CHALMERS | GÖTEBORGS UNIVERSITET

Institutionen för data- och informationsteknik

TENTAMEN

KURSNAMN	**Algoritmer och datastrukturer, 7.5p**
PROGRAM	**TIDAL-2, TKIEK-2** **LP4 – 2020**
KURSBETECKNING	**LET375**
EXAMINATOR	**Peter Ljunglöf**
TID FÖR TENTAMEN	**Fredag 2020–10–09, 14:00**
HJÄLPMEDEL	**Se slutet på nästa sida**
ANSVARIG LÄRARE	**Pelle Evensen, se https://chalmers.instructure.com/courses/12697/pages/instruktioner-for-e-examination för hur Pelle kontaktas under tentamen.**
DATUM FÖR ANSLAG	**Senast 2020–10–29.**
ÖVRIG INFORMATION	**Betygsgränser: Se nästa sida**

OMTENTAMEN

Algoritmer och datastrukturer

- Skriv rent dina svar. Oläsliga svar *rättas ej!*
- Om du lämnar flera olika svar på *samma uppgift* i *samma fil* eller olika filer (olika filnamn där det inte tydligt framgår vilken som är den sista) med svar på samma uppgift kan det bli så att svaret inte bedöms.
- Oprecisa eller alltför generella (vaga) svar ger inga poäng. Konkretisera och/eller ge exempel. Det är aldrig någon risk att vara övertydlig!
- Programkod skall skrivas i Java 5, eller senare version, och vara indenterad och renskriven.
- Onödigt komplicerade lösningar ger poängavdrag.
- Givna deklarationer, parameterlistor etc. *får inte ändras* om det inte uttryckligen är en del av uppgiften. Fråga i oklara fall!
- Läs igenom tentamenstesen och förbered eventuella frågor.
- Beskrivning av tillåtna/otillåtna hjälpmedel samt hur lärare kontaktas under tentamen finns beskrivet på kurshemsidan: https://chalmers.instructure.com/courses/12697/pages/instruktioner-for-e-examination

Tentamenspoäng och betygsgränser

Tesen innehåller 6 grundläggande och 3 avancerade frågor. Varje fråga kan ge maximalt 2 poäng.

Maxpoäng är följaktligen 18p (12p grundläggande + 6p avancerade). Avancerade poäng kan inte tillgodoräknas som grundläggande. Följande tabell används för betyg:

Betyg	Grundläggande poäng	Avancerade poäng
3	≥ 8	–
4	≥ 9	≥ 2
5	≥ 10	≥ 4

Lycka till!

Uppgift 1, grundläggande: Komplexitet & korrekthet

Betrakta följande funktion:

```
/**
 * @pre s != null && s is sorted in ascending order.
 * @param s the array in which to do a lookup.
 * @param v the value to look for.
 * @return true if v was in s
 * @return false otherwise.
 */
public static boolean vIsInS(final int[] s, final int v) {
   int startIdx = 0;
   int endIdx = s.length - 1;

   while(startIdx <= endIdx) {
      final int candidateIdx = (startIdx + endIdx) / 2;
      if(s[candidateIdx] == v) {
         return true;
      } else if(s[candidateIdx] < v) {
         startIdx = candidateIdx;
      } else {
         endIdx = candidateIdx;
      }
   }
   return false;
}
```

Tanken är att funktionen ska avgöra om värdet `v` finns med i arrayen `s`.

Tyvärr fungerar programmet inte som avsett; för vissa giltiga indata så hänger det sig och kommer aldrig ur while-loopen.

(A) Ge ett exempel på indata för vilket funktionen hänger sig.

(B) Åtgärda felet/felen, så att metoden inte hänger sig för några giltiga indata.

(C) Ge en så snäv värstafallskomplexitet för `vIsInS()` som möjligt. Använd S för längden på arrayen `s`. Motivera ditt svar.

