Tentamen
Datastrukturer (DAT037)

- Datum och tid för tentamen: 2016-01-09, 14:00–18:00.
- 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) 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.
- 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, 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 examinatorn 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++) {
       q.insert(t.deleteMin());
   }
   for (int i = 0; i < n; i++) {
       t.insert(q.deleteMin());
   }
   ```

   Använd kursens uniforma kostnadsmodell, och gör följande antaganden:

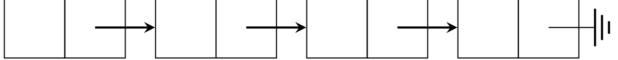
   - Att `n` är ett positivt heltal, och att typen `int` kan representera alla heltal.
   - Att `t` är ett AVL-träd som till att börja med innehåller `n` heltal, alla olika.
   - Att `q` är en leftistheap som till att börja med är tom.
   - Att den vanliga ordningen för heltal ($\ldots < -1 < 0 < 1 < 2 < \ldots$) används vid jämförelser.

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

2. (a) *För trea:* Implementera en metod/funktion `size` som beräknar storleken av (antalet element i) en enkellänkad lista, representerad av följande klass:

```
public class SinglyLinkedList<A> {
    private class Node {
        public A 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 int size() {
        // Din uppgift.
    }

    ...
}
```

Exempel: För följande lista (där endast `Node`-objekt och `next`-pekare ritas ut) ska svaret bli 4:

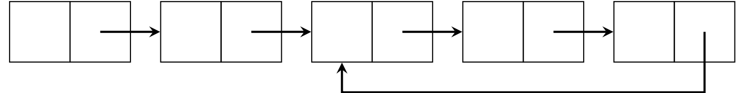


Endast detaljerad kod (inte nödvändigtvis Java) godkänns. Du får inte anropa andra procedurer e d om du inte implementerar dem själv som en del av din lösning, med undantag för enkla heltalsoperationer (som addition), samt följande datastrukturer: binärheapar, leftistheapar, hashtabeller, AVL-träd, prefixträd och skipplistor.

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

(b) *För fyra:* Som för deluppgift (a), men du ska också kunna hantera ”listor” med cykler. Dessutom ska du visa att din metod har tidskomplexiteten $O(n)$, där n är listans storlek.

Exempel: För följande ”lista” ska svaret bli 5:



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

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

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

   `someMemberStartsWith(`s`)` Avgör om det finns någon sträng i mängden som börjar med bitsträngen s (d v s om det finns någon sträng s' i mängden, och någon annan sträng s'', så att $s' = ss''$).

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

   ```
   Set strings = new Set();
   strings.insert("0000");
   strings.insert("1111");
   strings.insert("0101");
   return strings.someMemberStartsWith("") &&
          strings.someMemberStartsWith("00") &&
          strings.someMemberStartsWith("010") &&
          strings.someMemberStartsWith("1111") &&
          (! strings.someMemberStartsWith("10")) &&
          (! strings.someMemberStartsWith("00000"));
   ```

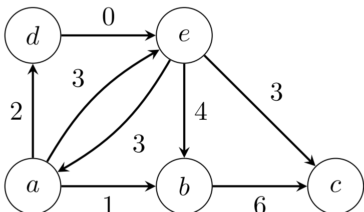
   Operationerna måste ha följande tidskomplexiteter (där n är antalet element i mängden, och ℓ är längden på strängargumentet som benämns s ovan):

   - *För trea:* `new`: $O(1)$, `insert`, `someMemberStartsWith`: $O(\ell n)$.
   - *För fyra:* `new`: $O(1)$, `insert`, `someMemberStartsWith`: $O(\ell \log n)$.
   - *För femma:* `new`: $O(1)$, `insert`, `someMemberStartsWith`: $O(\ell)$.

   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 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 Dijkstras algoritm för att beräkna kortaste vägen från startnoden (nod a) till alla noder i följande graf:



Svara med en lista av par (n, d), där n är en nodetikett och d längden på den kortaste vägen från startnoden till nod n. Varje nod i grafen ska förekomma exakt en gång i listan, och noderna ska placeras i listan i den ordning som algoritmen hittar de kortaste vägarna. Första paret ska alltså vara $(a, 0)$.

5. (a) *För trea:* Använd LSD-radixsortering för att sortera följande lista: $[357, 310, 824, 794, 150]$. Använd talbasen 10, d v s sortera med avseende på en decimal siffra i taget. Svara med en sekvens av listor: listan efter sortering med avseende på första siffran, listan efter sortering med avseende på andra siffran, o s v.

   (b) *För fyra:* När man implementerar LSD-radixsortering kan man använda olika underliggande sorteringsalgoritmer för att sortera med avseende på en given nyckel (t ex en given siffra, som i (a)-uppgiften). För vilka av följande underliggande sorteringsalgoritmer ger LSD-radixsortering alltid rätt svar?

   - Heapsort.
   - Mergesort.

   Anta att ”normala” implementationer av algoritmerna används. Om svaret för en viss algoritm är nej, ge då en lista för vilken algoritmen ger fel svar, och förklara varför algoritmen inte fungerar.

6. Beskriv en algoritm som uppfyller följande krav, och analysera algoritmens tidskomplexitet:

   - Indata: En oviktad, riktad, acyklisk graf G, samt två noder a, b i G.
   - Algoritmen ska avgöra om det finns någon nod c i G som kan nås från både a och b (via vägar av längd 0 eller längre), och som dessutom uppfyller kravet att alla andra noder som kan nås från både a och b kan nås från c.

   Algoritmen behöver inte beskrivas med detaljerad kod, men lösningen måste innehålla så mycket detaljer att tidskomplexiteten kan analyseras. För högsta betyg krävs att algoritmen är (någorlunda) effektiv; onödigt långsamma lösningar kan inte få högre betyg än fyra.

   Exempel:

Graf	Korrekt svar
a b	Nej
a b	Ja
a b	Nej

Graf	Korrekt svar
a b	Ja
a b	Ja

Lösningsförslag för tentamen i Datastrukturer (DAT037) från 2016-01-09

Nils Anders Danielsson

1. Träd- och köoperationerna har alla tidskomplexiteten $O(\log s)$, där s är antalet element i trädet/kön (notera att jämförelser tar konstant tid):

   - `t.deleteMin`: $O(\log s)$, där s är antalet element i `t`.
   - `t.insert`: $O(\log s)$, där s är antalet element i `t`.
   - `q.deleteMin`: $O(\log s)$, där s är antalet element i `q`.
   - `q.insert`: $O(\log s)$, där s är antalet element i `q`.

   Eftersom alla element är olika blir tidskomplexiteten

   $$2\left(O\left(\sum_{s=\mathtt{n}}^{1} \log s\right) + O\left(\sum_{s=1}^{\mathtt{n}-1} \log s\right)\right) + O(\mathtt{n}) = O(\log(\mathtt{n}!\,)) = O(\mathtt{n} \log \mathtt{n}),$$

   där termen $O(\mathtt{n})$ härrör från loopadministration: test av loopvillkor m m.

2. (a)
```
public int size() {
    int length = 0;
    Node here  = first;

