Algoritmer og datastrukturer
Løsningsforslag

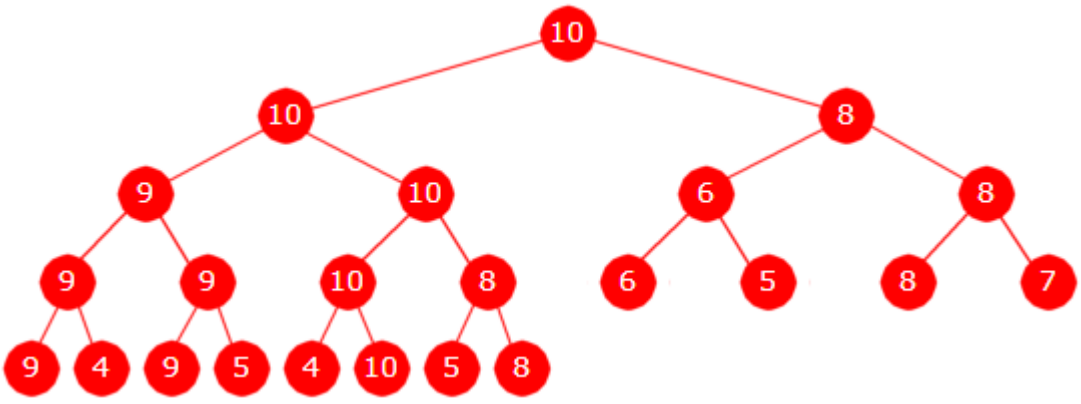
Eksamen – 30. november 2010

Oppgave 1A

Et turneringstre for en utslagsturnering med *n* deltagere blir et komplett binærtre med 2*n* – 1 noder. I vårt tilfelle får treet 2·12 – 1 = 23 noder. Verdiene (eller deltagerne) legges inn nederst i treet. Vi tenker oss at nodene har nummer fra 1 til 23 i nivåorden. Det betyr at de 12 verdiene legges inn på de 12 siste nodene slik at første verdi legges i node 12, andre verdi i node 13, osv. Det svarer til flg. kode når dette implementeres:

```
int[] deltagere = {6,5,8,7,9,4,9,5,4,10,5,8};
int n = deltagere.length;
int[] turnering = new int[2*n];  // posisjon 0 brukes ikke
System.arraycopy(verdi,0,turnering,n,n);  // kopieres inn bakerst

// Nå ser turneringstabellen slik ut:
// {0,0,0,0,0,0,0,0,0,0,0,0,6,5,8,7,9,4,9,5,4,10,5,8}

// Turneringen gjennomføres slik:
for (int k = 2*n - 2; k > 1; k -= 2)
  turnering[k/2] = Math.max(turnering[k],turnering[k+1]);

// Nå ser turneringstabellen slik ut (posisjon 0 brukes ikke):
// {0,10,10,8,9,10,6,8,9,9,10,8,6,5,8,7,9,4,9,5,4,10,5,8}
```

OBS: Oppgaven gikk kun ut på å tegne turneringstreet og finne de verdiene som vinneren slo ut. Det skulle ikke lages noen kode.



Vinneren 10 har (fra første til siste runde) slått ut 4, 8, 9, 8.

Oppgave 1B

I setningen `kø.leggInn(kø.taUt())` tas den første i køen ut og legges så inn i køen bakerst. Køen starter med 3, 1, 4, 2 og dermed blir det så 1, 4, 2, 3. Dette gjentas og tilsammen blir setningen utført 10 ganger. For hver 4. gang blir køen slik den opprinnelig var. Det ender med at køen til slutt inneholder 4, 2, 3, 1. Utskriften blir 4 2 3 1.

Oppgave 1C

Vi kan få kort kode hvis vi bruker en hjelpetabell og en metode fra klassen `Arrays`:

```
public static <T> String toString(Kø<T> kø)
{
  T[] a = (T[])new Object[kø.antall()];
  for (int i = 0; i < a.length; i++) a[i] = kø.taUt();
  for (T t : a) kø.leggInn(t);  // legger tilbake i køen
  return Arrays.toString(a);
}
```

