DAT036) från 2012-12-18

Nils Anders Danielsson

1. När koden körs kommer talsekvensen 0, 1, 2, …, $\mathtt{n}-1$ att sättas in i trädet. Eftersom sekvensen är sorterad kommer trädet att vara maximalt obalanserat, och insättningarna linjära i trädets storlek. Tidskomplexiteten blir

$$\Theta\left(\sum_{i=0}^{\mathtt{n}-1} i\right) = \Theta(\mathtt{n}^2).$$

2. (a) En möjlig lösning (skriven i ett språk som liknar Java):

```
public void reverse() {
    // En ny array med grannlistor.
    List<Integer>[] rev =
      new LinkedList<Integer>[nodes];

    // Till att börja med är grannlistorna tomma.
    for (int i = 0; i < nodes; i++) {
        rev[i] = new LinkedList<Integer>();
    }

    for (int i = 0; i < nodes; i++) {
        for (Integer j : adjacent[i]) {
            // Det finns en kant från i till j i
            // grafen, så låt oss lägga till en kant
            // från j till i i den reverserade grafen.
            rev[j].add(i);
        }
    }

    adjacent = rev;
}
```

Algoritmen utför $O(1)$ arbete för varje nod och kant, så den är linjär i grafens storlek.

(b) Ett förslag: Använd två arrayer med grannlistor, en för direkta efterföljare och en för direkta föregångare. Då kan man reversera grafen

genom att byta ut den ena arrayen mot den andra, och vice versa, på konstant tid (genom att ändra två pekare).

3. Jag gissar att många kommer att använda en hashtabell, men jag ger några andra lösningar.

   Låt M vara det maximala antalet hårstrån på ett huvud.

   En lösning: Sortera listan. Gå sedan igenom resultatet och avgör om två intilliggande värden är lika. Om vi använder radixsortering är tidskomplexiteten $O(d(N+n))$, där n är listans längd, N antalet olika siffror som används i talrepresentationen, och d antalet siffror i det största talet i listan. I vårt fall kan vi t ex välja $N = 2$,[1] vilket ger $d = O(\log M) = O(1)$ och den totala tidskomplexiteten $O(n) + O(n) = O(n)$.

   En annan lösning: Om listans längd är minst $M + 2$ så säger duvslagsprincipen att det finns två personer i listan med samma antal hårstrån på huvudet, så vi kan ge "ja" som svar utan att inspektera talen. För kortare listor kan vi använda algoritmen ovan. Totala tidskomplexiteten blir $O(1)$ (med en stor "konstant").

4. *För trea:* Använd AVL-träd med par av nycklar och värden i noderna, och implementera `new`, `insert` och `member` som vanligt (de här operationerna har rätt tidskomplexitet). Låt `nth-smallest(`i`)` gå igenom trädet i inordning, och ge värdet i nod nummer i som svar; värstafallstidskomplexiteten för den här operationen är $O(n)$.

   För fyra: Använd AVL-träd med ett extra storleksfält i noderna, så att varje nod innehåller information om hur många noder som finns i delträdet för vilket noden är rot:

```
public class Node {
    public  int key, value;
    private int height, size;
    private Node left, right;

    // Konstruerare och/eller metoder (alla O(1)) som
    // säkerställer att height och size uppdateras på ett
    // korrekt sätt (så länge man inte introducerar cykler
    // i trädet).

    ...

    // Vänster barn.
    public Node left() {
        return left;
    }
```

[1] Kanske inte det bästa valet, men det duger här.

```
        // Höger barn.
        public Node right() {
            return right;
        }

        // Storleken av trädet rotat i n (som får vara null).
        public static int size(Node n) {
            return n == null ? 0 : n.size;
        }
    }
```

Implementera `new`, `member` och `insert` på ungefär samma sätt som de vanliga AVL-trädsoperationerna (som har rätt tidskomplexiteter); vid insättning behöver man ibland ändra på storleksfälten, men om man använder ”rätt” implementationsteknik så kan det hanteras av de utelämnade `Node`-konstruerarna och -metoderna ovan. Implementera till sist `nth-smallest` på följande sätt:

```
public int nth-smallest(int i) {
    return nth-smallest(i, root);
}

// Hittar i-te noden (räknat från 1) i trädet rotat i n.
// Precondition: 1 <= i <= trädets storlek.
public int nth-smallest(int i, Node n) {
    assert 1 <= i && i <= Node.size(n);

    // Det vänstra delträdets storlek.
    int leftSize = Node.size(n.left());

    if (leftSize >= i) {
        return nth-smallest(i, n.left());
    } else if (leftSize + 1 == i) {
        return n.value;
    } else {
        return nth-smallest(i - (leftSize + 1), n.right());
    }
}
```

I värsta fallet är `nth-smallest` linjär i trädets höjd: $\Theta(\log n)$.

5. Se till exempel materialet från den föreläsning som gavs 2012-11-05.

6. Antag att en riktad, ändlig graf $G = (V, E)$ är given. Betrakta den underliggande acykliska grafen $G' = (V', E')$, där

$$V' = \{\, C \subseteq V \mid C \text{ består av noderna hos en SCC i } G \,\}$$

och

$$E' = \{ (c_1, c_2) \in V'^2 \mid \exists v_1 \in c_1, v_2 \in c_2 . \, (v_1, v_2) \in E \} .$$

Om G representeras på lämpligt sätt kan vi konstruera en grannlisterepresentation av G' (där varje nod representeras av ett tal, inte en delmängd av V) på linjär tid, $\Theta(|V| + |E|)$, med hjälp av en effektiv SCC-algoritm.

Betrakta nu följande fyra uttömmande fall:

- G' är tom: Då är G starkt sammanhängande.
- G' innehåller exakt en nod s utan ingående kanter och exakt en nod t utan utgående kanter: Då kan G göras starkt sammanhängande genom att lägga till som mest en kant från SCCn svarande mot t till SCCn svarande mot s.
- G' innehåller minst två noder utan ingående kanter: Då finns det två SCCer i G utan ingående kanter, så det krävs minst två kanter för att göra G starkt sammanhängande.
- G' innehåller minst två noder utan utgående kanter: Då krävs det också, på motsvarande sätt, minst två kanter för att göra G' starkt sammanhängande.

Vi kan alltså lösa problemet genom att beräkna antalet noder i G' utan ingående respektive utgående kanter, vilket om G' representeras med grannlistor kan göras på linjär tid ($O(|V'| + |E'|) = O(|V| + |E|)$).

Tidskomplexiteten blir $\Theta(|V| + |E|)$, vilket (åtminstone med en liten ”konstant”) nog får sägas vara effektivt.