Tentamen
Datastrukturer, DAT037 (DAT036)

- Datum och tid för tentamen: 2017-04-11, 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.
- 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++) {
    l.addLast(q.deleteMin());
}
```

   Använd kursens uniforma kostnadsmodell, och gör följande antaganden:

   - Att `int` kan representera alla heltal och att `n` är ett icke-negativt heltal.
   - Att `q` är en prioritetskö som är implementerad med en binär heap och från början innehåller `n` heltal.
   - Att `l` är en lista som är implementerad med en dynamisk array och från början är tom.
   - Att `addLast` lägger till ett element sist i listan.
   - Att jämförelserna av heltal som utförs i heapen 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 oriktade grafer som är representerade med följande klass:

```
public class Graph {
    private List<Set<Integer>> adj = new ArrayList<>();

    public int addNode() { ... }
    public void addEdge(int i, int j) { ... }
    public boolean isConnected() { ... }
}
```

Grafen representeras med grannlistor och noder(hörn) identifieras med talen 0 till $n-1$, där n är antalet noder i grafen. Längden på listan `adj` är n. För varje nod i i grafen så finns dess grannoder i mängden `adj.get(`i`)`. Alla element i mängderna är tal mellan 0 och $n-1$. För varje nod i och j gäller att om j finns i mängden `adj.get(`i`)` så finns också i i mängden `adj.get(`j`)`.

`addNode()` ska lägga till en ny nod som identifieras med talet n, där n är antalet noder i grafen före anropet. Metoden ska returnera n.

`addEdge(i, j)` ska lägga till en (oriktad) kant mellan nod `i` och nod `j`. Metoden kan anta att `i` och `j` motsvarar noder som finns i grafen.

`isConnected()` ska returnera `true` om grafen som objektet representerar är sammanhängande och `false` annars.

- *För trea:* Metoderna `addNode` och `addEdge` ska implementeras.
- *För fyra eller femma:* Även `isConnected` ska implementeras. Dess tidskomplexitet ska vara $O(V)$, där V är antalet noder, och du ska visa att så är fallet.

Implementationerna 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.

3. Konstruera en klass som representerar en mängd av heltal. Klassen ska ha följande operationer:

   En konstruerare som skapar ett objekt motsvarande en tom mängd.

   `add(`x`)` som lägger till ett heltal x till mängden om det inte redan finns. Om talet lades till ska operationen returnera `true`, annars `false`.

   `deleteMedian()` som returnerar och tar bort medianen i mängden. Precondition: mängden är icke-tom. Antag att elementen i mängden är $x_1, x_2, \ldots, x_n$ där $x_i < x_{i+1}$ för alla $1 \leq i \leq n-1$. Mängden innehåller alltså n element och x_1 är den minsta elementet, x_2 det näst minsta o.s.v. Då n är udda är medianen $x_{n/2+1}$. När n är jämnt ska $x_{n/2}$ räknas som medianen.

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

   - *För trea:* `add`: $O(n)$, `deleteMedian`: $O(n)$
   - *För fyra eller femma:* `add`: $O(\log n)$, `deleteMedian`: $O(\log n)$

   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.

   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.

   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. Antag att `sort` är en rekursiv funktion som sorterar en lista och är implementerad med Quicksort. Visa hur exekveringen går till när `sort` anropas för att sortera listan $\{8, 5, 7, 2, 1, 6, 4, 9\}$. Valet av pivot-element är fritt. Visa exekveringen genom att ange alla rekursiva anrop som sker. Varje anrop ska beskrivas genom att ange de tal som ska sorteras av det rekursiva anropet. Om t.ex. ett rekursivt anrop ska sortera talen 5 och 2, kan detta motsvaras av "{5, 2}" i ditt svar.

5. Konstruera en generisk klass som implementerar följande generiska gränssnitt för listor.

