Tentamen
Datastrukturer (DAT036)

- Datum och tid för tentamen: 2013-08-23, 8:30–12:30.
- Ansvarig: Nils Anders Danielsson. Nås på 0700 620 602 eller anknytning 1680. Besöker tentamenssalarna ca 9:30 och ca 11:30.
- Godkända hjälpmedel: Ett A4-blad med handskrivna anteckningar.
- För att få betyget n (3, 4 eller 5) måste du lösa minst n uppgifter. Om en viss uppgift har deluppgifter med olika gradering så räknas uppgiften som löst om du har löst *alla* deluppgifter med gradering n eller mindre.
- För att en uppgift ska räknas som "löst" så måste en i princip helt korrekt lösning lämnas in. Enstaka mindre allvarliga misstag kan *eventuellt* godkännas (kanske efter en helhetsbedömning av tentan; för högre betyg kan kraven vara hårdare).
- 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. Pseudokod får 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, men däremot motivera deras användning.

1. Analysera nedanstående kods tidskomplexitet, uttryckt i `n`:

   ```
   for (int i = 0; i < n; i++) {
     if (i < 10) {
       t.insert(i);
     }
   }
   ```

   Använd kursens uniforma kostnadsmodell, och gör följande antaganden:

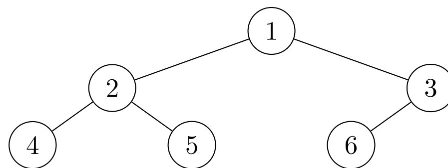
   - Att `n` är ett ickenegativt heltal, och att typen `int` kan representera alla heltal.
   - Att `t` är ett splayträd som till att börja med är tomt.
   - Att den vanliga ordningen för heltal ($... < -1 < 0 < 1 < 2 < ...$) används vid insättning i trädet.
   - Att om samma element sätts in två gånger i trädet så skrivs den tidigare förekomsten över.

   Svara med ett enkelt uttryck (inte en summa, rekursiv definition, eller dylikt). Onödigt oprecisa analyser kan underkännas; använd gärna Θ-notation.

2. Binära heapar använder arrayer för att representera träd. Träd kan också representeras av pekarstrukturer. Implementera en metod som konverterar ett arrayträd till ett pekarträd. Använd följande klass för att representera pekarträd:

```
public class Tree<A> {

    // Trädnoder; null representerar tomma träd.
    private class Node {
        A    contents;  // Innehåll.
        Node left;      // Vänstra barnet.
        Node right;     // Högra barnet.

        // Skapar en trädnod.
        Node(A contents, Node left, Node right) {
            this.contents = contents;
            this.left     = left;
            this.right    = right;
        }
    }

    // Roten.
    private Node root;

    // Din uppgift.
    public Tree(A[] t) {
        ...
    }
}
```

Exempel: Arrayen $\{1, 2, 3, 4, 5, 6\}$ ska konverteras till följande träd:



Endast detaljerad kod (inte nödvändigtvis Java) godkänns. Du får inte anropa några andra metoder/procedurer (förutom `Node`-konstrueraren), om du inte implementerar dem själv.

Metoden måste vara linjär i arrayens storlek ($O(n)$, där n är storleken). Visa att så är fallet.

Tips: Testa din kod, så kanske du undviker onödiga fel.

3. Beskriv en *linjär* algoritm som, givet två listor innehållandes heltal, avgör om listorna innehåller samma heltal.

   Exempel:

Lista 1	Lista 2	Resultat
$[0, 1, 2, 3]$	$[3, 1, 0, 2]$	Sant
$[0, 1, 2, 3]$	$[3]$	Falskt
$[0, 1]$	$[0, 1, 1]$	Sant

   Tips: Testa din algoritm, så kanske du undviker onödiga fel. Använd gärna standarddatastrukturer och/eller -algoritmer från kursen.

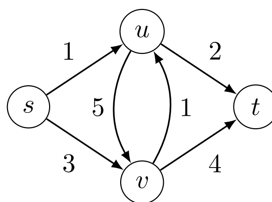
4. Anta att vi, givet en array `xs` med `n` distinkta naturliga tal och ett naturligt tal $\texttt{k} \leq \texttt{n}$, vill beräkna summan av de `k` största talen i arrayen. Följande algoritm fungerar inte:

   ```
   q = new Binary-Max-Heap()

   for each element x in xs do
     q.insert(x)
     if q.size() > k then
       q.delete-max()

   sum = 0
   while q.size() > 0 do
     sum = sum + q.delete-max()

   return sum
   ```

   Demonstrera med ett exempel att algoritmen ibland ger ett felaktigt resultat, och visa att det går att förändra algoritmen så att den blir korrekt.

