Kortfattade lösningsförslag för tentamen i Datastrukturer (DAT036) från 2012-08-24

Nils Anders Danielsson

1. Den yttre loopen körs exakt `m` gånger, för varje varv i den yttre loopen körs den inre loopen exakt `n` gånger, och övrig kod tar konstant tid varje gång den körs, så tidskomplexiteten är $\Theta(\texttt{mn})$.

2. Man kan representera en bijektion $f : S \to T$ med hjälp av två "maps", en som svarar mot f och en som svarar mot f^{-1}. En möjlig implementation (i Java):

```
import java.util.*;

public class Bijection<S, T> {

  private Map<S, T> to;
  private Map<T, S> from;

  public Bijection() {
    to   = new TreeMap<S, T>();
    from = new TreeMap<T, S>();
  }

  public void insert(S s, T t) {
    if (to.containsKey(s) || from.containsKey(t)) {
      throw new IllegalArgumentException();
    }
    to.put(s, t);
    from.put(t, s);
  }

  public T target(S s) {
    return to.get(s);
  }

  public S source(T t) {
    return from.get(t);
  }
}
```

Konstrueraren `Bijection` tar konstant tid, och om vi antar att `S` och `T` båda är `Integer` (och jämförelser därför tar konstant tid) så är övriga operationer logaritmiska i antalet bindningar.

Man kan troligtvis förbättra tidskomplexiteten genom att använda `HashMaps` istället för `TreeMaps`.

3. Enkel lösning: Skapa ett tomt träd, gå igenom arrayen (med en iterator) och sätt in varje element i trädet.

   Mer effektiv lösning (åtminstone asymptotiskt): Utnyttja att arrayen är sorterad och bygg upp trädet manuellt, som i följande Javakod:

```
// Skapar ett AVL-träd från en array.
//
// Krav: Arrayen måste vara sorterad.
public AVLTree(ArrayList<A> ns) {
  root = fromArray(ns, 0, ns.size() - 1);
}

// Skapar ett AVL-träd från en array. Endast noderna på position
// {from, from + 1, …, to} inkluderas.
//
// Krav: Arrayen måste vara sorterad, och om from <= to måste
// följande olikheter vara uppfyllda:
//   0 <= from <= to < ns.size().
private TreeNode fromArray(ArrayList<A> ns, int from, int to) {
  if (from <= to) {
    // Om vi låter ett av mittenelementen vara rot så kommer
    // vänster och höger delträd att uppfylla
    // AVL-trädsinvarianten (eftersom de är lika stora, ±1,
    // och har minimal höjd).
    //
    // Notera att om from <= to så har vi
    //   from <= middle <= to,
    // så vi indexerar inte utanför ns, vi uppfyller
    // fromArrays krav, och fromArray terminerar.
    int middle = from + (to - from) / 2;
    return new TreeNode(fromArray(ns, from, middle - 1),
                        ns.get(middle),
                        fromArray(ns, middle + 1, to));
  } else {
    return null;
  }
}
```

   Om man vill vara noggrann kan man bevisa att träden har minimal höjd med hjälp av induktion.

   Koden ovan utför konstant arbete per arrayelement, och är alltså linjär i antalet element.

4. Om variablerna representeras som konsekutiva heltal $0, 1, 2, \ldots, n-1$ så kan V representeras med ett enda heltal: n.

   Mängden O kan representeras av en ”set”-datastruktur innehållandes par av variabler (heltal), där paret (i, j) står för $i < j$.

   Algoritmens resultat kan representeras på följande sätt: Nej kan representeras av `null`, och en korrekt tilldelning kan representeras av en array av längd n som innehåller variabel is tilldelning på position i (om vi räknar från 0).

   Algoritmen (med V och O som indata):

   (a) Skapa en riktad graf G med en nod för varje variabel i V, och en kant från i till j om $i < j$ finns i O.

   (b) Försök sortera grafen topologiskt.

   - Om grafen kan sorteras topologiskt så tilldelas varje nod i ett topologiskt nummer t_i (givet att en lämplig algoritm används). De här numren har egenskapen att $t_i < t_j$ för noder i, j med $i < j \in O$. Ge $[t_0, t_1, \ldots, t_{n-1}]$ som svar.
   - Om grafen inte kan sorteras topologiskt så finns det en cykel i grafen, och alltså finns det ingen korrekt variabeltilldelning. Ge `null` som svar.

   Värstafallstidskomplexitet (om grafen representeras av en array, av längd $n = |V|$, som innehåller grannlistor):

   - Skapa graf: $\Theta(|V| + |O|)$.
   - Topologisk sortering: $O(|V| + |O|)$.
   - Skapa array: Om den topologiska sorteringen implementeras på lämpligt sätt så får vi arrayen som svar från sorteringsalgoritmen.

   Totalt: $\Theta(|V| + |O|)$, vilket nog får klassas som effektivt.[1]

[1] Notera dock att om O är tom så är algoritmen exponentiellt långsam, uttryckt i storleken av dess indata (om storleken av V är $\Theta(\log |V|)$).

5. Implementation av `reverse` (i Java):

```
public void reverse() {
  ListNode here = head;

  // Gå igenom listan och byt plats på next- och
  // prev-pekarna.
  while (here != null) {
    ListNode next = here.next;
    here.next = here.prev;
    here.prev = next;

    // Om vi är på sista noden, uppdatera head-pekaren.
    if (next == null) {
      head = here;
    }

    here = next;
  }
}
```

   Algoritmen utför konstant arbete för varje nod, och är alltså linjär i listans längd.

6. Låt oss använda potentialmetoden, där potentialen är en positiv konstant k gånger bakstackens längd: $\Psi(f, b) = k|b|$. Den här potentialfunktionen är OK, eftersom ursprungstillståndet (den tomma kön) har den minsta möjliga potentialen:

$$\forall f, b. \quad \Psi([\,], [\,]) = 0 \leq k|b| = \Psi(f, b).$$

   Det återstår att analysera de två operationerna:

   `enqueue(a)` Koden tar konstant (faktisk) tid, och potentialen ökar med k, så den amorterade tidskomplexiteten är

$$O(1) + k = O(1).$$

   `dequeue` Anta att kön består av framstacken f och bakstacken b. Om f är tom så är den amorterade tidskomplexiteten

$$O(|b|) + 0 - k|b|,$$

   vilket är $O(1)$ om k kan väljas tillräckligt stor. Om f inte är tom så ändras inte potentialen, varför den amorterade tidskomplexiteten är $O(1)$.

   Eftersom k kan väljas tillräckligt stor kan vi dra slutsatsen att båda operationerna tar amorterat konstant tid.