Kortfattade lösningsförslag för dugga i Datastrukturer (DAT036) från 2012-11-21

Nils Anders Danielsson

3. Använd en hashtabell för att hålla reda på antalet förekomster av varje tal:

   ```
   occurrences = new hash table
   ```

   Beräkna antalet förekomster av elementen i första listan:

   ```
   for i in the first list do
     n = occurrences.get(i)
     if n == null then n = 0
     occurrences.set(i, n + 1)
   ```

   Subtrahera antalet förekomster av elementen i andra listan, och ge `false` som svar om något element i den andra listan inte finns i den första:

   ```
   for i in the second list do
     n = occurrences.get(i)
     if n == null then
       return false
     else
       occurrences.set(i, n - 1)
   ```

   Kontrollera om antalet förekomster i de två listorna stämmer överens:

   ```
   for i in the first list do
     if not (occurrences.get(i) == 0) then
       return false

   return true
   ```

   Antag att listorna (eller den längsta listan) har längden n. Koden ovan anropar `set` och `get` $O(n)$ gånger, och utför i övrigt linjärt arbete. Om vi antar att vår hashfunktion är ”tillräckligt” bra, och dessutom kan beräknas på konstant tid, så kan vi dra slutsatsen att algoritmens tidskomplexitet är $O(n)$.

1. Några observationer:

   - Kön är tillräckligt stor, så `pq.delete-min` kommer inte att misslyckas.
   - Raden

     ```
     xs.add-first(pq.delete-min());
     ```

     kommer att köras `n` gånger.
   - Insättning *först* i en dynamisk array av storlek s har tidskomplexiteten $\Theta(s)$.

   Den totala tidskomplexiteten är således

   $$O\left(\sum_{i=0}^{\mathtt{n}-1}\left(\log(\mathtt{n}^3 - i) + i\right)\right) =$$
   $$O(\mathtt{n}\log\mathtt{n} + \mathtt{n}^2) =$$
   $$O(\mathtt{n}^2).$$

2. Implementation av algoritmen (i Java):

   ```
   // Konverterar ett TreeWithout-träd till ett TreeWith-träd
   // (utan att förstöra TreeWithout-trädet).
   public TreeWith(TreeWithout<A> t) {
       root = addParents(t.root, null);
   }

   // Konverterar en TreeWithout-nod till en TreeWith-nod.
   // Använder parent som den nya nodens förälder.
   private TreeNode addParents
       (TreeWithout<A>.TreeNode without, TreeNode parent) {
       if (without == null) return null;

       TreeNode with = new TreeNode(without.contents,
                                    null, null, parent);

       with.left  = addParents(without.left,  with);
       with.right = addParents(without.right, with);

       return with;
   }
   ```

   Algoritmen utför konstant arbete för varje nod, och är alltså linjär i trädets storlek.