Tentamen
Datastrukturer (DAT036/DAT037/DIT960)

- Datum och tid för tentamen: 2016-04-07, 14:00–18:00.
- Författare: Nils Anders Danielsson. (Tack till Per Hallgren och Nick Smallbone för feedback.)
- Ansvarig: Nils Anders Danielsson. Nås på anknytning 1680. Besöker tentamenssalarna ca 15:00 och ca 17:00.
- Godkända hjälpmedel: Ett A4-blad med handskrivna anteckningar.
- För att få betyget n (3, 4 eller 5) på tentan måste man få betyget n eller högre på minst n uppgifter. För att få betyget G måste man uppfylla kraven för betyget 3, och för betyget VG måste man uppfylla kraven för betyget 5.
- 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.
- Om en viss uppgift har deluppgifter med olika gradering (t ex ”För trea:”, ”För fyra:”) så behöver man, för att få betyget n på den uppgiften, få betyget n eller högre på alla deluppgifter med gradering n eller lägre.
- 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++) {
       xs.addFirst(q.dequeue());
   }
   ```

   Använd kursens uniforma kostnadsmodell, och gör följande antaganden:

   - Att `n` är ett positivt heltal, och att typen `int` kan representera alla heltal.
   - Att `xs` är en dynamisk array som till att börja med är tom, och att `xs.addFirst(x)` sätter in `x` först i `xs`.
   - Att `q` är en (dynamisk) cirkulär array som till att börja med innehåller `n` heltal.

   Svara med ett enkelt uttryck (inte en summa, rekursiv definition, eller dylikt). Onödigt oprecisa analyser kan underkännas.

2. Beskriv en algoritm som, givet ett binärt träd med möjligtvis felaktiga föräldrapekare, avgör om föräldrapekarna pekar på rätt noder. (En föräldrapekare pekar från en nod till dess förälder.) För betyget fyra måste du dessutom visa att din algoritm är linjär i trädets storlek ($O(n)$, där n är storleken).

   Du kan använda följande trädklass:

   ```
   // Binära träd med föräldrapekare.
   public class Tree<A> {

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

       // Roten.
       private Node root;

       // Din uppgift.
       private boolean parentsAreCorrect() {
           ...
       }

       ...
   }
   ```

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

Du kan om du vill anta att trädets djup är $O(\log n)$, där n är antalet noder i trädet.

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

3. Uppgiften är att konstruera en datastruktur för en prioritetskö-ADT med följande operationer:

   `empty()` eller `new PriorityQueue()` Konstruerar en tom prioritetskö.

   `insert(`v`)` Sätter in värdet v i kön.

   `deleteMin()` Tar bort och ger tillbaka det minsta värdet i kön, eller om det finns flera värden som är minst, något av dem. *Precondition*: Kön får inte vara tom.

   Alla värden är heltal.

   Om värdena sätts in i sorterad ordning så måste operationerna ha följande amorterade tidskomplexiteter:

   - `empty`, `insert`, `deleteMin`: $O(1)$.

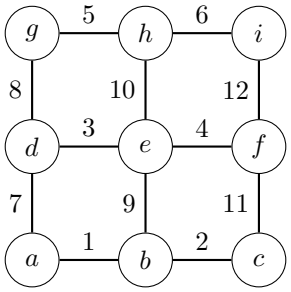
   Oavsett ordningen på insatta element så måste operationerna ha följande amorterade tidskomplexiteter (där ℓ är antalet element i kön):

   - *För trea:* `empty`: $O(1)$, `insert`, `deleteMin`: $O(\ell)$.
   - *För fyra:* `empty`: $O(1)$, `insert`: $O(\log \ell)$, `deleteMin`: $O(\log \ell)$.

   För betyget fyra 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 (oavsett betygsnivå).

   Tips: Konstruera om möjligt inte datastrukturen från grunden, bygg den hellre m h a standarddatastrukturer. Testa dina algoritmer, så kanske du undviker onödiga fel. Halvfärdiga lösningar godkänns knappast; om du inte lyckas konstruera en *korrekt* datastruktur som uppfyller kravet för ett visst betyg så kan det vara en bra idé att istället försöka konstruera en enklare datastruktur som uppfyller kravet för ett lägre betyg.

4. Använd Prims algoritm för att ta fram ett minsta uppspännande träd för följande graf:



Börja i nedre vänstra hörnet (nod a). Visa steg för steg vilka kanter som läggs till trädet.

Det räcker att svara med en lista med kanter. Om du vill får du motivera din lösning, vilket kanske kan vara till hjälp vid slarvfel.

5. Betrakta följande implementation av binära sökträd:

```
module BinarySearchTree
  (Tree, keys, values, empty, insert, delete)
  where

-- Binära sökträd. Notera att varje nyckel (av typ Int)
-- svarar mot en /lista/ innehållandes värden (av typ a),
-- d v s varje nyckel svarar mot noll eller flera värden.
data Tree a = Empty | Node (Tree a) Int [a] (Tree a)

-- Antalet nycklar i trädet.
keys :: Tree a -> Int
keys Empty          = 0
keys (Node l _ _ r) = 1 + keys l + keys r

-- Antalet värden i trädet.
values :: Tree a -> Int
values Empty           = 0
values (Node l _ vs r) = length vs + values l + values r

-- Ett tomt träd.
empty :: Tree a
empty = Empty

-- Sätter in en nyckel och ett värde i trädet. Om nyckeln
-- redan finns i trädet så läggs värdet till i nyckelns
-- lista, annars läggs en ny nod till.
insert :: Int -> a -> Tree a -> Tree a
insert k v Empty            = Node Empty k [v] Empty
insert k v (Node l k' vs r) = case compare k k' of
  LT -> Node (insert k v l) k' vs r
  GT -> Node l k' vs (insert k v r)
  EQ -> Node l k' (v : vs) r

-- Tar bort en av nyckelns värden från trädet. Om nyckeln
-- inte hittas, eller motsvarande lista är tom, så lämnas
-- trädet oförändrat.
delete :: Int -> Tree a -> Tree a
delete k Empty            = Empty
delete k (Node l k' vs r) = case compare k k' of
  LT -> Node (delete k l) k' vs r
  GT -> Node l k' vs (delete k r)
  EQ -> case vs of
    []      -> Node l k' vs  r
    _ : vs' -> Node l k' vs' r
```

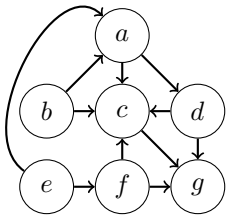
Analysera följande uttrycks tidskomplexitet: `delete k t`. Tidskomplexiteten ska uttryckas antingen i termer av antalet nycklar i trädet (`keys t`), eller antalet värden i trädet (`values t`) – välj det som är lämpligast. Använd inte variabeln n i ditt svar utan att definiera vad n står för.

Anta att typen `Int` kan representera alla heltal.

Notera att typens konstruerare inte exporteras från modulen, så sökträdsinvarianten gäller för alla träd.

Svara med ett enkelt uttryck (inte en summa, rekursiv definition, eller dylikt). Onödigt oprecisa analyser kan underkännas.

6. (a) *För trea:* Sortera följande graf topologiskt:



Svara med en lista, sorterad i topologisk ordning.

Det räcker att svara med en lista med noder. Om du vill får du motivera din lösning, vilket kanske kan vara till hjälp vid slarvfel.

(b) *För fyra:* Betrakta följande ofärdiga algoritm för att hitta de kortaste vägarna från en nod s till alla andra noder i en riktad, oviktad graf:

i. Allokera en tom avbildning d från noder till heltal. Sätt $d(s)$ till 0.

ii. Gå igenom grafen i _____-ordning, med utgångspunkt från s. När en nod v (förutom s) besöks första gången, via en kant $u \to v$, sätt $d(v)$ till $d(u) + 1$.

iii. Ge d som svar. (Avsaknad av en bindning $d(w)$ betyder att det inte finns någon väg från s till w.)

Notera platshållaren ovan. För vilka av följande uttryck fungerar algoritmen, om platshållaren byts ut mot det givna uttrycket?

- Djupet först.
- Bredden först.

Om algoritmen alltid fungerar för ett visst uttryck behöver svaret inte motiveras. Om algoritmen inte fungerar för ett visst uttryck, ge då en graf för vilken algoritmen kan ge fel svar, och förklara i detalj varför algoritmen inte fungerar för den givna grafen.

Delvis kortfattade lösningsförslag för tentamen i Datastrukturer (DAT036/DAT037/DIT960) från 2016-04-07

Nils Anders Danielsson

1. Notera först att alla `dequeue`-operationerna kommer att lyckas.

   Insättning först i `xs` har tidskomplexiteten $\Theta(\ell)$, där ℓ är antalet element i arrayen. Borttagning av det första elementet i `q` har tidskomplexiteten $\Theta(1)$ (möjligtvis amorterat, beroende på hur den cirkulära arrayen implementerats). Den totala tidskomplexiteten blir

   $$\Theta\left(\sum_{i=0}^{\mathtt{n}-1}(1+i)\right) = \Theta(\mathtt{n}^2).$$

2. Om trädet inte är alltför djupt så behöver vi kanske inte oroa oss för ”stack overflow”, och i så fall kan en rekursiv implementation vara lämplig:

   ```
   private boolean parentsAreCorrect() {
       return parentsAreCorrect(root, null);
   }

