NB: Kun et forslag, mange andre varianter er korrekte.

# Oppgave 1. Stabler og Køer (30%)

### 1.1

```
public class myStabel extends Stabel {
  public boolean eqQueue (Queue Q) {
    if (this.size() != Q.size()) return false;
    while (!this.isEmpty()) {
      if (this.top() != Q.front()) return false;
      this.pop();
      Q.dequeue();
    }
    return true;
  }
}
```

### 1.2

```
public class myStabel extends Stabel {
  public boolean eqQueue (Queue Q) {
    if (this.size() != Q.size()) return false;
    Stack tempStack = new Stabel();
    while (!this.isEmpty() && this.top() == Q.front()) {
      tempStack.push(this.pop());
      Q.enqueue(Q.dequeue());
    }
    if (this.isEmpty()) {
      while (!tempStack.isEmpty()) this.push(tempStack.pop());
      return true;
    }
    else {
      for (int i=0; i<this.size(); i++) Q.enqueue(Q.dequeue());
      while (!tempStack.isEmpty()) this.push(tempStack.pop());
      return false;
    }
  }

}
```

**1.3**

```
public class myStabel extends Stabel {
  public boolean eqQueue (Queue Q) {
    if (this.size() != Q.size()) return false;
    return eqQ(Q);
  }
  private boolean eqQ (Queue Q) {
    if (this.isEmpty()) return true;
    if (this.top() != Q.front()) return false;
    Object temp = this.pop();
    Q.dequeue();
    return eqQ(Q);
    this.push(temp);
  }
}
```

# Oppgave 2. Binære søketrær og heap (15%)

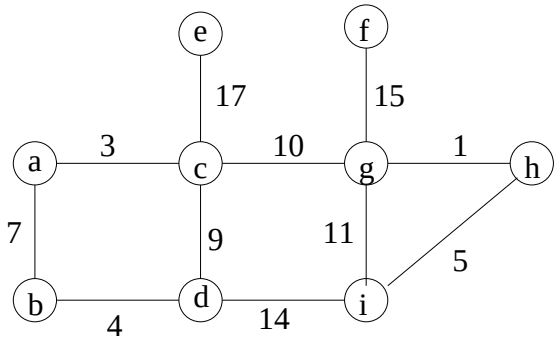
**2.1** Det vesentlige er at i) hver heap er et komplett binært tre,dvs har ett element på nivå 4 som ligger lengst til venstre, samt at ii) nøkkel for enhver node er større enn nøkkel hos eventuelle barn.

**2.2** Det vesentlige er at i) hvert søketre er et binært tre (0, 1 eller 2 barn) og ii) at for enhver node med nøkkel x er alle nøkler i dens venstre deltre mindre enn x og i dens høyre deltre større enn x.

**2.3** For heap brukes UpHeap-prosedyren, dvs at den nye nøkkelen starter i neste ledige posisjon, og 'bobler opp' til rett plass.
For binære søketrær skal den nye nøkkelverdien legges til i en ny løvnode på rett plass, og ingen andre endringer skjer i treet.

**2.4**

```
Algorithm printSorted(node v)
   if left(v) != null printSorted(left(v))
   print(v)
   if right(v) != null printSorted(right(v))
```




# Oppgave 3. Grafalgoritmer (20%)

**3.1** a, b, d, c, e, g, f, h, i

**3.2** a, b, c, d, e, g, i, f, h

**3.3** e, f, i, h, g, c, d, b, a

**3.4** ab, ac, cd, ce, cg, bd, di, gf, gh, gi, hi (alle fører til endring)

$a : 0$
$b : \infty, 7$
$c : \infty, 3$
$d : \infty, 12, 11$
$e : \infty, 20$
$f : \infty, 28$
$g : \infty, 13$
$h : \infty, 14$
$i : \infty, 25, 24, 19$

**3.5** Treet får kantene (i rekkefølge): ac, ab, bd, cg, gh, hi, gf, ce.

# Oppgave 4. Kompleksitetsanalyse (15%)

f1 er $O(NlogN)$. En ytre for-løkke som utfører while-løkken N ganger. For hver av disse utføres operasjonene inne i while-løkken $x$ ganger, hvor x er antall ganger vi kan dele N på 2 før vi når ned til 1. Per definisjon er $x = log_2 N$.

f2 er $O(N^2)$. Tildelingen i den indre for-løkken utføres 1 gang når i=1, 2 ganger når i=2, osv inntil n-1 ganger når i=n-1. Kjøretiden totalt blir dermed $O(\Sigma_{i=1}^{i=N-1} i) = O(N^2)$.

f3 er $O(N)$. k starter på 0, øker med 1 hver gang gjennom while-løkken, og vi stanser når $k = 2 * N - 1$.

f4 er $O(NlogN)$. Dette er Flettesorteringsalgoritmen (borsett fra en liten bug: c[i]=a[i] skulle være c[i-n/2]=a[i]). Analysen kan gjøres ved å se at kallet f4(N) leder til to kall f4(N/2) samt O(N) tid for å flytte elementer mellom tabeller og kjøre f3. Vi kan sette opp et (binært) tre av rekursive kall, som vil ha dybde logN, og se at det i hvert nivå vil utføres O(N) arbeid.

## Oppgave 5. Programmering (20%)

```
public class GenSort implements GeneriskSort {
  public void pQSort(RankedSequence S, Comparator C) {
    SimplePriorityQueue pq = new HeapSimplePriorityQueue(C);
    int i=0;
    while (!S.isEmpty)
      pq.insertItem(S.removeElemAtRank(i), null);
    while (!pq.isEmpty()) {
      S.insert(pq.minKey());
      pq.removeMinElement();
    }
  }
  public void enkelSort(RankedSequence S, Comparator C) {
     for (int j=0; j<S.size(); j++) {
        Object min=S.elemAtRank(j);
        int minRank=j;
        for (int i=j+1; i<S.size(); i++) {
          Object next=S.elemAtRank(i);
          if C.isLessThan(next, min) {
            min=next;
            minrank=i;
          }
        S.replaceElemAtRank(minrank, S.elemAtRank(j));
        S.replaceElemAtRank(j, min);
        }
     }
  }
}
```