Tentamen
Datastrukturer, DAT037 (DAT036)

- Datum och tid för tentamen: 2017-01-11, 14:00–18:00.
- Ansvarig: Fredrik Lindblad. Nås på tel nr. 031-772 2038. Besöker tentamenssalarna ca 15:00 och ca 17:00.
- Godkända hjälpmedel: Ett A4-blad med handskrivna anteckningar.
- Tentan innehåller 6 uppgifter. Varje uppgift får betyget U,3,4 eller 5.
- För att få betyget n (3, 4 eller 5) på tentan måste man få betyget n eller högre på minst n uppgifter.
- En helt korrekt lösning av en uppgift ger betyget 5 på den uppgiften. Lösningar med enstaka mindre allvarliga misstag kan *eventuellt* ge betyget 5, och sämre lösningar kan ge lägre betyg.
- Betyget kan i undantagsfall, med stöd i betygskriterierna för DAT036/ DAT037, efter en helhetsbedömning av tentan bli högre än vad som anges ovan.
- 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. Om inget annat anges får pseudokod 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 (sådant som har gåtts igenom på föreläsningarna), men däremot motivera deras användning.
- Efter rättning är tentamen tillgänglig vid expeditionen på plan 4 i EDIT-huset, och kan granskas där. Den som vill diskutera rättningen kan, inom tre veckor efter att resultatet har rapporterats, kontakta ansvarig lärare och boka tid för ett möte. Notera att tentamen i så fall inte ska tas med från expeditionen.

1. Analysera nedanstående kods tidskomplexitet, uttryckt i `n`:

```
for (int i = 0; i < n; i++) {
    b.addLast(s.member(a.getAt(i)));
}
```

   Använd kursens uniforma kostnadsmodell, och gör följande antaganden:

   - Att `int` kan representera alla heltal och att `n` är ett positivt heltal.
   - Att `a` är en dynamisk array av heltal med `n` element.
   - Att `s` är en mängd av heltal som är implementerad med ett AVL-träd och innehåller `n` element.
   - Att `b` är en dynamisk array av boolska värden, som från början är tom.
   - Att jämförelserna av heltal som utförs i AVL-trädet sker på konstant tid.

   Analysen ska bestå av en matematisk uträkning av tidskomplexiteten och ska hänvisa till programkoden. För datastrukturernas operationer kan du hänvisa till standardimplementeringarnas komplexitet. Svara med ett enkelt uttryck (inte en summa, rekursiv definition, eller dylikt). Onödigt oprecisa analyser kan underkännas.

2. Uppgiften handlar om binära träd som är representerade med följande klass:

```
public class Tree<A> {
    // Trädnoder; null representerar tomma träd.
    private class Node {
        A    contents;  // Innehåll.
        Node left;      // Vänstra barnet.
        Node right;     // Högra barnet.
    }

    private Node root;  // roten
    ...
}
```

Du har två alternativ:

i) *För trea:* Implementera metoden
`int height()`
som beräknar trädets höjd.

ii) *För fyra eller femma:* Implementera metoden
`boolean isAVLBalanced()`
som avgör om trädet är höjdbalanserat enligt det krav som gäller för AVL-träd.

Lös endast ett av de två alternativen. För båda alternativen måste metodens tidskomplexitet vara linjär i trädets storlek ($O(n)$, där n är storleken) *och* du måste visa att så är fallet.

Endast detaljerad kod godkänns, ej pseudokod. Att använda hjälpmetoder är tillåtet, men du får inte anropa några andra metoder, om du inte implementerar dem själv.

3. Konstruera en datastruktur för riktade, viktade grafer där nodernas etiketter är heltal och vikterna är icke-negativa heltal. Ange hur grafen representeras, d.v.s. vilka datastrukturer och variabler som används för att lagra grafen. Notera att valet av representation kan ha betydelse för tidskomplexiteterna hos operationerna nedan.

   Implementera därefter operationerna:

   `addNode(`i`)` som lägger till en nod med etiketten i till grafen. Om i redan finns är grafen oförändrad.

   `addEdge(`i`, `j`, `w`)` som lägger till en kant från nod i till nod j med vikten w.

   `closestFromNode(`f`, `t`)` där f är en lista av distinkta noder (heltal) och t är en nod. Operationen ska returnera en nod (ett heltal) bland dem i f som har kortast väg till noden t. Om det inte finns någon väg från någon av noderna i f till noden t så ska operationen returnera `null`.

   För grafrepresentationen och operationerna kan du använda följande standarddatastrukturer och dess standardoperationer utan att ange hur de implementeras: dynamisk array, stack, FIFO-kö, binär heap, AVL-träd och hashtabell. Operationen för att ta bort ett visst namngivet element ur binär heap kan antas ta logaritmisk tid. Uppslag och insättning i hashtabell kan antas ta konstant respektive amorterad konstant tid.

