Tentamen
Datastrukturer (DAT036)

- Datum och tid för tentamen: 2014-04-25, 14:00–18:00.
- Författare: Nils Anders Danielsson.
- Ansvarig: Nils Anders Danielsson. Nås på 0700 620 602 eller 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) 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 tidskomplexiteten för `print-graph(g)`, uttryckt i antalet noder ($|V|$) och kanter ($|E|$) i `g`:

```
print-graph(g) {
    for every node u in g {
        println(u);
        for every node v adjacent to u in g {
            println("  -> " + v);
        }
    }
}
```

   Använd kursens uniforma kostnadsmodell, och gör följande antaganden:

   - Att `g` representeras med grannlistor.
   - Att `println(...)` tar konstant tid.

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

2. Implementera en metod/funktion `deleteAll` som tar bort alla förekomster av ett givet heltal från en lista. Operationen får (men måste inte) förstöra den ursprungliga listan.

   Exempel: Resultatet av att ta bort 0 från $[3, 0, 2, 4, 0]$ ska vara $[3, 2, 4]$, och resultatet av att ta bort 0 från $[0, 0]$ ska vara $[\,]$.

   Du måste representera listor på ett av följande sätt:

   - Som vanliga Haskellistor, innehållandes heltal.
   - Med följande Javaklass:

```
public class SinglyLinkedList {
    private class Node {
        public int contents;  // Nodens innehåll.
        public Node next;     // Nästa nod; null för
                              // sista noden.

        public Node() {}
    }

    private Node first;  // Första noden; null om listan
                         // är tom.

    public SinglyLinkedList() {}

    public void deleteAll(int i) {
        // Din uppgift.
    }
}
```

   Endast detaljerad kod (inte nödvändigtvis Haskell eller Java) godkänns. Bortsett från listkonstruerare och operatorer för att testa (o)likhet får du inte anropa andra procedurer e d (inkl standardfunktioner som `(++)`), om du inte implementerar dem själv som en del av din lösning.

   Metoden/funktionen måste vara linjär i listans längd.

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

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

   `empty()` eller `new Set()` Konstruerar en tom mängd.

   `insert(`i`)` Lägger till heltalet i till mängden. (Lämnar mängden oförändrad om i redan finns i mängden.)

   `delete(`i`)` Tar bort heltalet i från mängden. (Lämnar mängden oförändrad om i inte finns i mängden.)

   `member(`i`)` Avgör om heltalet i finns i mängden.

   `delete-min-max()` Tar bort både det största och det minsta heltalet från mängden. (Lämnar mängden oförändrad om den är tom; om mängden innehåller ett enda tal tas endast det här talet bort.)

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

```
Set s = new Set();
s.insert(0); s.insert(1); s.insert(2); s.insert(3);
boolean b = s.member(0) && s.member(1)
          && s.member(2) && s.member(3);
s.delete(2);
b = b &&    s.member(0)  && s.member(1)
      && (! s.member(2)) && s.member(3);
s.delete-min-max();
b = b && (! s.member(0)) &&    s.member(1) &&
         (! s.member(2)) && (! s.member(3));
s.delete-min-max();
b = b && (! s.member(0)) && (! s.member(1)) &&
         (! s.member(2)) && (! s.member(3));
return b;
```

   Operationerna måste ha följande tidskomplexiteter (där n är antalet element i mängden):

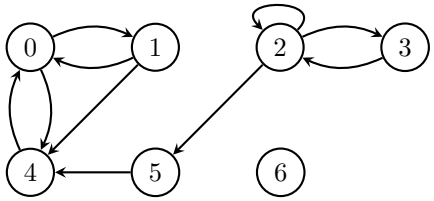
   - *För trea:* `new`: $O(1)$, `insert`: $O(\log n)$, `delete`, `member`, `delete-min-max`: $O(n)$.
   - *För fyra:* `new`: $O(1)$, `insert`, `delete`, `member`, `delete-min-max`: $O(\log n)$.
   - *För femma:* `new`, `member`: $O(1)$, `insert`, `delete`, `delete-min-max`: $O(\log n)$.