5. Beskriv en effektiv algoritm som, givet en riktad, viktad graf G med icke-negativa heltal som kantvikter, och två noder s och t från G, avgör om det finns en väg från s till t *med ett udda antal steg* och i så fall beräknar den minsta möjliga vikten för en sådan väg. Analysera algoritmens tidskomplexitet.

   Exempel: För den följande grafen ska svaret vara 6 (och inte 3):

6. Uppgiften är att konstruera en datastruktur för en mängd-ADT med följande operationer:

   `new Set(`n`)` Konstruerar en tom mängd. Mängden får bara innehålla ickenegativa heltal mindre än n, mängdens *kapacitet*. (Du kan anta att n är ett ickenegativt heltal.)

   `insert(`i`)` Utökar mängden med heltalet i. Om i redan finns i mängden så lämnas den oförändrad. (Du kan anta att $0 \leq i < n$, där n är mängdens kapacitet.)

   `member(`i`)` Avgör om heltalet i finns i mängden. (Du kan anta att $0 \leq i < n$, där n är mängdens kapacitet.)

   `union(`s`)` Utökar mängden med alla element från mängden s som inte redan finns i mängden. Operationen får förstöra s. (Du kan anta att de två mängderna har samma kapacitet.)

   Exempel: Följande kod, skriven i ett Javaliknande språk, ska ge `true` som svar:

```
boolean b;

Set s1 = new Set(3);
s1.insert(0);
b = s1.member(0) && ! s1.member(1) && ! s1.member(2);

Set s2 = new Set(3);
s2.insert(1);
b = b && ! s2.member(0) && s2.member(1) && ! s2.member(2);

s1.union(s2);
b = b && s1.member(0) && s1.member(1) && ! s1.member(2);

return b;
```

   Du måste visa att operationerna uppfyller följande tidskomplexitetskrav (där n är mängdernas kapacitet):

   - *För trea:* `new`: $O(n)$, `insert`, `member`: $O(1)$, `union`: $O(n)$.
   - *För fyra:* `new`: $O(n)$, `insert`, `member`, `union`: $O(1)$.

   Kraven ska läsas på *ett* av följande sätt: som värstafallstidskomplexitet eller som amorterad (värstafalls-) tidskomplexitet.

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

   Tips: Testa din algoritm, så kanske du undviker onödiga fel.

Delvis kortfattade lösningsförslag för tentamen i Datastrukturer (DAT036) från 2013-08-23

Nils Anders Danielsson

1. Notera att `t` kommer att innehålla som mest 10 olika värden. Tidskomplexiteten blir $\Theta(n)$.

2. Följande implementation är linjär eftersom konstant arbete utförs för varje element (plus konstant övrigt arbete):

```
// Skapar ett träd med samma struktur och innehåll som t.
// Arrayen t ses som ett komplett binärt träd, med noderna
// i nivåordning, och en eventuell rot på position 0.
public Tree(A[] t) {
    if (t == null) {
        return;
    } else {
        root = fromArray(t, 0);
    }
}

private Node fromArray(A[] t, int pos) {
    if (pos >= t.length) {
        return null;
    } else {
        return new Node(t[pos],
                        fromArray(t, pos * 2 + 1),
                        fromArray(t, pos * 2 + 2));
    }
}
```

3. Se uppgift 3 från duggan som gavs 2012-11-21. (Lösningen behöver modifieras något eftersom vi bara ska avgöra om listorna innehåller samma element, och inte om de är permutationer av varandra.)

4. Om `n` $= 2$, `k` $= 1$ och `xs` $= \{1, 2\}$ så blir resultatet 1, inte 2. Algoritmen kan fås att fungera genom att byta ut maxheapen mot en minheap, och `delete-max` mot `delete-min`.

5. Anta att grafen är $G = (V, E)$. (Jag har valt att låta kantmängden E bestå av tripplar av formen (u, v, w), där $u, v \in V$ och $w \in \mathbb{N}$. En sådan trippel står för en kant från u till v med vikten w.) Vi kan konstruera en ny graf $G' = (V', E')$ där

