Tentamen
Datastrukturer, DAT037

- Datum och tid för tentamen: 2018-01-10, 14:00–18:00.
- Ansvarig: Fredrik Lindblad. Nås på tel nr. 031-772 2038. Besöker tentamenssalarna ca 15:00 och ca 16:30.
- Godkända hjälpmedel: Ett A4-blad med handskrivna anteckningar (fram- och baksida).
- 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 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++) {
    t.add(s.pop());
}
for (int i = 0; i < n; i++) {
    l.addFirst(t.deleteMax());
}
```

   Använd kursens uniforma kostnadsmodell och gör följande antaganden:

   - Att `int` kan representera alla heltal och att `n` är ett positivt heltal.
   - Att `s` är en stack av heltal med från början `n` distinkta element och att stacken är implementerad med en dynamisk array.
   - Att `t` är ett från början tomt AVL-träd med heltalselement.
   - Att `l` är en från början tom länkad lista med heltalselement.

   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. Följande array av heltal representerar en binär heap. Roten är det första elementet från vänster.

3	5	8	10	8	13	16	12	13	15	20	20	14

   Hur ser arrayen ut efter ett anrop av `delete-min`/`poll`?

3. Konstruera en datastruktur som representerar en mängd av heltal. Antag att en heltalsmängd av storlek n innehåller elementen $x_1, x_2, \ldots, x_n$ och $x_1 < x_2 < \cdots < x_n$. I denna uppgift säger vi att mängden då har de $n-1$ *intervallen* $[x_1, x_2], [x_2, x_3], \ldots, [x_{n-1}, x_n]$. Längden på ett intervall $[x_k, x_{k+1}]$ är $x_{k+1} - x_k$. För ett intervall $[x_k, x_{k+1}]$ kallar vi x_k för den nedre gränsen.

   Datastrukturen ska ha följande operationer:

   `empty()` Skapar en tom mängd.

   `add(`x`)` Lägger till heltalet x till mängden. Om x redan finns i mängden förändras den inte.

   `higher(`x`)` Returnerar det minsta tal i mängden som är större än x. Om alla tal i mängden är mindre än x returneras `null`.

   `longestInterval()` Returnerar den nedre gränsen för det längsta intervallet som mängden har. Om mängden har flera intervall som är längst så returneras nedre gränsen för det intervall som ligger längst till vänster på tallinjen, vilket är det med minst nedre (och övre) gräns. Om mängden inte har något intervall returneras `null`.

   Följande pseudokod exemplifierar hur datastrukturen ska fungera:

```
s = empty()
s.add(3)
s.longestInterval()  --> null
s.add(10)
s.longestInterval()  --> 3
s.add(6)
s.add(14)
s.higher(12)         --> 14
s.longestInterval()  --> 6
s.add(8)
s.longestInterval()  --> 10
```

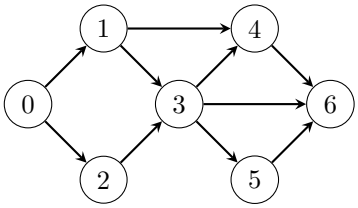
   Låt n vara antalet heltal i mängden. Operationerna ska ha följande tidskomplexitet:

   - *För trea:* `empty`: $O(1)$, `add`, `higher`, `longestInterval`: $O(n)$
   - *För fyra eller femma:* `empty`: $O(1)$,
     `add`, `higher`, `longestInterval`: $O(\log n)$

   För samtliga operationer kan angivet mått gälla amorterat eller i genomsnitt. Oavsett betyg ska du genom analys av koden visa att komplexitetskraven är uppfyllda.

   Du kan använda eller utgå från standarddatastrukturer och -algoritmer utan att beskriva dem i detalj. Implementationen av datastrukturen behöver inte beskrivas med detaljerad kod, men lösningen måste innehålla så mycket detaljer att tidskomplexiteten kan analyseras.

4. Implementera en metod som, givet en riktad, acyklisk graf och en start- och slutnod, beräknar antalet distinkta vägar mellan start- och slutnoden i grafen. Till exempel ska resultatet för grafen nedan med 0 som startnod och 6 som slutnod vara 7.



Nodetiketterna är heltal mellan 0 och $v-1$, där v är antalet noder. Grafen representeras med grannlistor enligt följande klassfragment:

```
class Graph {
    private int[][] adj;
    ...
    public int countPaths(int start, int dest) { ... }
}
```

För grafen ovan så är `adj` en array av längd sju och `adj[2]`, för att ta ett exempel, är en array av längd ett som innehåller talet 3.

Ditt jobb är alltså att implementera metoden `countPaths`.

För *fyra eller femma* ska metoden har tidskomplexiteten $O(v + e)$, där v är antalet noder och e antalet kanter i grafen. För dessa betyg måste du också visa att så är fallet.

Endast detaljerad kod godkänns. Med undantag av grafalgoritmer kan du använda standarddatastrukturer och -algoritmer utan att förklara hur dessa fungerar.

5. Uppgiften är att implementera metoder i följande klass:

```
class SinglyLinkedList<A> {
 private class Node {
     public A contents;  // Nodens innehåll.
     public Node next;   // Nästa nod; null för
                         // sista noden.
     public Node() {}
 }
 private Node first = null;
      // Första noden; null om listan är tom.

 public SinglyLinkedList() {}

 public boolean contains(A e) {
  ...
 }

 public void reverse() {
  ...
 }

 ...
}
```

Klassen `SinglyLinkedList` är generisk och representerar en lista med en enkellänkad lista utan vaktposter.

Implementera metoden `contains(e)`. Den ska returnera `true` om ett element lika med `e` finns i listan och annars `false`. Metoden ska inte förändra listan. Metoden ska ha komplexiteten $O(n)$, där n är antalet element i listan, och du ska visa att så är fallet.

För fyra eller femma ska du också implementera metoden `reverse()`. Denna metod ska förändra listan så att elementen hamnar i omvänd ordning. Om till exempel den listan som instansen representerar är $[1, 3, 2, 5]$ så ska den efter ett anrop av `reverse()` vara $[5, 2, 3, 1]$. Metoden ska ha komplexiteten $O(n)$, där n är antalet element i listan, och du ska 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.

6. Konstruera en datastruktur för att representera en mängd av strängar innehållande decimalsiffror (tecknen `'0'`, `'1'`, …, `'9'`). Datastrukturen ska följande operationer:

   `empty()` Skapar en tom mängd.

   `add(`*s*`)` Lägger till decimalsträngen *s* till mängden. Om strängen redan finns i mängden förändras den inte.

   `contains(`*s*`)` Returnerar `true` om decimalsträngen *s* finns i mängden, annars `false`.

   `restrictToPrefix(`*s*`)` Tar bort alla strängar i mängden som *inte* börjar med decimalsträngen *s*.

   Notera att för alla operationer som har *s* som argument så kan detta antas vara en sträng vars tecken alla är någon av siffrorna `'0'` till `'9'`.

   Följande kod exemplifierar hur operationerna ska fungera:

```
s = empty()
s.add("25")
s.add("253")
s.add("2539")
s.add("257")
s.add("332")
s.restrictToPrefix("253")
s.contains("25")                -> false
s.contains("253")               -> true
s.contains("2539")              -> true
s.contains("257")               -> false
s.contains("332")               -> false
```

   Operationerna måste ha följande tidskomplexiteter (där n är antalet element i mängden, och ℓ är längden på strängargumentet som benämns *s* ovan):

   - *För trea:* `empty`: $O(1)$,
     `add`, `contains`, `restrictToPrefix`: $O(\ell n)$.
   - *För fyra eller femma:* `empty`: $O(1)$,
     `add`, `contains`, `restrictToPrefix`: $O(\ell)$.

   Oavsett betyg ska du genom analys av koden visa att komplexitetskraven är uppfyllda.

   Du kan använda eller utgå från standarddatastrukturer och -algoritmer utan att förklara hur de fungerar. Implementationen av datastrukturen behöver inte beskrivas med detaljerad kod, men lösningen måste innehålla så mycket detaljer att tidskomplexiteten kan analyseras.