   Låt v vara antalet noder och e antalet bågar i grafen. Låt n vara längden på listan f. Operationerna ska ha följande tidskomplexitet:

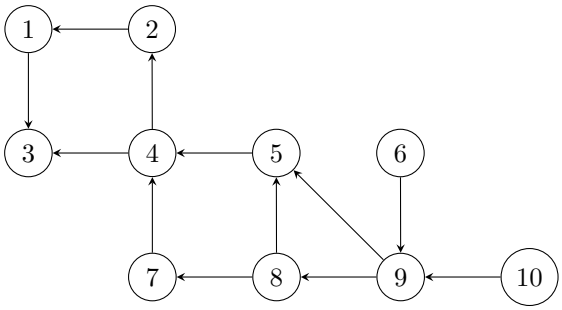
   - *För trea:* `addNode`, `addEdge`: $O(1)$ amorterat, `closestFromNode`: $O(n(v + e \log e))$ eller $O(n(v + e) \log v)$
   - *För fyra eller femma:* `addNode`, `addEdge`: $O(1)$ amorterat, `closestFromNode`: $O(v + e \log e)$ eller $O((v + e) \log v)$

   För betyget fyra eller femma ska du dessutom 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. För anrop till tillåtna datastrukturers operationer kan du hänvisa till standardimplementeringens tidskomplexitet.

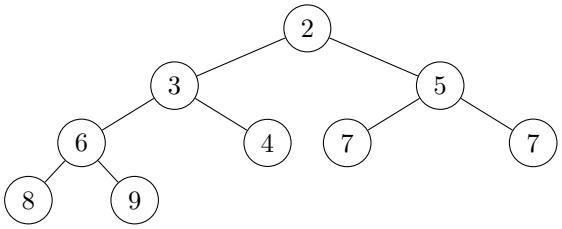
   Halvfärdiga lösningar godkänns knappast; om du inte lyckas konstruera en *korrekt* implementering som uppfyller kravet för ett visst betyg så kan det vara en bra idé att istället försöka konstruera en enklare implementering som uppfyller kravet för ett lägre betyg.

4. Ange en topologisk sortering för följande acykliska, riktade graf:



Svara med en lista av tal där talen motsvarar nodernas etiketter.

5. Följande träd representerar tillståndet i en min-heap med heltal.



Prioriteten för varje tal motsvaras av talet självt.

Visa med ett träd tillståndet i heapen efter

a) insättning av talet 1

b) anrop av `deleteMin` en gång

Operationerna i a) och b) ska *båda* utföras på den *ursprungliga* heapen, d.v.s. b) ska *ej* utföras på heapen som är svaret i uppgift a). Svaret ska bestå av två träd.

För betyget tre räcker det att svara rätt på en av deluppgifterna.

6. Konstruera en datastruktur som representerar en slags kö av icke-negativa heltal. Ur kön ska man kunna plocka minsta elementet med avseende på två olika ordningar, A och B. Ordning A är den vanliga för heltal ($0 < 1 < 2 < ...$). För ordning B är heltal x mindre än/lika med/större än heltal y då resten (modulo) för x dividerat med 100 är mindre än/lika med/större än resten för y dividerat med 100 (t.ex. 200<25<250<175).

   Implementera följande operationer:

   `empty` skapar en tom kö av den ovan angivna typen.

   `insert(`x**`)`** sätter in icke-negativa heltalet x i kön.

   `deleteMinA()/deleteMinB()` plockar bort och returnerar ett av de minsta elementen enligt ordning A/B. Om kön är tom returneras **`null`**. Observera att detta är två olika operationer.

   Du ska också visa hur datastrukturen representeras, d.v.s. ange vilka variabler och hjälpdatastrukturer du använder för att lagra kön.

   Du får använda datastrukturerna dynamisk array, FIFO-kö, binär heap, AVL-träd och skipplista utan att implementera dessa. Metoder som inte tillhör den/de standard-ADT som datastrukturen implementerar får användas, men implementeringen måste då visas och den interna representation av datastrukturen som du utgår ifrån måste beskrivas. Metoden för att ta bort ett visst namngivet element ur binär heap ska antas ta linjär tid (*ej* logaritmisk).

   Implementationen av datastrukturen behöver inte beskrivas med detaljerad kod, men lösningen måste innehålla så mycket detaljer att tidskomplexiteten kan analyseras. För anrop till tillåtna datastrukturers operationer kan du hänvisa till standardimplementeringens tidskomplexitet.

   Låt n vara antalet element i kön. Operationerna ska ha följande tidskomplexitet:

   - *För trea:* `DCPQueue`: $O(1)$, `insert`: $O(\log n)$, `deleteMinA` och `deleteMinB`: $O(n)$
   - *För fyra eller femma:* `DCPQueue`: $O(1)$, `insert`, `deleteMinA` och `deleteMinB`: $O(\log n)$

   Amorterad tidskomplexitet är tillåtet för alla operationerna oavsett betygsnivå. För betyget fyra eller femma ska du dessutom visa att tidskomplexiteterna är de angivna.

   Onödigt krångliga lösningar kan ge lägre betyg. Halvfärdiga lösningar godkänns knappast.