Tentamen
Datastrukturer, DAT037 (DAT036)

- Datum, tid och plats för tentamen: 2017-08-17, 8:30–12:30, M.
- Ansvarig: Fredrik Lindblad. Nås på tel nr. 031-772 2038. Besöker tentamenssalarna ca 9:30 och ca 11:00.
- Godkända hjälpmedel: Ett dubbelsidigt 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 värstafalls-tidskomplexitet, uttryckt i `n`:

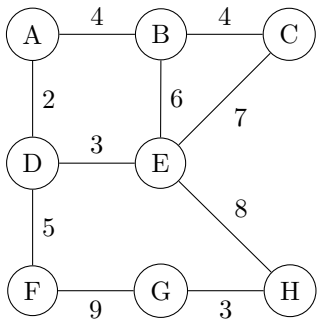
```
for (int i = 0; i < n; i++) {
    if (s.member(l.getAt(i))) {
        t.add(l.getAt(i));
    }
}
```

   Använd kursens uniforma kostnadsmodell och gör följande antaganden:

   - Att `int` kan representera alla heltal och att `n` är ett icke-negativt heltal.
   - Att `s` är en mängd av heltal som är implementerad med en hashtabell och som innehåller högst `n` element.
   - Att hashfunktionen som används i `s` är perfekt och att beräkning av hashvärden tar konstant tid.
   - Att `l` är en lista av heltal som är implementerad med en dynamisk array och som innehåller `n` element.
   - Att `t` är en mängd av heltal implementerad med ett AVL-träd och som från början är tom.
   - Att `member(x)` avgör om `x` ingår i mängden, `getAt(i)` returnerar elementet med index `i` i listan och att `add(x)` lägger in `x` i mängden.

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

2. Beräkna ett minimalt uppspännande träd för följande graf genom att utföra Prims algoritm med A som startnod.



   Redovisa genom att ange vilka kanter som ingår i det uppspännande trädet. Lista kanterna i den ordning som de läggs till när man utför Prims algoritm. Hänvisa till varje kant med hjälp av etiketterna på de två noder den sammanbinder, t.ex. DF för kanten mellan noderna D och F.

3. Java-klassen `Tree` representerar ett binärt obalanserat sökträd där noderna innehåller heltal:

```
class Tree {
  private class Node {
    int data;
    Node parent, left, right;
  }

  private Node root;

  ...

  public void removeAllLessThan(int x) { ... }
}
```

Trädets noder representeras av den lokala klassen `Node`. Variabeln `root` refererar till rotnoden. För tomma träd är denna `null`. Varje nod innehåller referenser till föräldranoden (`parent`), samt vänster- och högerbarn (`left` och `right`). Avsaknad av vänster- eller högerbarn representeras av `null`. Variabeln `parent` är `null` i rotnoden.

Implementera metoden `removeAllLessThan`. Metoden ska, givet ett heltal `x`, ta bort alla noder i aktuellt träd vars innehåll är ett tal mindre än `x`. Du kan utgå från att noderna är ordnade som de ska vara i ett sökträd. Du behöver inte upprätthålla någon särskild balans i trädet. De referenser till barn och föräldrar som påverkas av förändringen måste förstås uppdateras på ett korrekt sätt.

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

4. Konstruera en datastruktur som representerar en tvåställig relation mellan heltal.

   Datastrukturen ska ha följande operationer:

   `empty()` som skapar ett objekt som motsvarar en tom relation, d.v.s. en relation där inga par av heltal är relaterade.

   `addRelation(`x`, `y`)` som ändrar relationen så att x är relaterad till y. Om x från början är relaterad till y så händer ingenting.

   `removeRelation(`x`, `y`)` som ändrar relationen så att x inte är relaterad till y. Om x från början inte är relaterad till y så händer ingenting.

   `isRelated(`x`, `y`)` som returnerar `true` om x är relaterad till y och `false` annars.

   Nedanstående kod ger exempel hur datastrukturen ska fungera. Värdena som de booleska variablerna `b1`, `b2`, `b3`, `b4` och `b5` antar anges i slutet på respektive rad.

```
r = empty();
r.addRelation(2, 5);
r.addRelation(4, 3);
r.removeRelation(2, 5);
b1 = isRelated(2, 5);  // false
b2 = isRelated(4, 3);  // true
b3 = isRelated(3, 4);  // false
b4 = isRelated(4, 2);  // false
b5 = isRelated(7, 7);  // false
```

   Notera att om x är relaterad till y så är inte nödvändigtvis y relaterad till x.

   Låt n vara antalet heltalspar som är relaterade i relationen. Utrymmeskomplexiteten för datastrukturen ska tillhöra $O(n)$. Tidskomplexiteten för samtliga operationer ska tillhöra följande komplexitetsklass:

   - *För trea:* $O(n)$
   - *För fyra eller femma:* $O(1)$ amorterat

   Oavsett betyg ska du 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.

