Delvis kortfattade lösningsförslag för dugga i Datastrukturer (DAT036) från 2013-11-20

Nils Anders Danielsson

1. Notera först att kön är tillräckligt stor, så `q.delete-min()` kommer alltid att lyckas. Värstafallstidskomplexiteten är

$$O\left(\sum_{k=\mathtt{n}^2}^{1}(1+\log k)\right) = O(\mathtt{n}^2 \log \mathtt{n}).$$

   Om man känner till att jämförelsebaserad sortering av m element i värsta fallet kräver $\Omega(m \log m)$ jämförelser så kan man göra svaret mer precist. Notera att sekvensen av `delete-min`-operationer implicit ger oss en sorterad lista av prioriteter med $\mathtt{n}^2$ element. Tidskomplexiteten för att först bygga heapen och sedan utföra `delete-min`-operationerna är alltså $\Omega(\mathtt{n}^2 \log \mathtt{n})$. Man kan bygga en heap på linjär tid (i det här fallet $O(\mathtt{n}^2)$) med hjälp av `build-heap`, så tidskomplexiteten för `delete-min`-operationerna måste i värsta fallet vara $\Omega(\mathtt{n}^2 \log \mathtt{n})$.

   I och med att koden både har värstafallstidskomplexiteten $O(\mathtt{n}^2 \log \mathtt{n})$ och $\Omega(\mathtt{n}^2 \log \mathtt{n})$ så kan vi svara precist: $\Theta(\mathtt{n}^2 \log \mathtt{n})$.

2. Javaliknande kod:

```
// Sätter in a först i kön. Om kön är full lämnas den
// oförändrad. Resultatet är true om elementet sattes in,
// och annars false. Tidskomplexitet: O(1).
public boolean push(A a) {
    if (size == queue.length) {
        return false;
    }

    size++;
    if (front <= 0) {
        front = queue.length;
    }
    front--;
    queue[front] = a;

    return true;
}
```

3. Om man använder två prioritetsköer, en för värden vars prioritet är mindre än P, och en för värden med högre prioritet, så kan man implementera `delete-small` på konstant tid genom att byta ut den ena kön mot en tom kö. Om de underliggande prioritetsköerna är tillräckligt effektiva (vilket är fallet för t ex binomialheapar) så uppfylls även tidskomplexitetskraven för övriga operationer.