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:

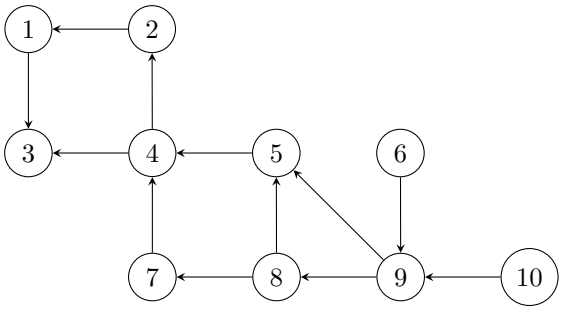
   - *För trea:* `addNode`, `addEdge`: $O(1)$ amorterat, `closestFromNode`: $O(n(v + e \log e))$ eller $O(n(v + e) \log v)$
   - *För fyra eller femma:* `addNode`, `addEdge`: $O(1)$ amorterat, `closestFromNode`: $O(v + e \log e)$ eller $O((v + e) \log v)$

   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.

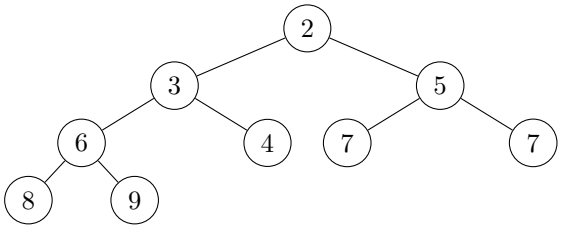
   Halvfärdiga lösningar godkänns knappast; om du inte lyckas konstruera en *korrekt* implementering som uppfyller kravet för ett visst betyg så kan det vara en bra idé att istället försöka konstruera en enklare implementering som uppfyller kravet för ett lägre betyg.

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.

Lösningsförslag till tentamen

Datastrukturer, DAT037 (DAT036), 2017-01-11

1. Loopen upprepas `n` gånger.

   `getAt` på en dynamisk array tar tiden $O(1)$.

   `member` på ett AVL-träd av storlek `n` tar tiden $O(\log \mathtt{n})$.

   `addLast` på en dynamisk array tar tiden $O(1)$ amorterat, `n` anrop av metoden tar tiden $O(\mathtt{n})$.

   Tiden det tar att utföra en iteration av loopen är oberoende av värdet på `i`.

   $$T(\mathtt{n}) = O(\mathtt{n}(1 + \log \mathtt{n}) + \mathtt{n}) = O(\mathtt{n} \log \mathtt{n})$$

2. i)
```
public int height() {
  return nodeHeight(root);
}
private int nodeHeight(Node node) {
  if (node == null) return -1;
  return 1 + Math.max(nodeHeight(node.left),
                      nodeHeight(node.right));
}
```

   Tidskomplexitet: `height` tar konstant tid bortsett från anropet till `nodeHeight`. `nodeHeight` tar i värsta fall konstant tid bortsett från rekursiva anropen. `nodeHeight` anropas en gång för roten och två gånger för varje nod. Antalet noder är n. $T(n) = O(1 + (1 + 2n) \cdot 1) = O(n)$

   ii)
```
public boolean isAVLBalanced() {
  return balancedHeight(root) >= -1;
}
// Returnerar höjden om delträdet är balanserat
// annars -2
private int balancedHeight(Node node) {
  if (node == null) return -1;
  int hl = balancedHeight(node.left);
  if (hl == -2) return -2;
  int hr = balancedHeight(node.right);
  if (hr == -2) return -2;
  if (Math.abs(hl - hr) <= 1)
    return 1 + max(hl, hr);
  else
    return -2;
}
```

   Tidskomplexitet: Samma resonemang som för i).

3. Ett lösning är att lägga alla startnoder i kön från början.

   Använd en grannlista för att lagra grafen. Använd avbildning implementerad med en Hashtabell och lista implementerad med dynamisk array. Definiera en hjälpklass

```
private class Edge {
  int weight;
  int adjNode;
}
```

   och använd föjande fält:

```
HashMap<Integer, ArrayList<Edge>> a;
```

   Implementering av `addNode`:

```
addNode(i)
  a.put(i, new ArrayList<Edge>)
```

   Skapa en ny dynamisk array tar konstant tid och lägga in element i hashtabell tar amorterad konstant tid. Alltså tidskomplexitet $O(1)$ amorterat.

   Implementering av `addEdge`:

```
addEdge(i, j, w)
  if (!a.containsKey(i)) return
  a.get(i).addLast(ny Edge med adjNode = j, weight = w)
```

   Uppslag i hashtabell och skapa nytt `Edge`-objekt tar konstant tid. Lägga till element i slutet av dynamisk array tar konstant tid amorterat. Tidskomplexitet: $O(1)$ amorterat.