Uppgift 2, grundläggande: Hashtabeller

Rita den hashtabell med closed hashing (open addressing) du får med följande värden för uppgift 2: https://chalmers.instructure.com/courses/12697/pages/probleminstanser

`X:Y` är nyckeln `X` som här hashar till värdet `Y`.

Hashtabellens storlek är $m = 10$.

Låt h_0 vara det ursprungliga hashvärdet (mod m).

Om en kollision sker så fås det nya hashvärdet genom formeln

$h_n = h_{n-1} + 2h_0 + 1 \pmod{m}$.

Vi gör alltså nästa probning med en steglängd som avgörs av det ursprungliga hashvärdet.

Sätt in värdena i den ordning du fått dem från problemsidan.

Ditt svar ska innehålla följande:

(A) Nycklarna och deras hashvärden.

(B) Din tabell, där du markerat vilka element som kolliderat och hur probningen ser ut för att hantera kollisionerna.
Markera varje probning med `X-a` på rätt ställe i tabellen där `X` är nyckeln och `a` är antalet probningar för nyckeln så långt. Ringa in `X-a` för att markera `X` slutgiltiga position i tabellen.
Exempel: Om `A` ska sättas in och första probningen är en kollision, skriv `A-1` för första probningen (på rätt tabellindex), sen `A-2`, o.s.v.

Exempel (antag $m = 6$ och ospecificerad kollisionshantering)

`0:` (A-0)

`1:` (C-0) `D-0`

`2:` (D-2)

`3:` (B-1) `D-1`

`4:` (E-0) `B-0`

`5: -`

Uppgift 3, grundläggande: Prioritetsköer

Vi ska här skapa en speciell prioritetskö med följande metoder:

```
public class SecondaryQueue<T extends Comparable<T>> {
    // Creates a new empty queue.
    public SecondaryQueue() {
        // Your implementation here.
    }

    // Removes and returns the second element, if it exists.
    // Otherwise, throws NoSuchElementException
    public T removeSecond() {
        // Your implementation here.
    }

    // Hint: Implement this method first and see hint below about using
    //       existing classes.
    private T removeFirst() {
        // Your implementation here.
    }

    // Adds e to the queue. It may be assumed that e is non-null.
    public void insert(T e) {
        // Your implementation here.
    }
}
```

Följande sekvens illustrerar det förväntade beteendet:

```
SecondaryQueue<Integer> q = new SecondaryQueue();
q.insert(1); q.insert(3); q.insert(2); q.insert(4); // q is now {1, 2, 3, 4}
q.removeSecond(); // Returns 2
q.removeSecond(); // Returns 3
q.removeSecond(); // Returns 4
q.removeSecond(); // Throws exception.
```

Skapa en implementation av klassen `SecondaryQueue`.

För full poäng så ska `removeSecond` och `insert` (som sämst) ha amorterad värstafallskomplexitet O(log n), då kön har n element.

Ditt svar ska innehålla följande:

(A) Fungerande källkod för konstruktorn, `removeSecond` och `insert`.

(B) En beskrivning och motivering av komplexiteten för konstruktorn, `removeSecond` och `insert`.

Tips: Du får använda alla standardklasser i JDK5 och senare. Du måste dock motivera vilken komplexitet du förväntar dig av färdiga klasser/metoder om du använder dig av dem (rekommenderas starkt).

Uppgift 4, grundläggande: Grafer

Den oriktade grafen nedan har 8 noder samt 17 kanter.

Med den startnod, ”START”, och slutnod, ”SLUT”, samt värde R som fås genom problemsidan: https://chalmers.instructure.com/courses/12697/pages/probleminstanser

(1) Sätt ut vikter på kanterna så att den kortaste vägen från START till SLUT har vikten exakt R.

(2) Kanternas vikter ska också uppfylla kriteriet att det minimala uppspännande trädet (minimal spanning tree, MST) har vikten R + 9.

