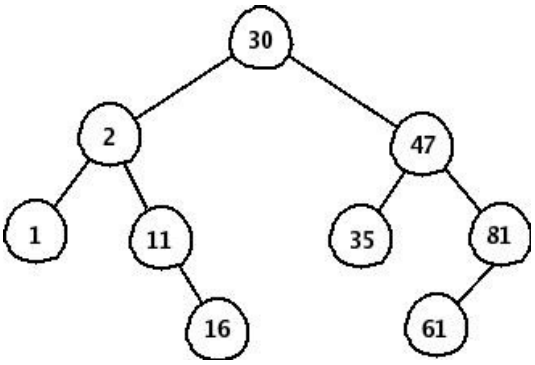
Løsningsforslag til eksamensoppgave

ITF20006 Algoritmer og datastrukturer – 22.05.2007

Oppgave 1

A.



Postorden traversering: 1-16-11-2-35-61-81-47-30

B.

```
velformet  = sann
Stack s = new Stack();

while(<flere elementer på filen og velformet er sann>){
        element = HTMLleseren.nesteElement();
        if(<element er en start-tagg>)
                s.push(element);
        else if(<element er en slutt-tagg>){
                if( s.isEmpty())
                        velformet = usann;
                else{
                        startTagg = s.pop();
                        if (<element.navn != startTagg.navn>)
                                velformet = usann;
                }
        }
}
if(!s.isEmpty()
        velformet=usann;
```

C.

1. Algoritme1 skriver summen av de n første heltallene til skjerm, som også er antallet ganger innerste løkke gjentas. For n=10 blir dette 55.

   Algoritme2 skriver de n første heltallene til skjerm, i stigende rekkefølge. For n=10: 1 2 3 4 5 6 7 8 9 10.

2. Algoritme 1: Innerste løkke gjentas n ganger for i=0, n-1 ganger for i=1 o.s.v. Totalt gjentas setningen i innerste løkke n(n+1)/2 ganger. $O(N^2)$.

   Algoritme 2: Lineær, rekursiv algoritme. O(N).

3. Algoritme 1: Nei. Siden algoritmen er kvadratisk, vil vi før oppgradering kunne forvente en 4-dobling av tidsbruken hvis vi dobler problemets størrelse: $T(2N) = c(2N)^2 = 4cN^2 = 4T(N)$. Hvis hver operasjon utføres på halve tiden etter oppgradering, vil vi følgelig kunne forvente en dobling av tidsbruken.

   Algoritme 2: Ja, siden algoritmen er lineær. $T(2N) = 2cN = 2T(N)$.

Oppgave 2

A.

```
public T fjernMinste() throws Exception{

if (start == null)
   throw new Exception("Tom liste.");

Node<T> minst = start;
start = start.neste;
antall--;

return minst.verdi;
}
```

B.

```
public void skriv() {
Node<T> n = start;
while(n!= null){
   System.out.print(n.verdi+" ");
   n=n.neste;
}
}
```

C.

```
public void settInn(T verdi ){


if (verdi == null)
   throw new NullPointerException();


antall++;

if(start == null || start.verdi.compareTo(verdi) > 0 ){
   start = new Node<T>(verdi, start);
   return;
}


Node<T> n = start;
while(n.neste != null){

   if(n.verdi.compareTo(verdi) <= 0 &&  n.neste.verdi.compareTo(verdi)>0){
      n.neste = new Node<T>(verdi, n.neste);
      return;
```

```
            }
            n=n.neste;
        }

        n.neste  = new Node<T>(verdi,  null);
    }
```

D. Her er det fristende å bruke metoden `settInn` fra oppgave 2C, og å bygge opp den nye listen ved å sette inn ett og ett element fra de to listene som skal flettes sammen. Denne algoritmen har arbeidsmengde $O(N^2)$, hvor N er summen av elementene i de to listene. En implementasjon er vist nedenfor:

```
public SortertListe<T> flett(SortertListe<T> l){

  SortertListe<T> flettet = new SortertListe<T>();

  flettet.antall = antall + l.antall;

  Node<T> n = this.start;
  while(n != null){
     flettet.settInn(n.verdi);
     n = n.neste;
  }

  n = l.start;
  while (n != null){
     flettet.settInn(n.verdi);
     n = n.neste;
  }

  return flettet;
}
```

En mer effektiv flette-algoritme er vist nedenfor. Vi starter opp med referanser til første element i hver av de to listene som skal flettes sammen, finner ut hvilken liste neste element skal velges fra og flytter tilhørende referanse ett hakk bakover i listen. Nye elementer settes sist i ny, flettet liste. Arbeidsmengde O(N).

```
public SortertListe<T> flett(SortertListe<T> liste) {

  SortertListe<T> flettet = new SortertListe<T>();
  flettet.antall = antall + liste.antall;

  flettet.start = new Node<T>(null,null);  // hjelpenode

  Node<T> p = start, q = liste.start, r = flettet.start;

  while (p != null && q != null) {
     if (p.verdi.compareTo(q.verdi) <= 0) {
        r = r.neste = new Node<T>(p.verdi,null); p = p.neste;
     } else {
        r = r.neste = new Node<T>(q.verdi,null); q = q.neste;
     }
  }

  if (p == null) p = q;

  while (p != null)  // tar med resten
  {
     r = r.neste = new Node<T>(p.verdi,null); p = p.neste;
  }

  flettet.start = flettet.start.neste;  // fjerner hjelpenoden

  return flettet;
}
```