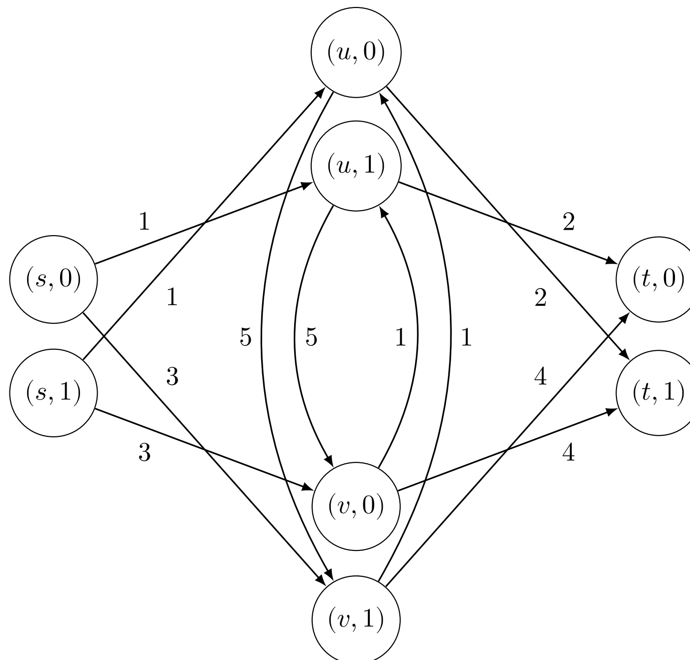
$$V' = \{ (v, 0) \mid v \in V \} \cup \{ (v, 1) \mid v \in V \}$$

och

$$\begin{aligned} E' = \; & \{ ((u, 0), (v, 1), w) \mid (u, v, w) \in E \} \cup \\ & \{ ((u, 1), (v, 0), w) \mid (u, v, w) \in E \} . \end{aligned}$$

Den nya grafen G' innehåller två noder för varje nod v i G: den "jämna" noden $(v, 0)$ och den "udda" noden $(v, 1)$. För varje kant från u till v i G finns det också två kanter i G' (med samma vikt som kanten i G): en kant från den jämna noden $(u, 0)$ till den udda noden $(v, 1)$, och en kant från den udda noden $(u, 1)$ till den jämna noden $(v, 0)$.

Exempel: Om G är grafen i uppgiftsspecifikationen så blir G' följande graf:



Notera att för varje väg

$$v_0 \xrightarrow{w_1} v_1 \xrightarrow{w_2} ... \xrightarrow{w_n} v_n$$

i G så finns det en motsvarande väg

$$(v_0, 0) \xrightarrow{w_1} (v_1, 1) \xrightarrow{w_2} ... \xrightarrow{w_n} (v_n, p)$$

i G', där p är 0 om n är jämn och 1 om n är udda. Vidare, för varje väg

$$(v_0, p_0) \xrightarrow{w_1} (v_1, p_1) \xrightarrow{w_2} ... \xrightarrow{w_n} (v_n, p_n)$$

i G' så finns det en väg

$$v_0 \xrightarrow{w_1} v_1 \xrightarrow{w_2} ... \xrightarrow{w_n} v_n$$

i G. Vi kan således avgöra om det finns en väg med ett udda antal steg från s till t i G genom att avgöra om det finns en väg från $(s, 0)$ till $(t, 1)$ i G', och i så fall har den kortaste (lättaste) vägen från $(s, 0)$ till $(t, 1)$ i G' samma vikt som den kortaste vägen med ett udda antal steg från s till t i G.

Om grannlistor används så kan algoritmen implementeras med följande tidskomplexitet:

- Konstruera G': $O(|V| + |E|)$.
- Avgöra om det finns en väg från $(s, 0)$ till $(t, 1)$ i G', och i så fall beräkna den minsta möjliga vikten för en sådan väg (med Dijkstras algoritm): $O(|V'| + |E'| \log |V'|) = O(|V| + |E| \log |V|)$.

Totalt: $O(|V| + |E| \log |V|)$, vilket nog får anses vara effektivt.

6. En Javaimplementation där mängderna representeras av bitarrayer:

```
public class Set {
    private boolean[] bits;

    public Set(int n) {
        bits = new boolean[n];
        for (int p = 0; p < n; p++) {
            bits[p] = false;
        }
    }

    public void insert(int i) {
        bits[i] = true;
    }

    public boolean member(int i) {
        return bits[i];
    }
```

```
    public void union(Set s) {
        for (int p = 0; p < bits.length; p++) {
            bits[p] = bits[p] || s.bits[p];
        }

        s.bits = null;
    }
}
```

Det är lätt att se att operationernas värstafallstidskomplexiteter uppfyller kravet för trea. En amorterad analys med (en positiv konstant gånger) summan av alla bitarrayernas längder som potential visar att vi också uppfyller kravet för fyra. (Notera att den amorterade analysen inte skulle fungera om raden `s.bits = null;` togs bort.)