Ditt svar ska innehålla följande:

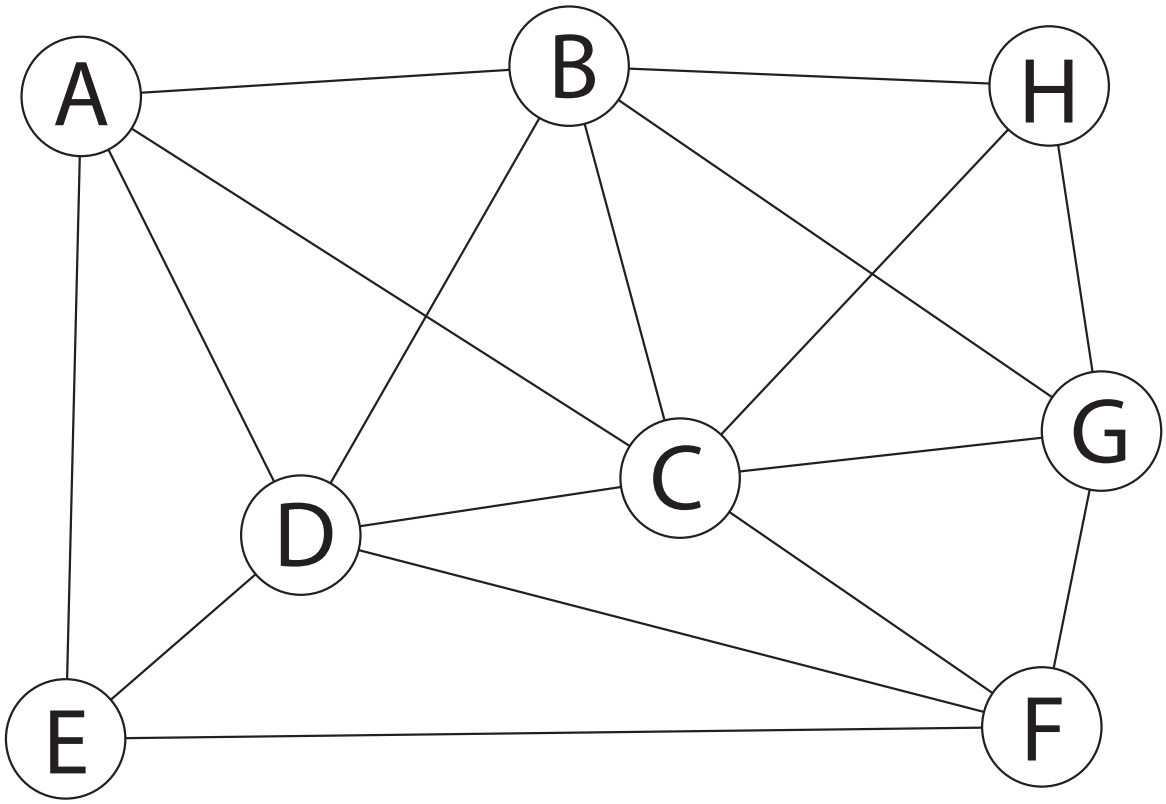
(A) START, SLUT och R som du fått genom att följa länken ovan.

(B) Grafen där alla kanters vikter tydligt syns. Förvissa dig om att din graf är lätt att läsa. Grafen ska alltså vara densamma som på figuren nedan, men med vikter utsatta.

(C) Den kortaste vägen mellan START och SLUT som en lista av noder, START → ... → SLUT. Summan av kanternas vikter ska vara R.

(D) Ditt MST, tydligt markerat/där du visar vilka kanter som ingår, samt deras summa (= R + 9).

Tips: Det är valfritt att visa hur du gått tillväga för att skapa ditt MST och din kortaste väg.