```
interface List<E> {
    void addLast(E elt);
    E get(int index);
}
```

Klassen ska implementeras med en enkellänkad lista, med eller utan vaktposter. Den ska ha en konstruktor utan argument som skapar en tom lista. Metoden `addLast` ska lägga till ett element i slutet av listan. Den ska ha tidskomplexiteten $O(1)$. Metoden `get` ska returnera elementet på plats `index` i listan. Första elementet har index 0. Metoden kan anta att `index` är giltigt, d.v.s. mellan 0 och $n - 1$, där n är antal element i listan.

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.

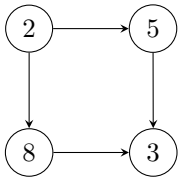
6. Implementera metoden `topol` i följande klass som representerar riktade grafer:

```
public class Graph {
    private Map<Integer, Set<Integer>> adj;

    ...

    public List<Integer> topol() { ... }
}
```

Grafen representeras av grannlistor och varje nod identifieras av ett unikt heltal. För varje nod i så utgör `adj.get(`i`)` grannlistan för i. Varje tal j i `adj.get(`i`)` motsvarar en kant mellan nod i och nod j.

Metoden `topol` ska returnera en lista med noder(heltal) som motsvarar den topologiska sortering som uppfyller beskrivningen nedan. Om ingen topologisk sortering finns för grafen så ska metoden returnera `null`.

Det kan finnas flera topologiska sorteringar för en graf. Exempelvis är både $\{2, 5, 8, 3\}$ och $\{2, 8, 5, 3\}$ topologiska sorteringar för följande graf:



Algoritmen du implementerar ska beräkna den topologiska sortering som resulterar när resultatlistan byggs upp från början till slut och varje gång en nod läggs till listan så väljs den (bland de möjliga noderna) som identifieras av det minsta talet. Metoden ska alltså för grafen ovan returnera listan $\{2, 5, 8, 3\}$ och *inte* $\{2, 8, 5, 3\}$.

Låt V vara antalet noder och E antalet kanter i grafen. Metoden ska ha tidskomplexiteten $O(E + V \log V)$. Visa att detta är fallet.

Implementationen av metoden 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.

Lösningsförslag till tentamen

Datastrukturer, DAT037 (DAT036), 2017-04-11

1. Loopen upprepas `n` gånger, för i från 0 till $\texttt{n} - 1$. Komplexiteten för `deleteMin` för en binär heap är $O(\log j)$ där j är heapens storlek, här $\texttt{n} - i$. Komplexiteten för `addLast` för en dynamisk array är $O(1)$ amorterat. Komplexiteten för n anrop till `addLast` är $O(n)$. Övrigt i loopen tar konstant tid.

$$T(\texttt{n}) = \sum_{i=0}^{\texttt{n}-1} \log(\texttt{n}-i) + O(\texttt{n}) = \sum_{i=1}^{\texttt{n}} \log i + O(\texttt{n}) = O(\texttt{n} \log \texttt{n} + \texttt{n}) = O(\texttt{n} \log \texttt{n})$$

2. Står felaktigt i tentauppgiften att komplexiteten ska vara $O(V)$. Det ska vara $O(V + E)$, där E är antalet kanter.

```
public int addNode() {
        int i = adj.size();
        adj.add(new TreeSet<>());
        return i;
}

public void addEdge(int n1, int n2) {
        adj.get(n1).add(n2);
        adj.get(n2).add(n1);
}

public boolean isConnected() {
        if (adj.size() == 0) return true;
        Set<Integer> visited = new HashSet<>();
        traverse(0, visited);
        return visited.size() == adj.size();
}

private void traverse(int n, Set<Integer> visited) {
        if (visited.contains(n)) return;
        visited.add(n);
        for (Integer adjn : adj.get(n)) {
                traverse(adjn, visited);
        }
}
```

Komplexiteten för `isConnected`: Allt annat än anropet till `traverse` är $O(1)$. Det bortsett från första raden i `traverse` exekveras max V gånger.

Varje exekvering av detta, exklusive loop-upprepningarna, tar $O(1)$ (förutsatt att hashtabellen fungerar väl). Eftersom loopen bara exekveras max 1 gång per nod så upprepas loopen och första raden i `traverse` totalt sett (för alla anrop exekveringar av `traverse`) maximalt $2E$ gånger. Varje exekvering av dessa satser tar $O(1)$ eftersom uppslag i hashtabellen och att stega fram iteratorn för en `TreeSet` tar konstant tid. Komplexiteten är alltså $O(V + E)$.

3.
```
public class MSet {
        TreeSet<Integer> a, b;
        // a innehåller den mindre halvan av elementen,
        //  dvs elementen mindre än medianen.
        // b innehåller den större halvan.
        // Om antalet element är udda så innehåller a
        // ett element fler än b.
        // Medianen är alltså hela tiden största
        // elementet i a.

