Dugga
Datastrukturer (DAT036/DAT037)

- Duggans datum och tid: 2015-11-27, 8:00–9:30.
- Författare: Nils Anders Danielsson.
- Godkända hjälpmedel: Ett A4-blad med handskrivna anteckningar.
- 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.
- Skriv namn och personnummer på varje blad.
- Lösningar kan underkännas om de är svårlästa, ostrukturerade eller dåligt motiverade. Om inget annat anges får pseudokod 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 (sådant som har gåtts igenom på föreläsningarna), men däremot motivera deras användning.

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

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

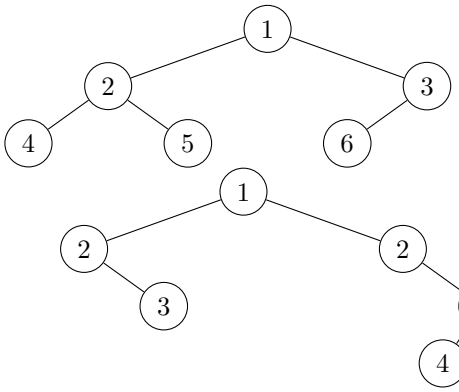
   Använd kursens uniforma kostnadsmodell, och gör följande antaganden:

   - Att `n` är ett positivt heltal, och att typen `int` kan representera alla heltal.
   - Att `q` är en leftistheap som till att börja med innehåller $\texttt{n}^2$ heltal.
   - Att den vanliga ordningen för heltal ($... < -1 < 0 < 1 < 2 < ...$) används vid jämförelser.

   Svara med ett enkelt uttryck (inte en summa, rekursiv definition, eller dylikt). Onödigt oprecisa analyser kan underkännas.

2. Implementera en metod/funktion som, givet ett binärt träd, ger en lista med trädets noder i *nivåordning*: Först roten, sedan alla rotens barn (från vänster till höger), sedan alla rotens barnbarn (från vänster till höger), och så vidare.

   Exempel:

Träd	Lista
	$[1, 2, 3, 4, 5, 6]$
	$[1, 2, 2, 3, 3, 4]$
Tomt	$[\,]$

Du måste representera binära träd på ett av följande sätt:

- Med följande Haskelldatatyp:

```
data Tree a = Empty
            | Node (Tree a) a (Tree a)
```

- Med följande Javaklass:

```
public class Tree<A> {
    // Trädnoder; null representerar tomma träd.
    private class Node {
        A    contents;  // Innehåll.
        Node left;      // Vänstra barnet.
        Node right;     // Högra barnet.
    }

    // Roten.
    private Node root;

    // Din uppgift.
    public List<A> levelOrder() {
        ...
    }

    ...
}
```

Endast detaljerad kod (inte nödvändigtvis Haskell/Java) godkänns. Du får inte anropa några andra metoder/procedurer/funktioner, om du inte implementerar dem själv, med följande undantag: du får använda datastrukturer för listor, stackar och köer.

Metoden/funktionen måste vara linjär i trädets storlek ($O(n)$, där n är storleken). Visa att så är fallet.

Tips: Att gå igenom trädets noder i nivåordning motsvarar att utföra en bredden först-sökning i en graf. 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()` eller `empty()` Konstruerar en tom kö.

   `insert(`v`,`p`)` Sätter in värdet v, med tillhörande prioritet p, i kön. *Precondition*: Värdet v får inte förekomma i kön.

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

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

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

   Du måste visa att operationerna uppfyller följande tidskomplexitetskrav (där n är antalet värden i kön): `new`: $O(1)$, `insert`, `delete-min`, `delete-max`: $O(\log n)$. (Du kan anta att värden och prioriteter är heltal.)

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

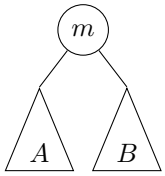
   Tips: Konstruera om möjligt inte datastrukturen från grunden, bygg den hellre m h a standarddatastrukturer. Testa dina algoritmer, så kanske du undviker onödiga fel.

