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:

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

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

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

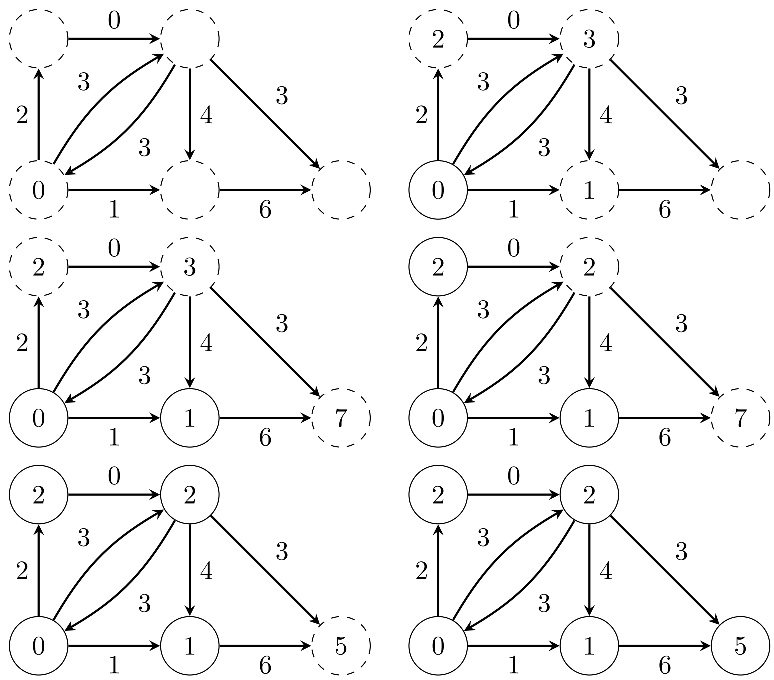
- $s = \texttt{0 : } s'$, $t = \texttt{Node \_ } l \texttt{ \_}$: Induktionshypotesen ger att

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

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

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

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 $\boxed{10}\boxed{11}$. 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 ($\boxed{10}\boxed{11}$). Nästa steg är att ta bort heapens översta element $r = 10$ ($\boxed{}\boxed{11}$), sätta heapens sista element överst ($\boxed{11}\boxed{}$), bubbla ned det översta elementet ($\boxed{11}\boxed{}$), och sätta in r sist i arrayen ($\boxed{11}\boxed{10}$). 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.