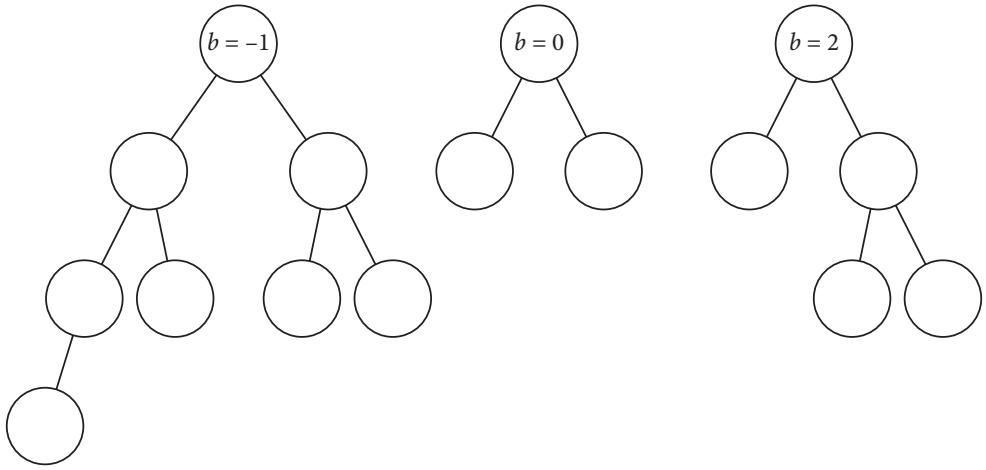


CHALMERS
Institutionen för data- och informationsteknik
Pelle Evensen, evensen@chalmers.se

Uppgift 5, grundläggande: Binära sökträd

Ett binärt sökträd kan vara olika mycket obalanserat.

Vår definition av ”balanserat” kan illustreras med följande figur:



Med utgångspunkt i en nod; balansen är (antalet barn till höger) – (antal barn till vänster). Om vänsternoderna är fler än högernoderna blir balansen negativ. Om högernoderna är fler än vänsternoderna blir balansen positiv. Om höger- och vänsternoderna är lika många blir balansen 0.

Uppgiften är att beräkna ”balanseringsindex” för en nod (eller ett träd, om noden är rotnoden i trädet). Detta index ska fås genom att man på en nod anropar metoden `getBalance()`.

Källkod för nod-klassen finns på nästa sida.

Ditt svar ska innehålla följande:

(A) Fungerande källkod för `getBalance()`.

(B) En kort förklaring till varför din källkod går i tid O(n), där n är antalet barn/ättlingar från en viss nod.

(C) En kort motivering till varför din implementation av `getBalance()` *inte kan ha lägre* tidskomplexitet än O(n) givet att `getBalance()` är den enda metoden du får implementera i `Node`.

```
public class Node<T extends Comparable<T>> {
    private T value;
    private Node<T> left, right; // null if no left/right child.
    private Node<T> right;

    public Node(final T v) {
        this.value = Objects.requireNonNull(v);
    }

    public T getValue() {
        return this.value;
    }

    // null if no left child.
    public Node<T> getLeftChild() {
        return this.left;
    }

    // null if no right child.
    public Node<T> getRightChild() {
        return this.right;
    }

    public void add(final T v) {
        // Details omitted; adds v to the subtree with this node
        // as its root.
    }

    // Returns how unbalanced the subtree with this node as its root is.
    public int getBalance() {
        // Your implementation.
    }
}
```

CHALMERS
Institutionen för data- och informationsteknik
Pelle Evensen, evensen@chalmers.se

Uppgift 6, grundläggande: Sortering

I vissa sammanhang kan det vara bra att identifiera *längsta stigande delsekvenser* i en sekvens. Vi kallar en sån delsekvens ”LRSS” för ”Longest Rising SubSequence”. Två på varandra följande identiska element betraktar vi som stigande.

```
// Input: an array S
// Output: the start index (inclusive) and end index (exclusive) of the longest
//         rising subsequence.
// If there are several LRSSs of the same length,
// the indices for the first one shall be returned.


findLRSS(S)

   // YOUR CODE GOES HERE.

   return (startIdx, endIdx)
```

