Delvis kortfattade lösningsförslag för tentamen i Datastrukturer (DAT036/DAT037/DIT960) från 2016-04-07

Nils Anders Danielsson

1. Notera först att alla `dequeue`-operationerna kommer att lyckas.

   Insättning först i `xs` har tidskomplexiteten $\Theta(\ell)$, där ℓ är antalet element i arrayen. Borttagning av det första elementet i `q` har tidskomplexiteten $\Theta(1)$ (möjligtvis amorterat, beroende på hur den cirkulära arrayen implementerats). Den totala tidskomplexiteten blir

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

2. Om trädet inte är alltför djupt så behöver vi kanske inte oroa oss för ”stack overflow”, och i så fall kan en rekursiv implementation vara lämplig:

   ```
   private boolean parentsAreCorrect() {
       return parentsAreCorrect(root, null);
   }

   private boolean parentsAreCorrect(Node n, Node parent) {
       return n == null
           || (n.parent == parent
               && parentsAreCorrect(n.left, n)
               && parentsAreCorrect(n.right, n));
   }
   ```

   Eftersom algoritmen utför $O(1)$ arbete per trädnod, och $O(1)$ övrigt arbete, så är algoritmen linjär i trädets storlek.

3. Använd en binär heap och en kö (en dynamisk cirkulär array), och sätt in varje värde antingen i heapen eller i kön. Invariant: Kön är sorterad. Implementation:

   - `empty`: Skapa en tom heap och en tom kö.
   - `insert`(v): Om kön är tom, eller om köns sista element $v' \leq v$, sätt in v sist i kön (som fortsätter vara sorterad). Annars, sätt in v i heapen.

- `deleteMin`: Om kön är icketom och antingen heapen är tom, eller köns första element är mindre än eller lika med heapens minsta element, ta bort och ge tillbaka köns första element. (Notera att kön fortsätter vara sorterad.) Annars, ta bort och ge tillbaka heapens första element. (Notera att prioritetskön antas vara icketom, så kön och heapen kan inte båda vara tomma.)

Om värdena sätts in i sorterad ordning så kommer inget element sättas in i heapen, och tidskomplexiteten blir i så fall $O(1)$ (amorterat) för varje operation. I det generella fallet blir tidskomplexiteten $O(\log \ell)$ (amorterat) för både `insert` och `deleteMin`.

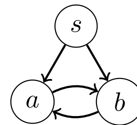
4. 1, 2, 7, 3, 4, 8, 5, 6.

5. Tidskomplexiteten för `delete k t` är, i värsta fallet, linjär i trädets höjd, och eftersom trädet inte behöver vara balanserat så är höjden i värsta fallet linjär i antalet noder (d v s nycklar) i trädet. Svaret blir $O(n)$, där n är antalet nycklar i trädet.

   Notera att svaret $O(v)$, där v är antalet värden i trädet, är fel. Det är möjligt att skapa träd med godtyckligt många nycklar, där varje nyckel svarar mot en tom lista.

6. (a) Alla korrekta svar:
   - b, e, a, d, f, c, g.
   - b, e, a, f, d, c, g.
   - b, e, f, a, d, c, g.
   - e, b, a, d, f, c, g.
   - e, b, a, f, d, c, g.
   - e, b, f, a, d, c, g.
   - e, f, b, a, d, c, g.

   (b) Algoritmen fungerar om bredden först-sökning används, men inte om djupet först-sökning används. Motexempel:



   Anta att djupet först-sökning används. Om sökningen besöker a innan b, så kommer det beräknade kortaste avståndet till b att bli 2 istället för 1. Om sökningen istället besöker b innan a så kommer det beräknade kortaste avståndet till a att bli 2 istället för 1.