Dugga
Datastrukturer (DAT036)

- Duggans datum: 2012-11-21.
- 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 tidskomplexitet, uttryckt i `n`:

   ```
   for(int i = 0; i < n; i++) {
     xs.add-first(pq.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 `pq` är en binär min-heap som till att börja med innehåller n^3 element.
   - Att `xs` är en till att börja med tom lista, implementerad som en dynamisk array.
   - Att `add-first` lägger till elementet först i listan.

   Onödigt oprecisa analyser kan underkännas.

2. Implementera en algoritm som, givet ett binärt träd utan föräldrapekare,[1] skapar ett motsvarande träd med föräldrapekare. Det nya trädet ska ha samma struktur och innehåll som det gamla (bortsett från föräldrapekarna).

   Du kan använda följande trädklasser:

   ```
   // Binära träd utan föräldrapekare.
   public class TreeWithout<A> {

       // Trädnoder; null representerar tomma träd.
       public class TreeNode {
           A        contents;  // Innehåll.
           TreeNode left;      // Vänstra barnet.
           TreeNode right;     // Högra barnet.
       }

       // Roten.
       public TreeNode root;
   }
   ```

[1] Pekare från en nod till dess förälder.

```
// Binära träd med föräldrapekare.
public class TreeWith<A> {

    // Trädnoder; null representerar tomma träd.
    private class TreeNode {
        A        contents;  // Innehåll.
        TreeNode left;      // Vänstra barnet.
        TreeNode right;     // Högra barnet.
        TreeNode parent;    // Föräldern; null för roten.

        // Skapar en trädnod.
        TreeNode(A contents,
                 TreeNode left, TreeNode right,
                 TreeNode parent) {
            this.contents = contents;
            this.left     = left;
            this.right    = right;
            this.parent   = parent;
        }
    }

    // Roten.
    private TreeNode root;

    // Din uppgift.
    public TreeWith(TreeWithout<A> t) {
        ...
    }
}
```

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

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

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

3. *En variant av en uppgift från en av Peter Dybjers tidigare tentor.*

   Beskriv en *linjär* algoritm som avgör om två listor innehållandes heltal är permutationer av varandra. Analysera algoritmens tidskomplexitet.

   Två listor är permutationer av varandra om de innehåller samma element, och samma antal av varje element. Elementen behöver inte förekomma i samma ordning.

   Exempel:

Lista 1	Lista 2	Resultat
$[0, 1, 2, 3]$	$[3, 1, 0, 2]$	Sant
$[0, 1, 2, 3]$	$[3]$	Falskt
$[0, 1]$	$[0, 1, 1]$	Falskt
$[1, 0, 1]$	$[0, 1, 1]$	Sant

   Tips: Testa din algoritm, så kanske du undviker onödiga fel. Använd gärna standarddatastrukturer och/eller -algoritmer från kursen.