Tentamen
Datastrukturer, DAT037

- Datum och tid för tentamen: 2018-04-05, 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`:

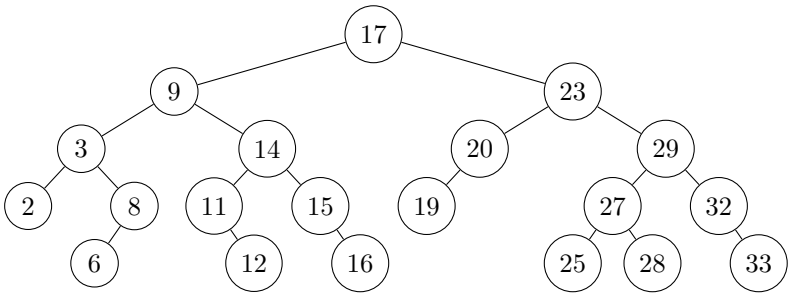
```
for (int i = 0; i < n; i++) {
    int x = q.dequeue();
    if (s.contains(x)) {
        pq.add(x);
    }
}
```

   Använd kursens uniforma kostnadsmodell och gör följande antaganden:

   - Att `int` kan representera alla heltal och att `n` är ett positivt heltal.
   - Att `q` är en kö implementerad med cirkulär array, dess element är av typen `int` och den innehåller från början minst `n` element.
   - Att `s` är en mängd implementerad med hashtabell, hashkoder tar konstant tid att beräkna, perfekt hashning råder och mängdens element är av typen `int`.
   - Att `pq` är en från början tom prioritetskö implementerad med leftist-heap och att dess element är av typen `int`.

   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 figur visar tillståndet i ett AVL-träd.



   Visa hur trädet ser ut efter att elementet 26 har satts in. Du behöver bara visa trädets slutliga utseende, ej några mellansteg.

3. Implementera en generisk datastruktur som representerar en samling med element av typen `E` (en typparameter) och har följande operationer:

   `empty()` som skapar en tom samling.

   `add(`x`)` som lägger till värdet x av typen `E` i samlingen.

   `deleteNewestMin()` som returnerar och tar bort det värde av typen `E` i samlingen som är minst. Om det finns fler än ett element i samlingen som är minst så ska det minsta värde som senast (mest nyligen) lades till samlingen returneras och tas bort. Om samlingen är tom så returneras `null`.

   Elementtypen, `E`, antas implementera `Comparable`. Detta behövs för att kunna avgöra vilka element som är minst.

   Antag att n är antal element i samlingen och att komparatorn har konstant tidskomplexitet. Operationerna måste ha följande tidskomplexiteter:

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

   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. Följande klass representerar en riktad, oviktad graf med heltal som nodetiketter.

```
class Graph {
    Map<Integer, List<Integer>> adj;
    ...
    List<TreeNode> spanningForest() { ... }
}
```

För representationen används grannlistor; varje nod i grafen avbildas i `adj` på en lista med de noder som är direkta efterföljare.

Implementera metoden `spanningForest` som beräknar en uppspännande skog för grafen. Den ska returnera en lista med trädnoder (`TreeNode`) där varje trädnod utgör (roten av) ett träd i skogen. `TreeNode` består av följande instansvariabler:

```
class TreeNode {
    int vertex;
    List<TreeNode> children;
}
```

`vertex` är etiketten för den nod i grafen som trädnoden motsvarar. `children` är en lista av barnnoder.

Metoden `spanningForest` ska ha tidskomplexiteten $O(V + E)$ där V är antalet noder i grafen och E antalet kanter. Du ska analysera koden och visa att så är fallet.

För fyra eller femma ska du också implementera en statisk metod som, utgående från en skog, skapar en lista med nodetiketter som definierar en ordning av noderna som ingår i skogen. Metoden ska fungera så, att om skogen är en uppspännande skog för en acyklisk graf så är ordningen en topologisk sortering för grafen. Metodens signatur ska vara följande:

```
static List<Integer> orderNodes(List<TreeNode> forest) { ... }
```

Metodens tidskomplexitet ska vara $O(n)$, där n är antalet noder i skogen.