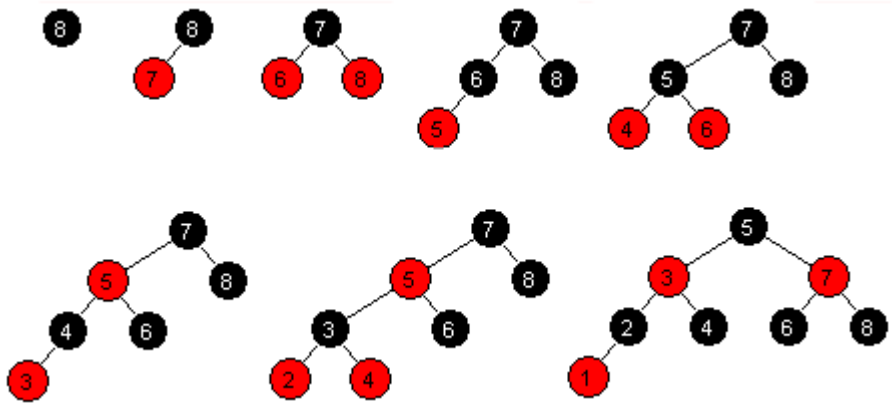
Det stod i oppgaven at det skulle brukes minst mulig ekstra «resurser». Metoden over vil derfor ikke gi fullt hus i en besvarelse siden det egentlig er helt unødvendig å bruke en hjelpetabell. Neste versjon er bedre:

```
public static <T> String toString(Kø<T> kø)
{
  StringBuilder s = new StringBuilder();
  s.append('[');

  int n = kø.antall();
  if (n > 0)
  {
    s.append(kø.kikk());
    kø.leggInn(kø.taUt());
  }

  for (int i = 1; i < n; i++)
  {
    s.append(',');
    s.append(' ');
    s.append(kø.kikk());
    kø.leggInn(kø.taUt());
  }

  s.append(']');

  return s.toString();
}
```

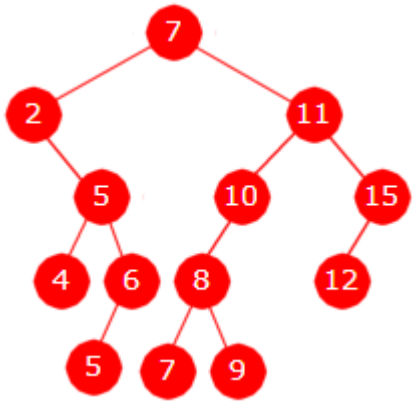
Oppgave 2A

```
public static int[] utenDuplikater(int[] a)
{
  if (a.length < 2) return a;

  Arrays.sort(a);
  int antall = 1;

  for (int i = 1; i < a.length; i++)
  {
    if (a[i-1] < a[i]) a[antall++] = a[i];
  }

  return Arrays.copyOf(a,antall);
}
```

Oppgave 2B



Oppgave 3A



Preorden: 7, 2, 5, 4, 6, 5, 11, 10, 8, 7, 9, 15, 12

Oppgave 3B

```
public T rotverdi()
{
  if (tom()) throw new NoSuchElementException("Treet er tomt!");
  return rot.verdi;
}
```

Oppgave 3C

Her er en 1. versjon av dybde-metoden:

```
private int dybde(Node<T> p)
{
  int dybde = 0;
  Node<T> q = rot;

  while (p != q)
  {
    if (comp.compare(p.verdi,q.verdi) < 0) q = q.venstre;
    else q = q.høyre;

    dybde++;
  }

  return dybde;
}
```

og en 2. versjon:

```
private int dybde(Node<T> p)
{
  Node<T> q = rot;
  int dybde = 0;

  while (true)
  {
    if (comp.compare(p.verdi,q.verdi) < 0)
      q = q.venstre;
    else if (p == q) return dybde;
    else q = q.høyre;

    dybde++;
  }
}
```

Begge versjonene av dybde-metoden vil virke og vil gi fullt hus i en besvarelse. Men er de like gode? I den 1. versjonen testes det først om *q* er lik *p*. Den testen utføres hver gang. Men for et gjennomsnittlig binært søketre vil sannsynligheten være nærmere 50% for at *p* ligger til venstre for *q*. Det tas det hensyn til i den 2. versjonen. Der vil i nær 50% av tilfellene kun den første testen utføres. Det betyr i gjennomsnitt 1,5 sammenligninger per iterasjon, mens det i 1. versjon blir alltid 2 sammenligninger i hver iterasjon. Med andre ord utføres det færre sammenligninger i den 2. versjonen.

Noen lager kode der en ikke tar hensyn til at treet kan inneholde dulikater. Det blir det trukket en del for!

Oppgave 3D

- Rotnoden er den første i preorden og dermed verdien 7
- Den neste i preorden til 11 er det venstre barnet, dvs. 10
- Den neste i preorden til 2 er det høyre barnet, dvs. 5
- Den neste i preorden til 9 er oppover i treet og til høyre, dvs. 15

Noen lager kode som om hver node har en forelderpeker. En slik peker er det ikke i denne oppgaven og slik kode er derfor verdiløs!

```
private Node<T> nestePreorden(Node<T> p)
{
  if (p.venstre != null) return p.venstre;
  else if (p.høyre != null) return p.høyre;
  else // p er en bladnode
  {
    // Må finne den nærmeste noden q oppover slik at p ligger
    // i det venstre subtreet til q samtidig som q.høyre er
    // forskjellig fra null. Da vil q.høyre være den neste i
    // preorden. Hvis ikke er den neste lik null. Letingen
    // må starte i roten:

    Node<T> q = rot;   // starter i roten
    Node<T> r = null;  // den neste i preorden

    while (true)
    {
      if (comp.compare(p.verdi,q.verdi) < 0)
      {
        if (q.høyre != null) r = q.høyre;
        q = q.venstre;
      }
      else if (p == q) return r;
      else q = q.høyre;
    }
  }
}
```