    while (here != null) {
        length++;
        here = here.next;
    }

    return length;
}
```

(b) Låt oss använda en hashtabell för att hålla reda på de noder som vi redan har besökt:

```
public int size() {
    Set<Node> seen = new HashSet<>();
    Node here      = first;

    while (here != null && ! seen.contains(here)) {
        seen.add(here);
        here = here.next;
    }

    return seen.size();
}
```

Om vi antar att hashtabellsoperationerna tar amorterat konstant tid så utförs amorterat konstant arbete per listnod, och amorterat konstant övrigt arbete, och i så fall är algoritmen linjär i listans storlek ($\Theta(n)$, där n är storleken).

3. Använd ett prefixträd. En möjlighet är att använda följande datatyp:

```
-- Invariant: Ingen BitTrie får ha formen
-- Node False Empty Empty.
data BitTrie
  = Empty
  | Node Bool BitTrie BitTrie
```

$[\![_]\!]$ ger semantiken av en `BitTrie` som en mängd av bitsträngar:

$$
\begin{aligned}
[\![\texttt{Empty}]\!] &= \emptyset \\
[\![\texttt{Node}\ b\ \ l\ \ r]\!] &= \{\ \texttt{[]} \mid b = \texttt{True}\ \} \ \cup \\
&\quad \{\ \texttt{0 : } s \mid s \in [\![l]\!]\ \} \cup \\
&\quad \{\ \texttt{1 : } s \mid s \in [\![r]\!]\ \}
\end{aligned}
$$

Notera att invarianten medför att $[\![t]\!] = \emptyset$ om och endast om $t =$ `Empty`. (Det kan bevisas med induktion.)

Operationerna kan implementeras på följande sätt:

- `empty`: Skapa ett tomt träd (`Empty`). Tidskomplexitet: $O(1)$.
- `insert`: Sätt in strängen i trädet. Tidskomplexitet: $O(\ell)$. Notera att det är lätt att se till att insättningsalgoritmen bevarar invarianten ovan.

  (För ett exempel på en implementation, se uppgift 6 på tentan från december 2013, men byt ut `True` mot 0 och `False` mot 1.)
- `someMemberStartsWith`: Uppgiften är att avgöra om en given sträng s är ett prefix av någon sträng i trädet. Det kan man göra genom att

söka efter s. Om man stöter på det tomma trädet så är svaret nej, och annars är svaret ja:

```
isPrefix :: [Bit] -> BitTrie -> Bool
isPrefix _       Empty        = False
isPrefix []      _            = True
isPrefix (0 : s) (Node _ l _) = isPrefix s l
isPrefix (1 : s) (Node _ _ r) = isPrefix s r
```

Notera att den här funktionen som värst är linjär i bitsträngens längd (tidskomplexitet: $O(\ell)$).

Man kan bevisa att funktionen är korrekt,

$$\texttt{isPrefix}\ s\ t\ =\ \texttt{True} \quad \Leftrightarrow \quad \exists s' \in [\![t]\!].\ s \text{ är ett prefix av } s',$$

med strukturell induktion. Betrakta följande fyra uttömmande fall:

- $t = \texttt{Empty}$: Här ska man bevisa en ekvivalens mellan två osanna påståenden:

$$\begin{aligned}
&\texttt{isPrefix}\ s\ t\ =\ \texttt{True} && \Leftrightarrow \\
&\texttt{False}\ =\ \texttt{True} && \Leftrightarrow \\
&\exists s' \in \emptyset.\ s \text{ är ett prefix av } s'. && \Leftrightarrow \\
&\exists s' \in [\![t]\!].\ s \text{ är ett prefix av } s'.
\end{aligned}$$

- $s = \texttt{[]}$, $t \neq \texttt{Empty}$: Invarianten ger att $[\![t]\!] \neq \emptyset$, varför man får att

$$\begin{aligned}
&\texttt{isPrefix}\ s\ t\ =\ \texttt{True} && \Leftrightarrow \\
&\texttt{True}\ =\ \texttt{True} && \Leftrightarrow \\
&[\![t]\!] \neq \emptyset && \Leftrightarrow \\
&\exists s' \in [\![t]\!].\ \texttt{[]} \text{ är ett prefix av } s'.
\end{aligned}$$

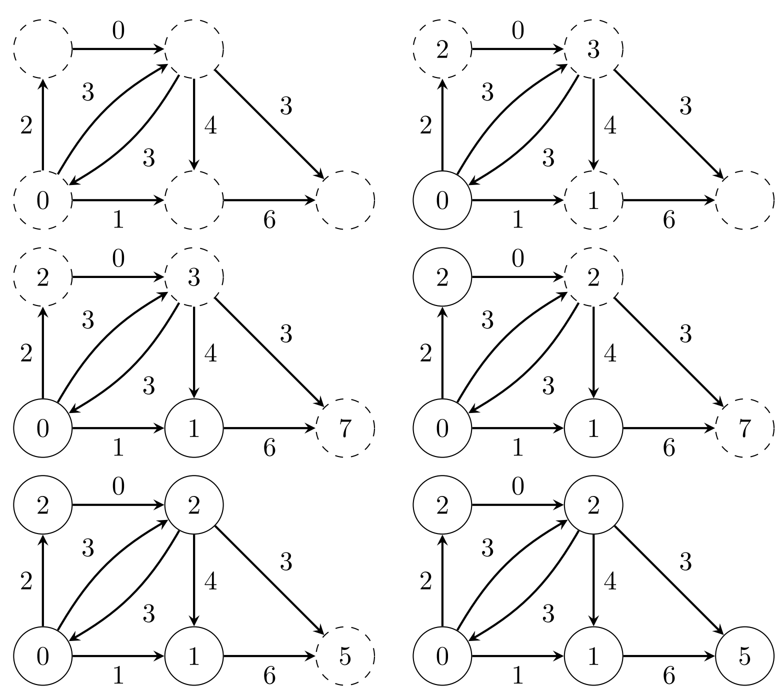
- $s = \texttt{0}\ \texttt{:}\ s'$, $t = \texttt{Node}\ \texttt{_}\ l\ \texttt{_}$: Induktionshypotesen ger att

$$\begin{aligned}
&\texttt{isPrefix}\ s\ t\ =\ \texttt{True} && \Leftrightarrow \\
&\texttt{isPrefix}\ s'\ l\ =\ \texttt{True} && \Leftrightarrow \\
&\exists s'' \in [\![l]\!].\ s' \text{ är ett prefix av } s'' && \Leftrightarrow \\
&\exists s'' \in [\![t]\!].\ \texttt{0}\ \texttt{:}\ s' \text{ är ett prefix av } s''.
\end{aligned}$$

- $s = \texttt{1}\ \texttt{:}\ s'$, $t = \texttt{Node}\ \texttt{_}\ \texttt{_}\ r$: Induktionshypotesen ger att

$$\begin{aligned}
&\texttt{isPrefix}\ s\ t\ =\ \texttt{True} && \Leftrightarrow \\
&\texttt{isPrefix}\ s'\ r\ =\ \texttt{True} && \Leftrightarrow \\
&\exists s'' \in [\![r]\!].\ s' \text{ är ett prefix av } s'' && \Leftrightarrow \\
&\exists s'' \in [\![t]\!].\ \texttt{1}\ \texttt{:}\ s' \text{ är ett prefix av } s''.
\end{aligned}$$

4. Algoritmens steg (”kända” noder har heldragna kanter, och nodetiketterna är de bästa kända avstånden):

   Vi får följande lista: $[(a, 0), (b, 1), (d, 2), (e, 2), (c, 5)]$.

5. (a)
      i. $[310, 150, 824, 794, 357]$.
      ii. $[310, 824, 150, 357, 794]$.
      iii. $[150, 310, 357, 794, 824]$.

   (b)
      - Mergesort fungerar, givet att en stabil implementation används.