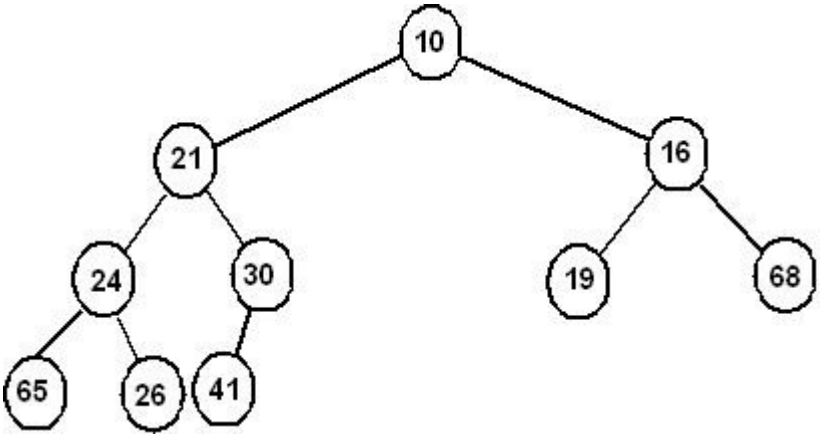
E.

```
public class ListePrioritetsKø<T extends Comparable<? super T>> implements PrioritetsKø<T> {

   private SortertListe<T> liste;


   public ListePrioritetsKø() {
      liste = new SortertListe();
   }

   public void settInn(T verdi) {
      liste.settInn(verdi);
   }

   public T taUt() {
      return liste.fjernMinste();
   }

   public int antall() {return liste.hentAntall();}
}
```

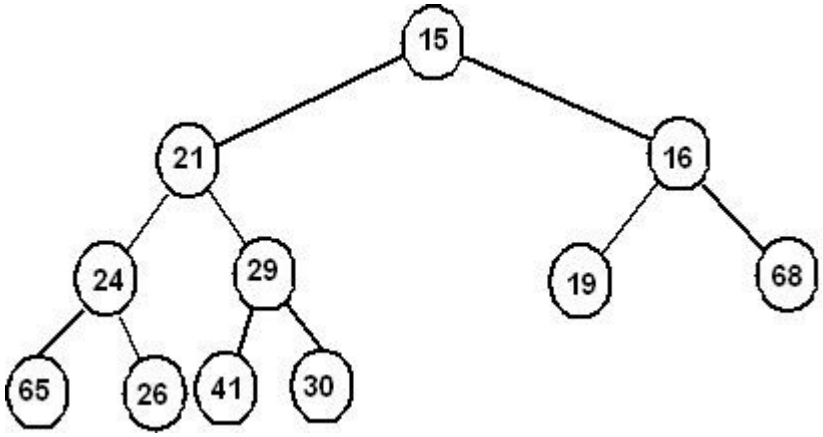
Oppgave 3

A. Foreldrenoden: k/2 (heltallsdivisjon)
Venstre barn: 2k
Høyre barn: 2k+1

B.



Etter at operasjonene er utført:



C.

```
public boolean erKomplett(T [] heap){
      boolean komplett = true;

   for (int i=1; i<heap.length && heap[i] != null; i++)
      antall++;

   for (int i = antall+1; i<heap.length && komplett; i++)
      komplett = heap[i] == null;

   return komplett;
 }
```

```
// Iterativ løsning – erMiniHeap:

private boolean erMiniHeap(T[] heap){
    int m = (antall + 1)/2;

    for (int i = 1; i < m; i++) // går kun halvveis i tabellen
    {
      if (heap[i].compareTo(heap[2*i]) > 0) return false;
      if (heap[i].compareTo(heap[2*i+1]) > 0) return false;
    }

    if ((antall & 1) == 0) // en ekstra node hvis antall er et partall
      if (heap[m].compareTo(heap[antall]) > 0) return false;

    return true;
}

// Rekursiv løsning:
      private boolean erMiniHeap(T [] heap){

      if (!erMiniHeap(heap, 1))
         return false;

      return true;
   }

   private boolean erMiniHeap(T [] heap, int index ){
      // Venstre barn:
      if (2*index <= antall){
         if (heap[index].compareTo(heap[2*index]) > 0 )
            return false;
         if (!erMiniHeap(heap, 2*index))
            return false;
```

```
        }

        // Høyre barn:
        if (2*index +1 <= antall){
             if (heap[index].compareTo(heap[2*index+1]) > 0 )
                  return false;
             if (!erMiniHeap(heap, 2*index+1))
         return false;
    }
    return true;
}
```

D.

```
    private void lagMiniHeap(T [] heap){

    rot = lagSubtre(heap, 1);
}

private Node<T> lagSubtre (T [] heap, int index){
    if(index > antall )
       return null;

    Node<T> venstre = lagSubtre(heap, 2*index);
    Node<T> hoyre = lagSubtre(heap, 2*index +1 );

    return new Node(heap[index], venstre, hoyre );
}
```