Det står i oppgaveteksten at metoden `traverserPreorden` skal kodes ved hjelp av metoden `nestePreorden`. Derfor vil det ikke bli gitt poeng for den vanlige rekursive metoden som traverserer i preorden.

```
public void traverserPreorden(Oppgave<? super T> oppgave)
{
  Node<T> p = rot;
  while (p != null)
  {
    oppgave.utførOppgave(p.verdi);
    p = nestePreorden(p);
  }
}
```

Oppgave 3E

i) Noden med verdien 2 er en forgjenger til noden med verdien 6. Dermed er avstanden mellom dem lik lengden på veien fra den første til den andre, dvs. at avstanden er 2.

Noden med verdien 9 og noden med verdien 12 hra noden med verdien 11 som felles nærmeste forgjenger. Avstanden mellom dem blir derfor lik 3 + 2 = 5.

ii) En mulig løsning er å gå fra roten og ned til nodene *x* og *y* og samtidig registrere veien. Det kan gjøres ved å notere hvilke noder som ble passert eller f.eks. ved hjelp av 0 (for venstre) og 1 (for høyre). Siden dette må gjøres for hver av nodene, kan det kanskje være lurt med en hjelpemetode.

Flg. metode legger nodene på veien fra roten og ned til noden *p* inn i en liste:

```
private Liste<Node<T>> vei(Node p)
{
  Liste<Node<T>> v = new TabellListe<Node<T>>();
  Node<T> q = rot;

  while (true)
  {
    v.leggInn(q);
    if (comp.compare(p.verdi,q.verdi) < 0) q = q.venstre;
    else if (p == q) return v;
    else q = q.høyre;
  }
}
```

Nærmeste forgjenger til nodene *x* og *y* finner vi der veiene skilles:

```
private int avstand(Node x, Node y)
 {
   Liste<Node<T>> xvei = vei(x);
   Liste<Node<T>> yvei = vei(y);

   int n = 0;
   while (n < xvei.antall() && n < yvei.antall())
   {
     if (xvei.hent(n) != yvei.hent(n)) break;
     n++;
   }

   return xvei.antall() + yvei.antall() - 2*n;
 }
```

I løsningen over starter vi i roten to ganger. En mulig forbedring kunne være å gå fra roten og ned til nodenes nærmeste forgjenger *z* ved hjelp av den vanlige ordningen i et binært søketre. Deretter kunne vi finne avstanden fra *z* til *x* og deretter avstanden fra *z* til *y*. Her må vi imidlertid være ekstra nøyaktige siden duplikater er tillatt. Det kan med andre ord være flere noder som har samme verdi som *x* og flere som har samme verdi som *y*.

Først, hvis verdien til *x* ikke er mindre enn eller lik verdien til *y*, lar vi de to bytte plass. La så *z* være rotnoden. Hvis verdien til *y* nå er mindre enn verdien til *z*, må både *x* og *y* ligge til venstre for *z*. Hvis ikke, dvs. hvis verdien til *y* er større enn eller lik verdien til *z*, så er enten *z* nærmeste felles forgjenger for *x* og *y* eller så må både *x* og *y* ligge til høyre for *z*. Her må vi passe på spesialtilfellene at *x* er lik *z* eller at *y* er lik *z*:

```
private int avstand(Node<T> x, Node<T> y)
{
```

```
  if (comp.compare(y.verdi,x.verdi) < 0)
  {
    Node<T> temp = x; x = y; y = temp;
  }

  // Vet nå at verdien til x er <= verdien til y

  Node<T> z = rot;

  while (true)
  {
    if (comp.compare(y.verdi,z.verdi) < 0) z = z.venstre;
    else if (comp.compare(x.verdi,z.verdi) < 0 || x == z || y == z) break;
    else z = z.høyre;
  }

  // Nå er z nærmeste felles forgjenger for x og y. Spesielt har vi at
  // hvis x er en forgjenger til y, så er z lik x og omvendt hvis y er
  // en forgjenger til x, så er z lik y.

  int avstand = 0;

  // går først fra z til x

  Node<T> p = z;
  while (true)
  {
    if (comp.compare(x.verdi,p.verdi) < 0) p = p.venstre;
    else if (p == x) break;
    else p = p.høyre;

    avstand++;
  }

  // går så fra z til y

  p = z;
  while (true)
  {
    if (comp.compare(y.verdi,p.verdi) < 0) p = p.venstre;
    else if (p == y) break;
    else p = p.høyre;

    avstand++;
  }

  return avstand;
}
```