Dugga
Datastrukturer (DAT036)

- Duggans datum: 2013-11-20.
- Författare: Nils Anders Danielsson.
- För att en uppgift ska räknas som ”löst” så måste en i princip helt korrekt lösning lämnas in. Enstaka mindre allvarliga misstag kan *eventuellt* godkännas. Notera att duggan kan komma att rättas ”hårdare” än tentorna.
- Lämna inte in lösningar för flera uppgifter på samma blad.
- Lösningar kan underkännas om de är svårlästa, ostrukturerade eller dåligt motiverade. Pseudokod får gärna användas, men den får inte utelämna för många detaljer.
- Om inget annat anges så kan du använda kursens uniforma kostnadsmodell när du analyserar tidskomplexitet (så länge resultaten inte blir uppenbart orimliga).
- Om inget annat anges behöver du inte förklara standarddatastrukturer och -algoritmer från kursen, men däremot motivera deras användning.

1. Analysera nedanstående kods värstafallstidskomplexitet, uttryckt i `n`:

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        m.insert(i + j, q.delete-min());
    }
}
```

   Använd kursens uniforma kostnadsmodell, och gör följande antaganden:

   - Att `n` är ett ickenegativt heltal, och att typen `int` kan representera alla heltal.
   - Att `m` är en hashtabell som till att börja med är tom, och att en $O(1)$ perfekt hashfunktion används.
   - Att `q` är en binär heap som till att börja med har storlek $\texttt{n}^2$, och att tidskomplexiteten för att jämföra två prioriteter är $O(1)$.

   Svara med ett enkelt uttryck (inte en summa, rekursiv definition, eller dylikt). Onödigt oprecisa analyser kan underkännas; använd gärna Θ-notation.

2. Anta att vi håller på att implementera en klass för cirkulära arrayer, med följande tillståndsvariabler:

```
A[] queue; // Arrayen. Invariant: queue.length > 0.
int size;  // Köns storlek.
           // Invariant: 0 <= size <= queue.length.
int front; // Invariant: 0 <= front <= queue.length.
           // Om size > 0 pekar front på köns första
           // element. Om size > 1 pekar
           // (front + 1) % queue.length på köns andra
           // element. Och så vidare.
```

   Din uppgift är att implementera en metod som sätter in ett element av typ `A` *först* i kön. Insättning måste ta konstant tid. Du kan anta att kön inte är full.

   Endast detaljerad kod (inte nödvändigtvis Java) godkänns. Du får inte anropa några andra metoder/procedurer, om du inte implementerar dem själv.

   Tips: Testa din kod, så kanske du undviker onödiga fel.

3. Uppgiften är att konstruera en datastruktur för en prioritetskö-ADT med följande operationer:

   `new Priority-queue(`P`)` eller `empty(`P`)` Konstruerar en tom kö. Argumentet P används av `delete-small` nedan. Notera att varje kös P-värde är konstant: när kön väl har konstruerats kan P inte ändras.

   `insert(`v`,`p`)` Sätter in värdet v, med tillhörande prioritet p.

   `delete-min()` Tar bort och ger tillbaka värdet med lägst prioritet (eller, om det finns flera värden med lägst prioritet, ett av dem). *Precondition*: Kön får inte vara tom.

   `delete-small()` Tar bort alla värden vars prioritet är mindre än P.

   Exempel: Följande kod, skriven i ett Javaliknande språk, ska ge `true` som svar:

   ```
   Priority-queue q = new Priority-queue(5);
   q.insert('a', 0);
   q.insert('b', 3);
   q.insert('c', 4);
   q.insert('d', 5);
   boolean b = q.delete-min() == 'a';
   q.delete-small();
   b = b && (q.delete-min() == 'd');
   return b;
   ```

   Du måste visa att operationerna uppfyller följande tidskomplexitetskrav (där n är köns storlek): `new`, `delete-small`: $O(1)$, `insert`, `delete-min`: $O(\log n)$. (Du kan anta att prioriteterna är heltal.)

   Implementationen av datastrukturen behöver inte beskrivas med detaljerad kod, men lösningen måste innehålla så mycket detaljer att tidskomplexiteten kan analyseras.

   Tips: Konstruera om möjligt inte datastrukturen från grunden, bygg den hellre m h a standarddatastrukturer. Testa dina algoritmer, så kanske du undviker onödiga fel.

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.