Tentamen
Datastrukturer (DAT036)

- Datum och tid för tentamen: 2012-08-24, 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 kods tidskomplexitet, givet att `m` och `n` är naturliga tal, att `stack` är en stack som till att börja med har längden |`stack`|, och att typen `int` kan representera alla heltal:

```
for(int r = 0; r < m; r++) {
  for(int c = 0; c < n; c++) {
    stack.push(c);
  }
}
```

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

2. Uppgiften är att konstruera en datastruktur som representerar ”dubbelriktade maps” (bijektioner mellan ändliga mängder).

   Exempel: Låt $S = \{ 1, 2, 3 \}$ och $T = \{ 3, 4, 5 \}$. Målet är att kunna representera de relationer mellan S och T som relaterar varje element i S med exakt ett element i T, och vice versa. Följande sex relationer är möjliga:

$1 \leftrightarrow 3$	$1 \leftrightarrow 3$	$1 \leftrightarrow 4$	$1 \leftrightarrow 4$	$1 \leftrightarrow 5$	$1 \leftrightarrow 5$
$2 \leftrightarrow 4$	$2 \leftrightarrow 5$	$2 \leftrightarrow 3$	$2 \leftrightarrow 5$	$2 \leftrightarrow 3$	$2 \leftrightarrow 4$
$3 \leftrightarrow 5$	$3 \leftrightarrow 4$	$3 \leftrightarrow 5$	$3 \leftrightarrow 3$	$3 \leftrightarrow 4$	$3 \leftrightarrow 3$

   Datastrukturen ska ha följande operationer:

   `Bijection` Konstruerar en bijektion mellan två tomma mängder.

   `insert(`s`,`t`)` Anta att bijektionen relaterar mängderna S och T. Om s redan finns i S eller t redan finns i T så ska operationen rapportera att ett fel inträffat. Annars läggs paret $s \leftrightarrow t$ till bijektionen (som då relaterar mängderna $S \cup \{ s \}$ och $T \cup \{ t \}$).

   `target(`s`)` Om det finns ett par $s \leftrightarrow t$ i bijektionen så ges t som svar, annars `null`.

   `source(`t`)` Om det finns ett par $s \leftrightarrow t$ i bijektionen så ges s som svar, annars `null`.

   Du kan anta att alla element är heltal. Analysera operationernas tidskomplexiteter och se till så att varje operation har tidskomplexiteten $O(\log n)$, där n är antalet bindningar.

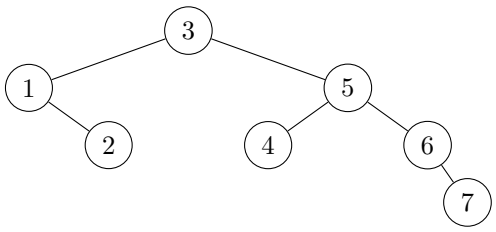
   Exempel: Följande kod, skriven i ett Javaliknande språk, ska ge `true` som svar:

```
Bijection b = new Bijection();
b.insert(1, 2);
b.insert(2, 3);
return b.target(1) == 2 && b.source(2) != 3;
```

   Tips: Konstruera inte datastrukturen från grunden, bygg den hellre m h a standarddatastrukturer.

3. *För trea:* Beskriv en algoritm som konverterar en sorterad array till ett AVL-träd.

   Exempel: $[1, 2, 3, 4, 5, 6, 7]$ kan konverteras till följande träd:



   Tips: Det är fritt fram att använda standarddatastrukturer och -algoritmer från kursen.

   För fyra: Som för trea, men algoritmen måste vara linjär i antalet element ($O(n)$, där n är arrayens längd). Visa att så är fallet.