Exempel på in- och utdata:

Indata	Utdata	Indata	Utdata
{ }	(0, 0)	{1, 2, 3, 2, 3, 4, 5, 1}	(3, 7)
{1}	(0, 1)	{3, 2, 1}	(0, 1)
{1, 2}	(0, 2)	{1, 2, 1, 1, 2, 1}	(2, 5)

Ditt svar ska innehålla följande:

(A) Dina två exempelarrayer `S1` och `S2` som fås genom genom https://chalmers.instructure.com/courses/12697/pages/probleminstanser

(B) Din (pseudo-)kod för `findLRSS` ovan. För poäng så ska `findLRSS` ha linjär värstafallskomplexitet i storleken hos S.

(C) Resultatet av din implementation applicerad på `S1` och `S2`.

Uppgift 7, avancerad: Komplexitet

Betrakta följande funktion för att sortera en array av strängar och samtidigt ta bort duplikat:

```
public static Long[] sortNoDuplicates(final Long[] array) {
    final List<Long> output = new ArrayList<>();
    final Set<Long> set = new HashSet<>();

    for(final Long s : array) {
        if(!set.contains(s)) {
            set.add(s);
            output.add(s);
        }
    }

    // You may assume sort is in O(n log n).
    Collections.sort(output);

    return output.toArray(new Long[0]);
}
```

Beskriv komplexiteten för `sortNoDuplicates()` som en funktion av indatats storlek, n = `array.length` och m = antalet olika/distinkta element i `array`.

Du får anta att jämförelser och beräkning av hashvärden tar O(1) tid och att en bra hashfunktion används.

Du måste motivera ditt svar; att bara nämna komplexiteten räcker inte.

För 1 poäng:

Korrekt analys med värstafallskomplexitet som en funktion av n och m.

För 2 poäng:

Korrekt analys med både värstafalls- och genomsnittskomplexitet som en funktion av n och m.

Uppgift 8, avancerad: Annorlunda traversering

Givet är följande (något förenklade) Javakod för ett binärt träd:

```
public class BinaryTree<T> {
    private class Node {
        T value;
        Node left;
        Node right;
    }

    private Node root;

    void printNodesFromMiddle( ) {
       // Your code here.
    }
}
```

Din uppgift är att skriva (pseudo-)kod som traverserar ett binärt träd av typen ovan(inte nödvändigtvis ett binärt *sök*träd) genom att i första hand ta noder som ligger så nära mitten av trädet som möjligt.

För trädet i illustrationen på nästa sida så ska först värdet hos alla noder i det grå fältet markerat ”1” skrivas ut, sedan noderna i fält 2, följt av fält 3, 4, 5 och 6. Om två noder eller delträd ligger lika långt från mitten så ska vänstersidan avverkas först.

Givet att trädet på nästa sida finns i variabeln `tree` så ska `tree.printNodesFromMiddle( )` ge utskriften ”`A E F B I C D G H`” (eller t.ex. ”`F A E I B C D G H`”).

Delsekvenserna AEF och BI kan skrivas i vilken inbördes ordning som helst men A, E och F måste komma före B och I. ”`F A E I B C D G H`” är alltså en lika giltig utskrift för `tree.printNodesFromMiddle( )`.