   Konstruera om möjligt inte datastrukturen från grunden, bygg den hellre m.h.a. standarddatastrukturer. För anrop till standarddatastrukturers operationer kan du hänvisa till standardimplementeringens tidskomplexitet.

5. Följande Java-klass representerar en riktad graf där noderna identifieras med heltal:

```
public class Graph {
    private Map<Integer, List<Integer>> alist;

    public Graph() { ... }
    public void addNode(int x) { ... }
    public void addEdge(int x, int y) { ... }
    public boolean equals(Graph g) { ... }
}
```

Grafen representeras med grannlistor i instansvariabeln `alist`. Klassen representerar ej multi-grafer d.v.s. för varje ordnat par (x, y) av noder finns högst en kant från x till y. Observera att kanten från x till y inte är densamma som kanten från y till x.

Din uppgift är att implementera konstrueraren och de tre operationerna vars signaturer framgår ovan.

`Graph` ska skapa ett objekt som representerar en tom graf.

`addNode(`x`)` ska lägga till en ny nod som identifieras med talet x. Du kan utgå från att en nod identifierad av x inte redan finns i grafen.

`addEdge(`x`, `y`)` ska lägga till en kant från noden som identifieras med talet x till noden som identifieras med talet y. Du kan utgå från att noderna x och y finns i grafen och att det inte redan finns en kant från x till y.

`equals(`g`)` ska returnera `true` om grafen g representerar samma graf som aktuellt objekt (`this`). Annars ska `false` returneras. Graferna ska ej förändras. Två grafer anses lika om de innehåller samma mängd noder och kanter. Två noder anses lika om de har samma heltals-identifierare i respektive graf. Två kanter anses lika om från-noderna har samma identifierare och till-noderna har samma identifierare i respektive graf.

Inga andra instansvariabler än `alist` får användas.

Låt V vara antal noder och E antalet kanter i aktuellt objekt. Tidskomplexiteten för de olika operationerna ska tillhöra följande klasser:

- *För trea:* `addNode`: $O(\log V)$ *ej* amorterat, `addEdge`: $O(\log V + E)$ *ej* amorterat, `equals`: $O(V + E)$.
- *För fyra eller femma:* `addNode` och `addEdge`: $O(\log V)$ *ej* amorterat, `equals`: $O(V + E)$.

Oavsett betyg ska du visa att så är fallet. (forts. nästa sida)

Implementationen av metoderna behöver inte beskrivas med detaljerad kod, men lösningen måste innehålla så mycket detaljer att tidskomplexiteten kan analyseras.

Standarddatastrukturer får användas utan att implementeras, dock ej operationer för grafer. För anrop till standarddatastrukturers operationer kan du hänvisa till standardimplementeringens tidskomplexitet.

6. Implementera en datastruktur som representerar en mängd av heltal.

   Datastrukturen ska ha följande operationer:

   `empty()` som skapar en tom mängd.

   `add(`x`)` som lägger till talet x till aktuell mängd. Om x redan finns i mängden så förändras den inte.

   `remove(`x`)` som tar bort talet x från mängden. Om x inte finns i mängden så förändras den inte.

   `contains(`x`)` som returnerar `true` om mängden innehåller talet x och `false` annars.

   `successorOf(`x`)` som returnerar det minsta heltal som ingår i mängden och är större än x. Om det inte finns något sådant tal i mängden ska operationen returnera `null`. Även om x inte ingår i mängden ska alltså ett tal returneras (såvida det finns ett tal i mängden som är större än x).

   Låt n beteckna antalet heltal i mängden. Tidskomplexiteterna för de olika operationerna ska tillhöra följande komplexitetsklasser:

   - *För trea:* `empty` : $O(1)$, `add` och `remove`: $O(n)$, `contains`: $O(\log n)$, `successorOf`: $O(\log n)$
   - *För fyra eller femma:* `empty` : $O(1)$, `add` och `remove`: $O(\log n)$ amorterat, `contains`: $O(1)$, `successorOf`: $O(\log n)$

   Oavsett betyg ska du 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. Konstruera om möjligt inte datastrukturen från grunden, bygg den hellre m.h.a. standarddatastrukturer. För anrop till standarddatastrukturers operationer kan du hänvisa till standardimplementeringens tidskomplexitet.

Lösningsförslag till tentamen

Datastrukturer, DAT037 (DAT036), 2017-08-17

1. Loopen upprepas `n` gånger, för `i` från 0 till $\texttt{n} - 1$. För varje upprepning utförs i värsta fall dessa operationer:

   - 2 stycken `l.getAt`, uppslag i dynamisk array. Det tar $O(1)$.
   - `s.member`, sökning i perfekt hashtabell. Det tar $O(1)$.
   - `t.add`, insättning i AVL-träd som för varje upprepning maximalt har storleken `i`. Det tar $O(\log \texttt{i})$.

   Övrigt i och utanför loopen tar konstant tid.