Inga instans- eller klassvariabler ska läggas till i klasserna. Hjälpmetoder är tillåtna.

I båda deluppgifterna kan du använda standarddatastrukturer och -algoritmer utan att förklara hur de fungerar, undantaget grafalgoritmer. Implementationen av metoderna behöver inte beskrivas med detaljerad kod, men lösningen måste innehålla så mycket detaljer att tidskomplexiteten kan analyseras.

5. Uppgiften är att implementera delar av merge sort-algoritmen. Algoritmen ska delas upp på följande sätt:

```
public static void mergeSort(int[] x) {
    int[] tmp = new int[x.length];
    msort(x, tmp, 0, x.length - 1);
}

private static void msort(int[] x, int[] tmp,
                          int l, int r) {
   ...
}

private static void merge(int[] x, int[] tmp,
                          int l1, int l2, int r2) {
   ...
}
```

Implementeringen består av en huvudmetod, `mergeSort`, som skapar en temporär array, `tmp`, av samma längd som arrayen som ska sorteras (`x`). Precondition för denna metod är att $\texttt{x} \neq \texttt{null}$.

Sedan anropas en rekursiv hjälpmetod, `msort`. Preconditions för denna metod är att $\texttt{x} \neq \texttt{null}$, $\texttt{tmp} \neq \texttt{null}$, $\texttt{x.length} = \texttt{tmp.length}$ och $0 \leq \texttt{l} \leq \texttt{r}+1 \leq \texttt{x.length}$. Argumenten `l` och `r` anger det intervall av element i `x` som ska sorteras (`x[l]` ... `x[r]`). Resultatet av sorteringen ska hamna i samma element.

`msort` anropar, förutom sig själv, hjälpmetoden `merge`. Preconditions för denna metod är att $\texttt{x} \neq \texttt{null}$, $\texttt{tmp} \neq \texttt{null}$, $\texttt{x.length} = \texttt{tmp.length}$, $0 \leq \texttt{l1} \leq \texttt{l2} \leq \texttt{r2}+1 \leq \texttt{x.length}$ samt att elementen `x[l1]` ... `x[l2-1]` respektive elementen `x[l2]` ... `x[r2]` är inbördes sorterade. Metoden ska med hjälp av `tmp` sortera elementen `x[l1]` ... `x[r2]` och restultatet ska hamna i samma element.

Implementera metoden `msort` enligt den givna specifikationen. Tidskomplexiteten för `mergeSort` ska vara $O(n \log n)$, där $n = \texttt{x.length}$. Du kan förutsätta att `merge` har komplexiteten $O(\texttt{r2} - \texttt{l1})$ och att `x` inte är `null`. Analys av tidskomplexiteten behöver inte redovisas.

För fyra eller femma ska du också implementera metoden `merge` och den ska ha ovan angivna tidskomplexitet. Du ska också visa att denna metod uppfyller komplexitetskravet.

Endast detaljerad kod godkänns, ej pseudokod.

6. Konstruera en datastruktur för att representera en mängd av heltal. Datastrukturen ska ha följande operationer:

   `empty()` Skapar en tom mängd.

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

   `contains(`x`)` Returnerar `true` om talet x finns i mängden, annars `false`.

   `nextNonMember(`x`)` Returnerar det minsta heltal som inte finns i mängden och som är större än eller lika med x.

   Om mängden innehåller talen $\{4, 6, 8, 9, 10, 15, 16, 18\}$ så ska `nextNonMember(9)` returnera 11 och `nextNonMember(13)` returnera 13.

   Antag att n är antal element i mängden. Operationerna måste ha följande tidskomplexiteter:

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

   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.