Kortfattade lösningsförslag för dugga i Datastrukturer (DAT036/DAT037) från 2015-11-27

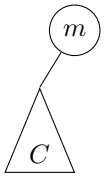
Nils Anders Danielsson

1. Notera att `q` hela tiden innehåller $\mathtt{n}^2$ eller $\mathtt{n}^2 - 1$ heltal. Tidskomplexiteten för varje leftistheapoperation är $O(\log(\mathtt{n}^2)) = O(\log \mathtt{n})$, så den totala tidskomplexiteten blir $O(\mathtt{n}^2 \log \mathtt{n})$.

   Vi kan dock göra en mer noggrann analys. Notera att `q` till att börja med har följande form:

   Om vi kör `delete-min` (tidskomplexitet: $O(\log \mathtt{n})$) får vi en ny kö genom att slå ihop A och B, låt oss kalla den C. (Den här kön är eventuellt tom.) Om vi sedan sätter in m får vi, på konstant tid, en ny kö D med följande form:

   Om vi kör `delete-min` igen får vi på konstant tid C. Efter första anropet till `delete-min` så kommer `q` alltså att växla mellan C och D, och varje köoperation (efter den första) tar konstant tid. Den totala tidskomplexiteten blir alltså $\Theta(\mathtt{n}^2)$.

2. *Haskellösning:* Anta att vi har tillgång till en FIFO-ködatastruktur med följande operationer, som alla tar amorterat konstant tid:

```
empty   :: Queue a
enqueue :: a -> Queue a -> Queue a
dequeue :: Queue a -> Maybe (a, Queue a)
```

I så fall kan uppgiften lösas på följande sätt:

```
-- Noderna i träden, i nivåordning: först alla rötter (med
-- början på första trädet i kön), sedan rötternas barn,
-- och så vidare.

levelOrder' :: Queue (Tree a) -> [a]
levelOrder' q = case dequeue q of
  Nothing     -> []
  Just (t, q) -> case t of
    Empty      -> levelOrder' q
    Node l x r ->
      x : levelOrder' (enqueue r (enqueue l q))

-- Noderna i trädet, i nivåordning.

levelOrder :: Tree a -> [a]
levelOrder t = levelOrder' (enqueue t empty)
```

Algoritmen utför amorterat konstant arbete per trädnod, och amorterat konstant övrigt arbete, så implementationen är linjär i trädets storlek.

Javalösning:

```
// Noderna i trädet, i nivåordning.
public List<A> levelOrder() {
    // En kö som kan hantera null-värden.
    Deque<Node> q  = new LinkedList<>();
    List<A>     ns = new ArrayList<>();

    q.addLast(root);

    while (! q.isEmpty()) {
        Node n = q.pollFirst();

        if (n != null) {
            ns.add(n.contents);
            q.addLast(n.left);
            q.addLast(n.right);
        }
    }

    return ns;
}
```

3. Tidskomplexiteterna som anges nedan är (ibland) amorterade.

   Använd två binära heapar, en minheap och en maxheap. I båda fallen ska heapen kombineras med en hashtabell som mappar värden till heappo-

sitioner, och tabellen ska uppdateras varje gång heapen ändras. Låt oss anta att hashfunktionerna är tillräckligt bra, så att de relevanta hashtabelloperationernas tidskomplexiteter är $O(1)$.

Prioritetsköoperationerna kan då implementeras på följande sätt:

- `empty`: Skapa två tomma heapar ($O(1)$).
- `insert`: Sätt in värdet och prioriteten i båda heaparna ($O(\log n)$).
- `delete-min`: Ta bort ett av värdena med lägst prioritet från minheapen ($O(\log n)$). Låt oss kalla värdet v. Använd ena hashtabellen för att hitta positionen för v i maxheapen ($O(1)$), och ta bort v: bubbla upp ($O(\log n)$), kör `delete-max` ($O(\log n)$). Total tidskomplexitet: $O(\log n)$.
- `delete-max`: På motsvarande sätt.

Alternativ lösning: Använd ett AVL-träd som mappar prioriteter till icketomma länkade listor med värden.