   Värstafallskomplexiteten blir

   $$T(\texttt{n}) = O(1) + \sum_{i=1}^{\texttt{n}-1}(\log i + 3O(1)) = \sum_{i=1}^{\texttt{n}-1} \log i = O(\texttt{n} \log \texttt{n})$$

2. AD, DE, AB, BC, DF, EH, HG

3. ```
   private void removeAllLessThan(Node n, int x) {
     if (n == null) return;
     if (n.data >= x) {
       removeAllLessThan(n.left, x);
     } else {
       if (n.parent == null) {
         root = n.right;
       } else {
         n.parent.left = n.right;
       }
       if (n.right != null) {
         n.right.parent = n.parent;
         removeAllLessThan(n.right, x);
       }
     }
   }

   public void removeAllLessThan(int x) {
     removeAllLessThan(root, x);
   }
   ```
   ```

4. Antar perfekt hashfunktion för hashtabellerna.

```
class Relation {
  Map<Integer, Set<Integer>> r;

  public Relation() {
    r = new HashMap<>();  // O(1)
  }  // = O(1)
  public void addRelation(int t, int u) {
    if (!r.containsKey(t)) {  // O(1)
      r.put(t, new HashSet<>());  // O(1) + O(1) amorterat
    }
    r.get(t).add(u);  // O(1) + O(1) amorterat
  } // = O(1) amorterat

  public void removeRelation(int t, int u) {
    if (r.containsKey(t)) {  // O(1)
      r.get(t).remove(u);  // O(1) + O(1) amorterat
    }
  } // = O(1) amorterat

  public boolean isRelated(int t, int u) {
    if (!r.containsKey(t)) return false;  // O(1)
    return r.get(t).contains(u);  // O(1) + O(1)
  }  // = O(1)
}
```

5. Antar perfekt hashfunktion för hashtabellen som används. Antar också att iterator för balanserade sökträd som används traverserar trädet inorder, vilket betyder att om två träd innehåller samma nycklar så kommer de besökas i samma ordning.

```
public class Graph {
  private Map<Integer, List<Integer>> alist;
  public Graph() {
    alist = new TreeMap<>();
  }
  public void addNode(int x) {
    alist.put(x, new LinkedList<>());  // O(1) + O(log V)
  }
  public void addEdge(int x, int y) {
    alist.get(x).add(y);  // O(log V) + O(1)
  }
  public boolean equals(Graph g) {
    if (alist.size() != g.alist.size()) return false;  // O(1)
    Iterator<Entry<Integer,List<Integer>>> gait =
        g.alist.entrySet().iterator();  // O(1)
    for (Entry<Integer,List<Integer>> kv : alist.entrySet()) {  // V times
        // implicit next() för for-loopens iterator: O(log V), totalt O(V)
      Entry<Integer,List<Integer>> gkv = gait.next();  // O(log V), totalt O(V)
      if (kv.getKey() != gkv.getKey()) return false;  // O(1)
      List<Integer> a = kv.getValue();  // O(1)
      List<Integer> ga = gkv.getValue();  // O(1)
      if (a.size() != ga.size()) return false; // O(1)
      Set<Integer> gas = new HashSet<>();  // O(1)
      for (int y : ga) {  // max E gånger totalt
          // implicit next() för for-loopens iterator: O(1)
        gas.add(y);  // O(1) amorterat, O(E) totalt
      }
      for (int y : a) {  // max E gånger totalt
          // implicit next() för for-loopens iterator: O(1)
        if (!gas.contains(y)) return false; // O(1)
      }
    }
    return true;
  }
}
```

Lägger man ihop komplexiteten för de olika delarna får man:

- `addNode`: $O(\log V)$
- `addEdge`: $O(\log V)$
- `equals`: $O(V + E)$

6. Antar perfekt hashfunktion.

```
empty() = låt hs vara en mängd implementerad med hashtabell (O(1))
          låt ts vara en mängd implementerad med balanserat sökträd (O(1))

add(x) = sätt in x i hs (O(1)) amorterat
         sätt in x i ts (O(log n))

remove(x) = ta bort x från hs (O(1)) amorterat
            ta bort x från ts (O(log n))

contains(x) = slå upp x i hs (O(1))
```

`successorOf` kan implementeras som en variant på vanligt sökning i sökträd. Antag (förenklat) att noderna i trädet representeras klassen `Node` och att det finns en instansvariabel som pekar på rot-noden.

```
class Node {
  int data;
  Node left, right;
}
Node root;
```

Man kan då implementera metoden så här:

```
public Integer successorOf(int x) {
  return successorOf(x, root);
}

private Integer successorOf(int x, Node n) {
  if (n == null) {
    return null;
  }
  if (x < n.data) {
    Integer res = successorOf(x, n.left);
    if (res == null) res = n.data;
    return res;
  } else {
    return successorOf(x, n.right);
  }
}
```

Liksom för vanlig sökning i sökträd så traverseras en väg. Varje besökt nod tar $O(1)$. Antalet noder i vägen är begränsat av $O(\log n)$ eftersom trädet är balanserat.