   Implementering av `closestFromNode`: Dijkstras algoritm kan användas. Man kan utföra Dijkstras en gång för varje nod i `f` och välja den med kortast väg. Man kan också utföra Dijkstras en gång genom att lägga in alla noder i `f` i prioritetskön i början. När man hittat slutnoden söker man tillbaka för att hitta startnoden som är närmst. Det andra sättet leder till komplexiteten som efterfrågas för betyg fyra eller fem. Förslag på implementering enligt andra sättet:

```
closestFromNode(f, t)
  let d = new HashMap<Integer, Integer>  // bästa kända avstånd till nod
      v = new HashSet<Integer>  // nod besökt
      p = new HashMap<Integer, Integer>  // förågende nod i kortaste väg
      q = new PriorityQueue<Integer>
           with priority = d.get(i) for each element i
  for each int i in f {
    d.put(i, 0)
    q.offer(i)
  }
```

```
 while q not empty {
  i = q.poll()
  if (i == t) findStartNode (se nedan)
  v.add(i)
  for each Edge e in a.get(i) {
    j = e.adjNode
    if (!v.contains(j)) {
      newd = d.get(i) + e.weight
      if (!d.containsKey(j) || newd < d.get(j)) {
        q.remove(j)
        d.put(j, newd)
        p.put(j, i)
        q.offer(j)
      }
    }
  }
 }
 // t kan ej nås
 return null

findStartNode:
 // t har nåtts, hitta startnod
 i = t
 while (p.containsKey(i)) {
   i = p.get(i)
 }
 return i
```

Initiering av `d`, `v`, `p` och `q` tar konstant tid.

Första loopen upprepas n gånger. Insättning i `d` tar $O(1)$ amorterat. Insättningarna i `q` går på konstant tid amorterat eftersom alla element har samma prioritet. Noderna i `f` är distinkta, så $n \leq v$. Första loopen: $O(n(1+1)) = O(n) = O(v)$

Andra loopen upprepas i värsta fall v gånger eftersom en nod finns med max 1 gång i kön och en besökt nod inte läggs i kön igen. Kön innehåller max v element. Uttag ur `q` tar $O(\log v)$. Insättning i `v` tar $O(1)$ amorterat. Andra loop bortsett från dess inre loop: $O(v \log v)$

Inre loopen i andra loopen upprepas totalt max e gånger. Uppslag i `d` tar $O(1)$. Borttagning ur och insättning i `q` tar $O(\log v)$ (amorterat för insättning). Insättning i `d` och `p` tar $O(1)$ amorterat. Inre loopen totalt: $O(e \log v)$

While-loopen som letar upp startnoden upprepas max v gånger och varje iteration tar konstant tid. $O(v)$

Totalt för operationen: $O(v + v \log v + e \log v + v) = O((v + e) \log v)$

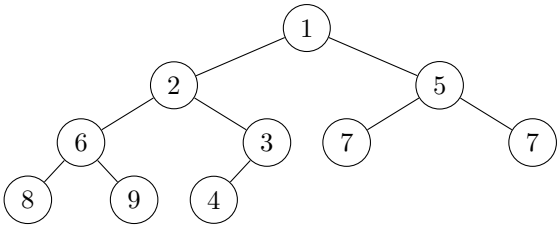
Om man väljer en lösning där element inte uppdateras/tas bort i kön så kan man komma fram till $O(v + e \log e)$.

4. Detta är en topologisk sortering för grafen:

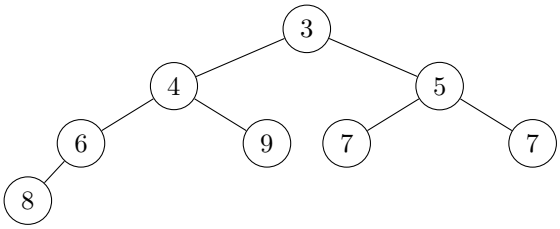
$$6, 10, 9, 8, 5, 7, 4, 2, 1, 3$$

6 och 10 samt 5 och 7 kan byta plats.

5. a) insättning av talet 1



b) anrop av `deleteMin` en gång



6. Lösningsförslag som ger logaritmisk tidskomplexitet för `deleteMinA` och `deleteMinB`. Använd två mängder, `a` och `b`, implementerade med AVL-träd. Låt `a` vara sorterad enligt ordning A och `b` enligt ordning B. Antag att representationen av AVL-träden ser ut så här:

```
public class AVLTree {
    private class Node {
        A    contents;  // Innehåll.
        Node left;      // Vänstra barnet.
        Node right;     // Högra barnet.
        ...  // höjd/balans
    }
    private Node root;  // roten
    ...
}
```

```
empty()
  a = nytt tomt AVL-träd sorterat enligt x.compareTo(y)
  b = nytt tomt AVL-träd sorterat enligt
        (x % 100).compareTo(y % 100)
```

Skapa ett tomt träd tar konstant tid, så $O(1)$.

```
insert(x)
  a.add(x)
  b.add(x)
```

Insättning i AVL-träd tar logaritmisk tid och varje träd innehåller n element, så $O(\log n)$.

```
deleteMinA()
  if (a.isEmpty()) return null;
  x = a.findMin()
  a.remove(x);
  b.remove(x);
  return x;

findMin()
  // trädet ej tomt
  n = root
  while (n.left != null) {
    n = n.left;
  }
  return n.contents;
```

`findMin` är operation hos våra AVL-träd. Exekveringen tar konstant tid förutom loopen. En iteration i loopen tar konstant tid. Antal iterationer är begränsad av trädets höjd och den är begränsad av $\log n$. Tidskomplexiteten för `findMin` är $O(\log n)$.

Tidskomplexiteten för `deleteMinA`: `isEmpty` för träd tar konstant tid. Borttagning ur AVL-träd tar logaritmisk tid. Alltså $O(\log n)$

`deleteMinB` implementeras på samma sätt med enda skillnaden att `b.findMin()` anropas istället.