        public MSet() {
                a = new TreeSet<>();
                b = new TreeSet<>();
        }

        public boolean add(int x) {
                if (a.contains(x) || b.contains(x)) return false;
                if (a.size() > b.size()) {
                        int y = a.last();
                        if (x < y) {
                                a.remove(y);
                                a.add(x);
                                b.add(y);
                        } else {
                                b.add(x);
                        }
                } else {  // a.size() == b.size()
                        if (b.isEmpty()) {
                                a.add(x);
                        } else {
                                int y = b.first();
                                if (x > y) {
                                        b.remove(y);
                                        b.add(x);
                                        a.add(y);
                                } else {
                                        a.add(x);
                                }
```

```
                }
        }
        return true;
}

public int deleteMedian() {
        int x = a.last();
        a.remove(x);
        if (a.size() < b.size()) {
                int y = b.first();
                b.remove(y);
                a.add(y);
        }
        return x;
}
}
```

Komplexitet för `add`: `TreeSet`-operationerna `contains`, `remove`, `add` är alla $O(\log n)$. Allt annat i metoden tar konstant tid. Alltså: $O(\log n)$. Komplexitet för `deleteMedian`: `TreeSet`-operationerna `last`, `first`, `remove`, `add` är alla $O(\log n)$. Allt annat i metoden tar konstant tid. Alltså: $O(\log n)$.

4. Pivot-element: första elementet i delarrayen.

```
{8, 5, 7, 2, 1, 6, 4, 9}
 {4, 5, 7, 2, 1, 6}
  {2, 1}
   {1}
   {}
  {7, 5, 6}
   {6, 5}
    {5}
    {}
   {}
 {9}
```

5.
```
public class LinkedList<E> implements List<E> {
        private class Node {
                E e;
                Node next = null;
                Node(E e) {
                        this.e = e;
                }
        }
        Node first = null, last = null;
```

```
        public void addLast(E elt) {
                Node n = new Node(elt);
                if (first == null) {
                        first = last = n;
                } else {
                        last.next = n;
                        last = n;
                }
        }

        public E get(int index) {
                Node n = first;
                for (; index > 0; index--) {
                        n = n.next;
                }
                return n.e;
        }
}
```

6. Utför beräkning av topologisk sortering som vanligt med skillnaden att noderna som står på kö läggs i en prioritetskö.

```
public List<Integer> topol() {
        List<Integer> list = new ArrayList<>();
        Map<Integer, Integer> indeg = new HashMap<>();
        for (Set<Integer> adjNodes : adj.values()) {
                for (Integer adjNode : adjNodes) {
                        int count = 0;
                        if (indeg.containsKey(adjNode)) {
                                count = indeg.get(adjNode);
                        }
                        indeg.put(adjNode, count + 1);
                }
        }
        PriorityQueue<Integer> q = new PriorityQueue<>();
        for (Integer node : adj.keySet()) {
                if (!indeg.containsKey(node)) {
                        q.add(node);
                }
        }
        while (!q.isEmpty()) {
                Integer node = q.poll();
                list.add(node);
                for (Integer adjNode : adj.get(node)) {
                        int count = indeg.get(adjNode) - 1;
                        indeg.put(adjNode, count);
```

```
                        if (count == 0) {
                                q.add(adjNode);
                        }
                }
        }
        if (list.size() == adj.size()) {
                return list;
        } else {
                return null;
        }
}
```

Första slingan genomlöper hashtabellen med grannlistorna. För stora V är hastabellens kapacitet begränsad av en V multiplicerad med en konstakt faktor. Den inre for-loopen i denna uppreapas totalt E gånger och `containsKey`, `get` och `put` tar idealt konstant tid. Komplexiteten för första slingan är $O(V + E)$.

Andra slingan upprepas V gånger och iteratorns stegning är liksom för första $O(V)$. `containsKey` tar konstant tid och `add` $O(V)$ eftersom storleken på prioritetskön maximalt är V. Komplexiteten för andra slingan är $O(V \log V)$.

I tredje slingan upprepas `q.poll`, `list.add` och `q.add` maximalt V gånger. Dessa har komplexiterna $O(\log V)$, $O(1)$ och $O(\log V)$. `indeg.get` och `indeg.put` upprepas maximalt E gånger och har båda komplexiteten $O(1)$. Tredje slingen har alltså komplexiteten $O(V \log V + E)$.

Hela metoden har komplexiteten $O(E + V \log V)$.