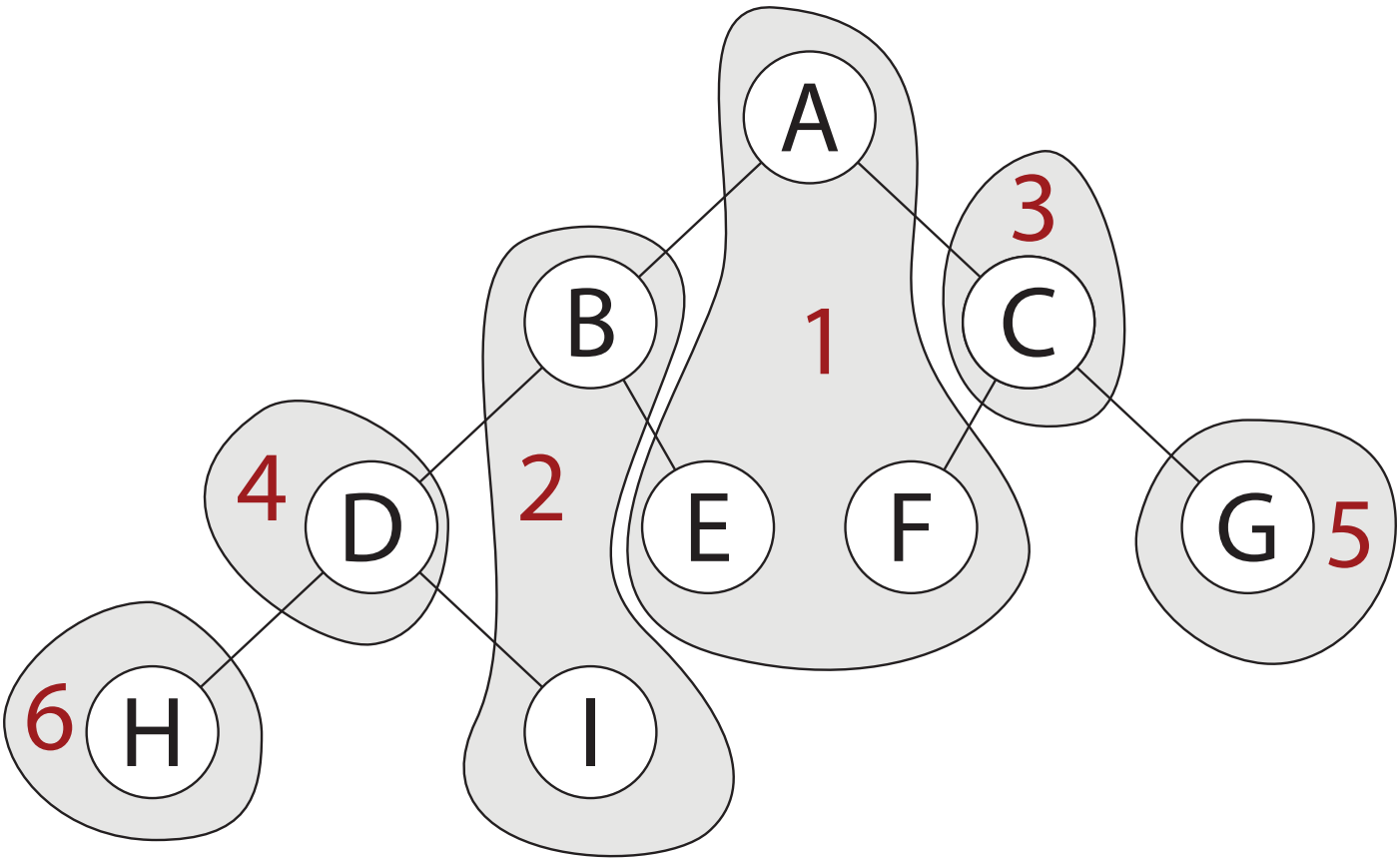
För 1 poäng:

Fungerande pseudokod som löser uppgiften korrekt.