      - Heapsort är vanligtvis inte stabil, i vilket fall algoritmen inte fungerar. Betrakta följande lista: $[10, 11]$. I första steget sorterar vi med avseende på sista siffran, och får då $[10, 11]$. I andra steget sorterar vi med avseende på första siffran. Vi utgår från arrayen | 10 | 11 |. Första steget i sorteringen är att bygga en maxheap med `build-heap`-algoritmen. Eftersom trädet som arrayen representerar redan uppfyller heapinvarianten så ligger alla element kvar (| 10 | 11 |). Nästa steg är att ta bort heapens översta element $r = 10$ (| | 11 |), sätta heapens sista element överst (| 11 | |), bubbla ned det översta elementet (| 11 | |), och sätta in r sist i arrayen (| 11 | 10 |). Den slutgiltiga listan blir $[11, 10]$, som inte är sorterad.

6. Låt oss anta att noderna är numrerade från 0 till $v-1$, och att grafen representeras av grannlistor: en array med storlek v, där position i innehåller en länkad lista med nod is direkta efterföljare.

   Notera att man kan ta fram en mängd innehållandes alla noder som kan nås från en nod u på följande sätt: utför en djupet först-sökning med början i u, och lägg varje nod som besöks i en från början tom mängd. Om mängden implementeras med en bitarray av storlek v så är tidskomplexiteten[1] $O(v+e)$ (där e är antalet kanter i grafen).

   Algoritmen:

   (a) Använd algoritmen ovan för att ta fram en mängd A med noderna som kan nås från a. Tidskomplexitet: $O(v+e)$.

   (b) Använd algoritmen ovan för att ta fram en mängd B med noderna som kan nås från b. Tidskomplexitet: $O(v+e)$.

   (c) Beräkna snittet av de två mängderna. Resultatet är en mängd $C = A \cap B$ som innehåller exakt de noder som kan nås från både a och b. Avgör om C är tom, i vilket fall algoritmens svar är nej. Tidskomplexitet: $\Theta(v)$.

   (d) Sortera grafen topologiskt, och notera vilken nod d i C som hamnar tidigast i den topologiska ordningen. Tidskomplexitet: $\Theta(v+e)$.

   (e) Det återstår att avgöra om någon nod c i C har egenskapen att alla andra noder i C kan nås från c. Definitionen av topologisk ordning ger att den enda noden som kan ha den här egenskapen är d. Använd algoritmen ovan för att ta fram en mängd D med noderna som kan nås från d, och avgör om C är en delmängd av D. Om så är fallet så är algoritmens resultat ja, och annars är resultatet nej. Tidskomplexitet: $O(v+e)$.

   Sammanlagd tidskomplexitet: $\Theta(v+e)$. Algoritmen är alltså linjär i grafens storlek, vilket kanske kan anses vara effektivt. Notera dock att algoritmen besöker noder som inte kan nås från a eller b, och plats allokeras dessutom för sådana noder i mängderna. Med lite andra val av datastrukturer och algoritmer (t ex kan hashtabeller användas för att implementera mängder) kan man (givet tillräckligt bra hashfunktioner) konstruera en algoritm med tidskomplexitet $\Theta(v'+e')$, där v' är antalet noder som kan nås från a och/eller b, och e' är totala antalet kanter som lämnar dessa noder.

[1] Med en effektiv implementation. Liknande brasklappar gäller för flera andra påståenden i det här lösningsförslaget.