   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.

   Tips: Konstruera om möjligt inte datastrukturen från grunden, bygg den hellre m h a standarddatastrukturer. Testa din algoritm, så kanske du undviker onödiga fel.

4. Sätt in värdena 1, 2, …, 6 i en från början tom binomialheap. Heapen ska vara en *minheap*, och den vanliga ordningen för naturliga tal ska användas. Visa hur heapen ser ut efter varje insättning.

5. (a) *För trea:* Hur många starkt sammanhängande komponenter har följande graf, och vilka noder innehåller de?



   (b) *För fyra:* Uppgiften är att konstruera en datastruktur som representerar riktade, oviktade grafer, och uppfyller följande krav:

   i. Givet två noder ska det gå att avgöra, på konstant tid, om noderna hör till samma starkt sammanhängande komponent.

   ii. Givet två noder u, v ska det gå att avgöra, på konstant tid, om det finns en kant från u till v.

   Visa att kraven ovan uppfylls. Beskriv också en algoritm som lägger till en kant till grafen (om kanten inte redan finns där), och analysera dess tidskomplexitet. Algoritmen behöver inte vara effektiv.

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

   Tips: Konstruera om möjligt inte datastrukturen från grunden, bygg den hellre m h a standarddatastrukturer.

6. Vi har inte gått igenom ”skeva heapar” (skew heaps) tidigare i kursen. De kan representeras i Haskell på följande sätt:

```
data SkewHeap a = Node (SkewHeap a) a (SkewHeap a)
                | Empty
```

Den centrala operationen för skeva heapar är sammanslagning, som kan implementeras på följande sätt (se t ex Okasakis ”Fun with binary heap trees”):

```
merge :: Ord a => SkewHeap a -> SkewHeap a -> SkewHeap a
merge Empty           r               = r
merge l               Empty           = l
merge (Node ll lx lr) (Node rl rx rr) =
  if lx <= rx then
    Node lr lx (merge ll (Node rl rx rr))
  else
    Node rr rx (merge rl (Node ll lx lr))
```

Analysera följande uttrycks tidskomplexitet: `merge l r`. Du kan anta att jämförelser (`<=`) tar konstant tid.

Svara med ett enkelt uttryck (inte en summa, rekursiv definition, eller dylikt). Onödigt oprecisa analyser kan underkännas. Använd inte variabeln n i ditt svar utan att definiera vad n står för.

Kortfattade lösningsförslag för tentamen i Datastrukturer (DAT036) från 2014-04-25

Nils Anders Danielsson

1. Proceduren utför konstant arbete för varje nod, och konstant arbete för varje kant, så tidskomplexiteten blir $\Theta(|V| + |E|)$.

2. Haskellösning:

```
deleteAll :: Int -> [Int] -> [Int]
deleteAll i []       = []
deleteAll i (j : js) = if i == j then js' else j : js'
  where js' = deleteAll i js
```

   Rekursiv Javalösning (varning för ”stack overflow”):

```
public void deleteAll(int i) {
    first = deleteAll(i, first);
}

private Node deleteAll(int i, Node n) {
    if (n == null) {
        return n;
    }

    n.next = deleteAll(i, n.next);

    if (n.contents == i) {
        return n.next;
    } else {
        return n;
    }
}
```

Iterativ Javalösning:

```
public void deleteAll(int i) {
    Node current  = first;
    Node previous = null;

    while (current != null) {

        if (current.contents == i) {
            if (previous == null) {
                // Specialfall för borttagande av
                // listans första element.
                first = current.next;
            } else {
                previous.next = current.next;
            }
        } else {
            previous = current;
        }

        current = current.next;
    }
}
```

Alla implementationerna utför konstant arbete per nod, och konstant övrigt arbete, och är därför linjära.

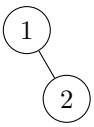
3. *För fyra:* Använd AVL-träd, och implementera `new`, `insert`, `delete` och `member` på vanligt sätt. Implementera `delete-min-max` genom att ta bort elementet längst till vänster (om det finns något) och sedan elementet längst till höger (om det finns något).