För 2 poäng:

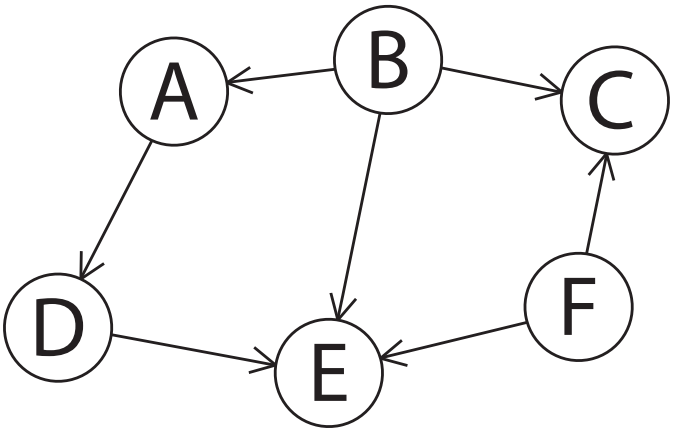
Värstafallskomplexiteten får inte vara högre (men kan vara lägre) än $O(n \log n)$ där n är antalet noder i trädet – du måste förklara varför detta uppnås.

Tips: Till hjälp får du får använda dig av vilka standardklasser i JDK5, eller senare, du vill (men den förväntade komplexiteten hos operationerna ska nämnas). Formateringen hos utskriften är inte viktig.

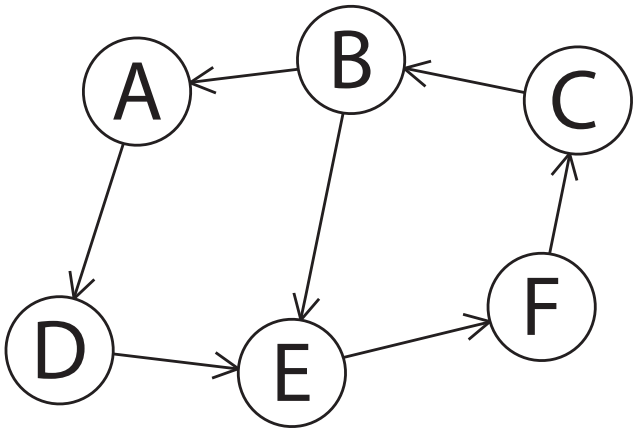

A
B
C
D
E
F
G
H
I
1
2
3
4
5
6

Uppgift 9, avancerad: Riktade acykliska grafer

Betrakta följande två riktade grafer:



Graf 1



Graf 2

Den vänstra grafen innehåller inga cykler men den högra innehåller två;
{A, D, E, F, C, B, A} och {B, E, F, C, B}.

Givet pseudokod för klassen `Digraph` på nästa sida, på nästa sida, visa exekveringen av `isAcyclic` på lämpligt vis där `graphNodes` kommer i den ordning du får från sidan med probleminstanser:
https://chalmers.instructure.com/courses/12697/pages/probleminstanser

Detta ska göras för både graf 1 och graf 2 ovan.

För 1 poäng:

En demonstration av hur `isAcyclic` körs, på både graf 1 och graf 2, givet att `graphNodes` loopas över i den ordning du fått från probleminstanssidan.

För 2 poäng:

Ge en kort motivering till vad komplexiteten för `isAcyclic` är, som en funktion av |E|, antalet kanter och |V|, antalet noder. Du behöver nämna vilka antaganden du gör vad gäller operationer på listor och mängder.

```
Digraph {
   graphNodes = empty set

   inner class Node {
      String id
      edgesTo = new empty list of nodes

      Node(x) {
         id = x
      }

      void addEdge(Node destination) {
         add destination to E
      }
   }

   boolean internalIsAcyclic(Node origin,
                             Set visited,
                             Set tmpVisit) {
      if origin is in tmpVisit
         return false

      if origin is in visited
         return true

      add origin to visited
      add origin to tmpVisit

      for each node n in origin.edgesTo
         if not internalIsAcyclic(n, visited, tmpVisit)
            return false

      remove origin from tmpVisit
      return true
   }

   boolean isAcyclic() {
      visited = new empty set
      tmpVisit = new empty set

      for each n in graphNodes
         if not internalIsAcyclic(n, visited, tmpVisit)
            return false

      return true
   }
}
```