Lösningsförslag till tentamen

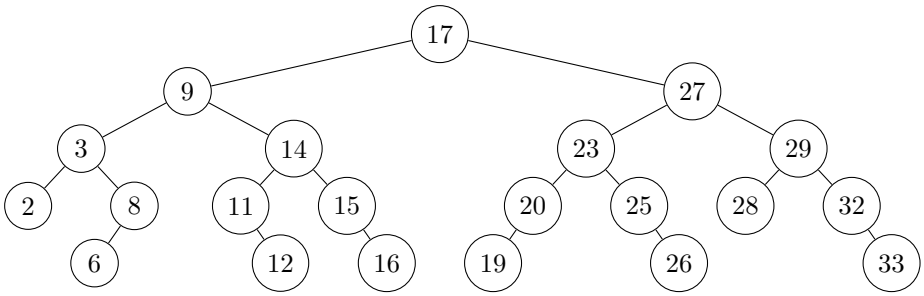
Datastrukturer, DAT037, 2018-04-05

1.
   - `q.dequeue()` tar $O(1)$ (eventuellt amorterat)
   - `s.contains(x)` tar $O(1)$
   - `pq.add(x)` tar $O(\log i)$

   I värsta fall exekveras innehållet i if-satsen. En iteration tar $O(1+1+\log i)$. Loopen itereras för $i = 0, 1, \ldots, \mathtt{n} - 1$.

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

2. Efter att ha satt in 26 får man följande träd:



3. För trea kan man spara elementen sist i en lista och söka linjärt efter minsta elementet enligt komparatorn.

   För högre betyg kan man använda en binär heap där elementens inbördes ordning förutom den generiska typen också avspeglar insättningsordningen.

   Låt representationen vara

```
Ett heltal t.
En prioritetskö q implementerad med en binär heap.
  Kön innehåller par av typen <E, int>.
  Ordningen, komparatorn, som används för jämförelser i kön jämför först det första
   komponenten i paren.
  Om de är olika är det denna ordning som gäller.
  Om de är lika är det omvända ordningen mellan andra komponenten i paren som gäller.
   (Större heltal innebär högre prioritet.)
```

   Konstruktorn och operationerna implementeras så här:

```
empty:
  t = 0  O(1)
  q = tom binär heap med komparator enligt beskr. ovan  O(1)

add(x):
  q.add(<x, t>)  O(log n)
  t = t + 1  O(1)

deleteNewestMin():
  p = q.deleteMin()  O(log n)
  returnera första komponenten i paret p (värdet av typen E)  O(1)
```

   Genom att definiera ordningen för kö-elementen på det sättet som beskrivs ovan och öka `t` vid varje insättning kommer det nyaste element tas bort när det finns fler än ett minsta element.

   Konstruktorn har tidskomplexiteten $O(1)$ och operationerna $O(\log n)$.

4. För att beräkna uppspännande skog kan man utföra en dfs varje besökt nod skapar en trädnod och varje omstart skapar ett nytt träd.

```
public List<TreeNode> spanningForest() {
    List<TreeNode> sf = new ArrayList<>();  // O(1)
    Set<Integer> visited = new HashSet<>();  // O(1)
    Iterator<Integer> i = adj.keySet().iterator();  // O(1)
    while (i.hasNext()) {  // O(1) (villkoret)
        int v = i.next();  // O(1)
        TreeNode t = dfs(visited, v);
        if (t != null) {
            sf.add(t);  // O(1) amorterat
        }
    }
    return sf;
}
private TreeNode dfs(Set<Integer> visited, int v) {
    if (visited.contains(v)) return null;  // O(1)
    visited.add(v);  // O(1)
    TreeNode t = new TreeNode();  // O(1)
    t.data = v;
    List<Integer> a = adj.get(v);  // O(1)
    for (int i = 0; i < a.size(); i++) {
        TreeNode st = dfs(visited, a.get(i));
        if (st != null) {
            t.children.add(st);  // O(1) amorterat
        }
    }
    return t;
}
```

Loopen i `spanningForest` upprepas V gånger. `dfs` anropas max $V + E$ gånger. Koden fr o m rad två i `dfs` utförs V gånger. Loopen i `dfs` utförs totalt E gånger eftersom varje gången exekveringen når loopen så är det för en ny nod. Tidskomplexiteten blir $O(V + E)$.