   För femma: Använd dessutom en hashtabell (med en tillräckligt bra hashfunktion). Använd hashtabellen för att implementera `member`. Då trädet uppdateras, uppdatera hashtabellen på motsvarande sätt; för `delete-min-max`, använd information från trädet för att avgöra vilket eller vilka element som ska tas bort.

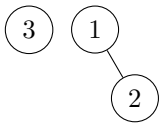
4. 1:

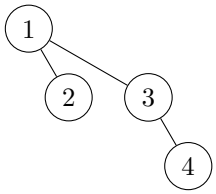
   2:

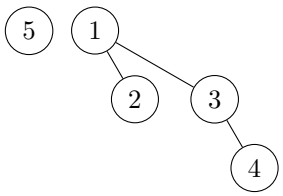
3:



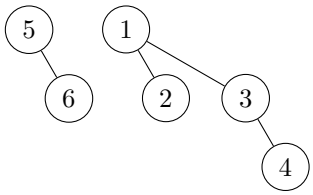
4:



5:



6:



5. (a) 4 stycken: $\{ 0, 1, 4 \}$, $\{ 2, 3 \}$, $\{ 5 \}$, $\{ 6 \}$.

   (b) Representera grafen på följande sätt:

   - Ett heltal `n`, antalet noder i grafen. (Låt oss för enkelhets skull anta att noderna är numrerade från 0 till $\texttt{n} - 1$.)
   - En array `component` av storlek `n`, innehållandes komponentnummer: för noder u och v ska vi ha $\texttt{component}[u] = \texttt{component}[v]$ omm u och v hör till samma komponent.
   - En grannmatris av storlek $\texttt{n} \times \texttt{n}$.

   Arrayen `component` gör det lätt att uppfylla krav i, och grannmatrisen gör det lätt att uppfylla krav ii.

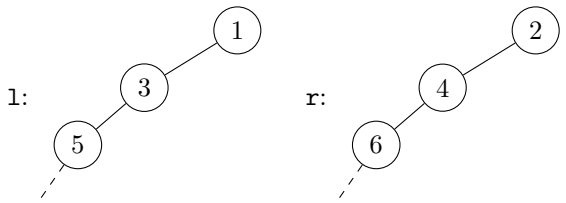
   För att lägga till en (ny) kant från u till v kan man använda följande (ickeoptimerade) algoritm:

   - Uppdatera grannmatrisen ($O(1)$).
   - Ersätt `component`-arrayen med en ny, beräknad med (en variant av) SCC-algoritmen som gicks igenom i kursen. Notera att i och med att en grannmatris används så blir tidskomplexiteten för djupet först-sökningarna och SCC-algoritmen $\Theta(\texttt{n}^2)$.

   Total tidskomplexitet: $\Theta(\texttt{n}^2)$.

6. Summan av de två trädens höjder minskar med minst ett i varje rekursivt `merge`-anrop. Operationen `merge` anropas alltså $O(h_{\texttt{l}} + h_{\texttt{r}})$ gånger, där h_t är höjden hos trädet t. Varje `merge`-anrop tar, bortsett från rekursiva `merge`-anrop, konstant tid, så tidskomplexiteten är $O(h_{\texttt{l}} + h_{\texttt{r}})$.

   Notera att för par av träd med följande struktur och nodinnehåll är tidskomplexiteten $\Theta(h_{\texttt{l}} + h_{\texttt{r}})$, givet att den vanliga ordningen för naturliga tal används:



   Tidskomplexiteten för den här sortens trädpar är också $\Theta(s_{\texttt{l}} + s_{\texttt{r}})$, där s_t är antalet noder hos trädet t. Trots detta kan man ge `merge l r` den *amorterade* tidskomplexiteten $O(\log s_{\texttt{l}} + \log s_{\texttt{r}})$; se t ex Okasakis ”Fun with binary heap trees”.