   Du kan (men måste inte) anta att AVL-träd representeras av följande Javaklass (där vi låtsas att typen `int` kan representera alla heltal), och att du har tillgång till klassens privata delar:

```
public class AVLTree<A extends Comparable<? super A>> {
  // Trädnoder. Tomma träd representeras av null.
  private class TreeNode {
    A        contents;  // Innehåll.
    TreeNode left;      // Vänstra barnet.
    TreeNode right;     // Högra barnet.
    int      height;    // Trädets höjd.

    // Skapar en trädnod. Krav: Skillnaden mellan de två
    // delträdens höjder får vara max 1.
    TreeNode(TreeNode left, A contents, TreeNode right) {
      int leftHeight  = left  == null ? -1 : left.height;
      int rightHeight = right == null ? -1 : right.height;

      assert(Math.abs(leftHeight - rightHeight) <= 1);

      this.contents = contents;
      this.left     = left;
      this.right    = right;
      this.height   = 1 + Math.max(leftHeight, rightHeight);
    }
  }

  // Roten.
  private TreeNode root;

  ...
}
```

4. *För trea:* Beskriv en algoritm som, givet en ändlig mängd variabler V och en mängd strikta olikheter $O \subseteq \{ x < y \mid x, y \in V \}$, avgör om variablerna kan bytas ut mot heltal på ett sådant sätt att alla olikheterna uppfylls (samtidigt). Här står $<$ för den vanliga strikta olikheten $\ldots < -2 < -1 < 0 < 1 < 2 < \ldots$.

   Exempel:

V	O	Resultat
$\{ a, b, c, x, y \}$	$\{ a < b, x < y \}$	Ja
$\{ x, y, z \}$	$\{ x < y, y < z \}$	Ja
$\{ x, y, z \}$	$\{ x < y, y < x \}$	Nej
$\{ \}$	$\{ \}$	Ja

   För fyra: Som för trea, med följande skillnader:

   - Om olikheterna kan uppfyllas ska algoritmens resultat utökas med ett exempel på en korrekt variabeltilldelning (d v s en variabeltilldelning som gör så att alla olikheterna uppfylls).
   - Algoritmen måste vara effektiv, och algoritmens värstafallstidskomplexitet ska analyseras. Redovisa algoritmens detaljer tillräckligt noggrant för att kunna analysera tidskomplexiteten (det kan till exempel vara lämpligt att dokumentera hur du representerar mängder).

   Exempel:

V	O	Resultat
$\{ a, b, c, x, y \}$	$\{ a < b, x < y \}$	Ja: $a = c = x = 0, b = y = 1$
$\{ x, y, z \}$	$\{ x < y, y < z \}$	Ja: $x = 0, y = 1, z = 2$
$\{ x, y, z \}$	$\{ x < y, y < x \}$	Nej
$\{ \}$	$\{ \}$	Ja:

5. Betrakta följande Javaklass, som representerar dubbellänkade listor:

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

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

6. Låt oss implementera en kö på följande sätt:

   Tillståndsvariabler Kön består av två stackar:

   ```
   private Stack<A> front, back;
   ```

   Den första stacken, ”framstacken”, innehåller början av kön. Den andra, ”bakstacken”, innehåller resten av kön, men i omvänd ordning. Till att börja med är båda köerna tomma.

   Exempel: Låt oss använda notationen (f, b) för en kö med framstack f och bakstack b. Kön $[1, 2, 3, 4, 5]$ kan representeras av $([1, 2], [5, 4, 3])$. Den kan också representeras av $([1], [5, 4, 3, 2])$.

   `enqueue(a)` Elementet `a` läggs till sist i kön genom att det stoppas in först i bakstacken:

   ```
   back.push(a);
   ```

   Exempel: Om `enqueue(6)` appliceras på $([1], [5, 4, 3, 2])$ blir resultatet $([1], [6, 5, 4, 3, 2])$.

   `dequeue` Köns första element tas ut på följande sätt (ni kan anta att den här operationen bara appliceras på icketomma köer):

   - Först testas det om framstacken är tom. I så fall flyttas alla element från bakstacken över till framstacken:

     ```
     if (front.isEmpty()) {
       while (! back.isEmpty()) {
         front.push(back.pop());
       }
     }
     ```

   - Sedan poppas framstackens första element:

     ```
     return front.pop();
     ```

   Exempel: Om `dequeue` appliceras på $([1], [5, 4, 3, 2])$ blir resultatet 1 samt den nya kön $([\,], [5, 4, 3, 2])$. Om `dequeue` appliceras på den kön blir resultatet 2 samt den nya kön $([3, 4, 5], [\,])$.

   Bevisa att de två operationerna ovan tar amorterat konstant tid (givet att en kö bara kan inspekteras eller modifieras m h a de operationerna).