   private boolean parentsAreCorrect(Node n, Node parent) {
       return n == null
           || (n.parent == parent
               && parentsAreCorrect(n.left, n)
               && parentsAreCorrect(n.right, n));
   }
   ```

   Eftersom algoritmen utför $O(1)$ arbete per trädnod, och $O(1)$ övrigt arbete, så är algoritmen linjär i trädets storlek.

3. Använd en binär heap och en kö (en dynamisk cirkulär array), och sätt in varje värde antingen i heapen eller i kön. Invariant: Kön är sorterad. Implementation:

   - `empty`: Skapa en tom heap och en tom kö.
   - `insert`(v): Om kön är tom, eller om köns sista element $v' \leq v$, sätt in v sist i kön (som fortsätter vara sorterad). Annars, sätt in v i heapen.

- `deleteMin`: Om kön är icketom och antingen heapen är tom, eller köns första element är mindre än eller lika med heapens minsta element, ta bort och ge tillbaka köns första element. (Notera att kön fortsätter vara sorterad.) Annars, ta bort och ge tillbaka heapens första element. (Notera att prioritetskön antas vara icketom, så kön och heapen kan inte båda vara tomma.)

Om värdena sätts in i sorterad ordning så kommer inget element sättas in i heapen, och tidskomplexiteten blir i så fall $O(1)$ (amorterat) för varje operation. I det generella fallet blir tidskomplexiteten $O(\log \ell)$ (amorterat) för både `insert` och `deleteMin`.

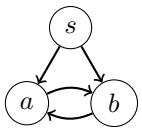
4. 1, 2, 7, 3, 4, 8, 5, 6.

5. Tidskomplexiteten för `delete k t` är, i värsta fallet, linjär i trädets höjd, och eftersom trädet inte behöver vara balanserat så är höjden i värsta fallet linjär i antalet noder (d v s nycklar) i trädet. Svaret blir $O(n)$, där n är antalet nycklar i trädet.

   Notera att svaret $O(v)$, där v är antalet värden i trädet, är fel. Det är möjligt att skapa träd med godtyckligt många nycklar, där varje nyckel svarar mot en tom lista.

6. (a) Alla korrekta svar:
   - b, e, a, d, f, c, g.
   - b, e, a, f, d, c, g.
   - b, e, f, a, d, c, g.
   - e, b, a, d, f, c, g.
   - e, b, a, f, d, c, g.
   - e, b, f, a, d, c, g.
   - e, f, b, a, d, c, g.

   (b) Algoritmen fungerar om bredden först-sökning används, men inte om djupet först-sökning används. Motexempel:



   Anta att djupet först-sökning används. Om sökningen besöker a innan b, så kommer det beräknade kortaste avståndet till b att bli 2 istället för 1. Om sökningen istället besöker b innan a så kommer det beräknade kortaste avståndet till a att bli 2 istället för 1.