För fyra eller femma kan man implementera en preorder höger-till-vänster traversering av skogen. Detta ger en topologisk sortering för en uppspännande skog av en acyklisk graf.

```
static List<Integer> preorderRL(List<TreeNode> f) {
    List<Integer> l = new ArrayList<>();
    preorderRL(f, l);
    return l;
}
private static void preorderRL(List<TreeNode> f, List<Integer> l) {
    for (int i = f.size() - 1; i >= 0; i--) {
        l.add(f.get(i).data);
        preorderRL(f.get(i).children);
    }
}
```

5.
```
private static void msort(int[] x, int[] tmp, int l, int r) {
    if (r <= l) return;
    int m = l + (r - l) / 2;
    msort(x, tmp, l, m);
    msort(x, tmp, m + 1, r);
    merge(x, tmp, l, m + 1, r);
}

private static void merge(int[] x, int[] tmp, int l1, int l2, int r2) {
    int r1 = l2 - 1, i = l1, i1 = l1, i2 = l2;
    while (i1 <= r1 && i2 <= r2) {
     // repeated at most (r1 - l1 + 1) + (r2 - l2 + 1) = r2 - l1 + 1 times
        if (x[i1] <= x[i2]) tmp[i++] = x[i1++];
        else tmp[i++] = x[i2++];
    }
    while (i1 <= r1)  // repeated at most r1 - l1 + 1 = l2 - l1 times
        tmp[i++] = x[i1++];
    while (i2 <= r2)  // repeated at most r2 - l2 + 1 times
        tmp[i++] = x[i2++];
    for (int j = l1; j <= r2; j++)  // repeated r2 - l1 + 1 times
        x[j] = tmp[j];
}
```

Alla satser förutom looparna i `merge` tar konstant tid. De första tre looparna upprepas tillsammans $\texttt{r2}-\texttt{l1}+1$ gånger. Den sista loopen upprepas lika många gånger. Tidskomplexiteten är $O(\texttt{r2}-\texttt{l1}+1) = O(\texttt{r2}-\texttt{l1})$.

6. För trea kan man spara heltalen sorterat i en lista.

   För fyra eller femma kan man använda ett balanserat sökträd men inte lagra heltal i noderna utan intervall (två heltal som representerar det minsta och största talet i intervallet). Ett intervall är mindre än ett annat om talen som det innehåller är mindre än talen som det andra intervallet innehåller. Inget tal kommer att finnas med i mer än ett intervall och inget intervall kommer vara tomt.

   Använd en hjälpklass för intervall:

```
class Interval {
    int first, last;
}
```

   Definiera Konstruktorn och operationerna så här.

```
empty() =
 t = tomt balanserat sökträd med talpar som element ordnade enligt ovan  O(1)

add(x) =
 Interval i = find(x);  // O(log n)
 if (i != null) return;
 Interval il = find(x - 1);  // O(log n)
 Interval ir = find(x + 1);  // O(log n)
 if (il != null) t.remove(il);  // O(log n)
 if (ir != null) t.remove(ir);  // O(log n)
 Interval newi = new Interval();
 newi.first = il != null ? il.first : x;
 newi.last = ir != null ? ir.last : x;
 t.add(newi);  // O(log n)

contains(x) =
 return find(x) != null;  // O(log n)

nextNonMember(x) =
 Interval i = find(x);  // O(log n)
 if (i == null) return x;
 return i.last + 1;
```

   `t.remove` och `t.add` är standardborttagning och -insättning för det balanserade sökträdet.

`find(x)` är en hjälpmetod som avgör om heltalet `x` finns i någon av intervallen i `t`. Finns talet med så returneras det intervall som det ingår i, annars returneras `null`.

```
find(x) =  O(log n)
 n = t.root;  // trädets rotnod
 while (n != null) {  // O(log n)
     if (n.data.first <= x && n.data.last >= x) {
         return n.data;
     } else if (n.data.first > x) {
         n = n.left;
     } else {
         n = n.right;
     }
 }
 return null;
```

Koden ovan utgår från att `t.root` utgör rotnoden i trädet och att noderna tillhör följande klass:

```
class Node {
    Interval data;
    Node left, right;
}
```

Varje iteration av loopen i `find` tar $O(1)$. Loopen upprepas $O(\log n)$ gånger eftersom trädet är balanserat.