Algoritmer og datastrukturer
Løsningsforslag

Eksamen – 29. november 2011

Oppgave 1A

Verdien til variabelen `m` blir lik posisjonen til den «minste»verdien i tabellen, dvs. bokstaven *A*, og det blir 6.

Oppgave 1B

OBS: Metodene `leggInn` og `fjern` oppfører seg nøyaktig slik metodene `add` og `remove` gjør i `List`-klassene i Java.

```
Metodekall                    Resultat
--------------------------------------
leggInn('A');                 A
leggInn('B');                 AB
leggInn(1,'C');               ACB
leggInn(1,'D');               ADCB
liste.fjern(2);               ADB
liste.leggInn(0,'E');         EADB
liste.fjern(1);               EDB

println(liste);               [E, D, B]
```

Oppgaven gikk ut på å avgjøre hva utskriften ble. I alle listeklassene som har blitt laget (inklusiv klassen `DobbeltLenketListe` i Oblig 2) og alle andre aktuelle klasser, har metoden `toString` blitt laget slik at tegnstrengen har hakeparenteser og komma + mellomrom mellom hvert element. Slik er det også i `java.util`. Setningen `System.out.println(liste)` gjør implisitt et kall på `toString`-metoden. Svaret på oppgaven skal derfor være: `[E, D, B]`

Oppgave 1C

Her kan vi bruke samme idé som den som brukes når innholdet i en tabell skal snus. Da går vi fra begge ender i tabellen mot midten og bytter verdier på veien.

```java
public static <T> void snu(Liste<T> liste)
{
  int v = 0, h = liste.antall() - 1;

  while (v < h)
  {
    T vtemp = liste.hent(v);
    T htemp = liste.hent(h);

    liste.oppdater(v, htemp);
    liste.oppdater(h, vtemp);

    v++; h--;
  }
}
```

Idéen over kan skrives med kortere kode:

```java
public static <T> void snu(Liste<T> liste)
{
  for (int v = 0, h = liste.antall() - 1; v < h; v++, h--)
  {
    liste.oppdater(v,liste.oppdater(h,liste.hent(v)));
  }
}
```

Det er en viktig del av denne oppgaven å kunne redegjøre for effektiviteten til metoden når den brukes på henholdsvis en `TabellListe` og `DobbeltLenketListe`. Den første klassen er detaljert beskrevet i kompendiet og den andre var emne for Oblig 2.

Metoden(e) over virker ok hvis den kjøres på en `TabellListe`. Både `hent`-metoden og `oppdater`-metoden har da direkte aksess til verdiene i den underliggende tabellen. De er med andre ord av konstant orden. Det betyr at `snu`-metoden blir av lineær orden (dvs. av orden *n* der *n* er antallet verdier i listen) siden ombyttingene skjer i en løkke.

Hvis metoden(e) brukes på en `DobbeltLenketListe` blir det annerledes. Der er både`hent`-metoden og `oppdater`-metoden av lineær orden siden det må letes etter rett posisjon enten fra venstre eller fra høyre ende av listen. Det betyr at `snu`-metoden blir av kvadratisk orden (dvs. av orden n^2der *n* er antallet verdier i listen).

Er det mulig av lage en løsning som blir av lineær orden i begge tilfellene, dvs. både for`TabellListe` og for `DobbeltLenketListe`? Java-klassen `Collections`har en snu-metode for lister (dvs. `List`). Den heter`Collections.reverse(List<?> list)`. Men alle listeklassene i Java har en`ListIterator`. En slik iterator kan gå begge veier i listen og samtidig både fjerne, legge til og oppdatere verdier under itereringen. Med slike muligheter er det mulig å lage`reverse` slik at den får lineær orden i alle tilfellene.

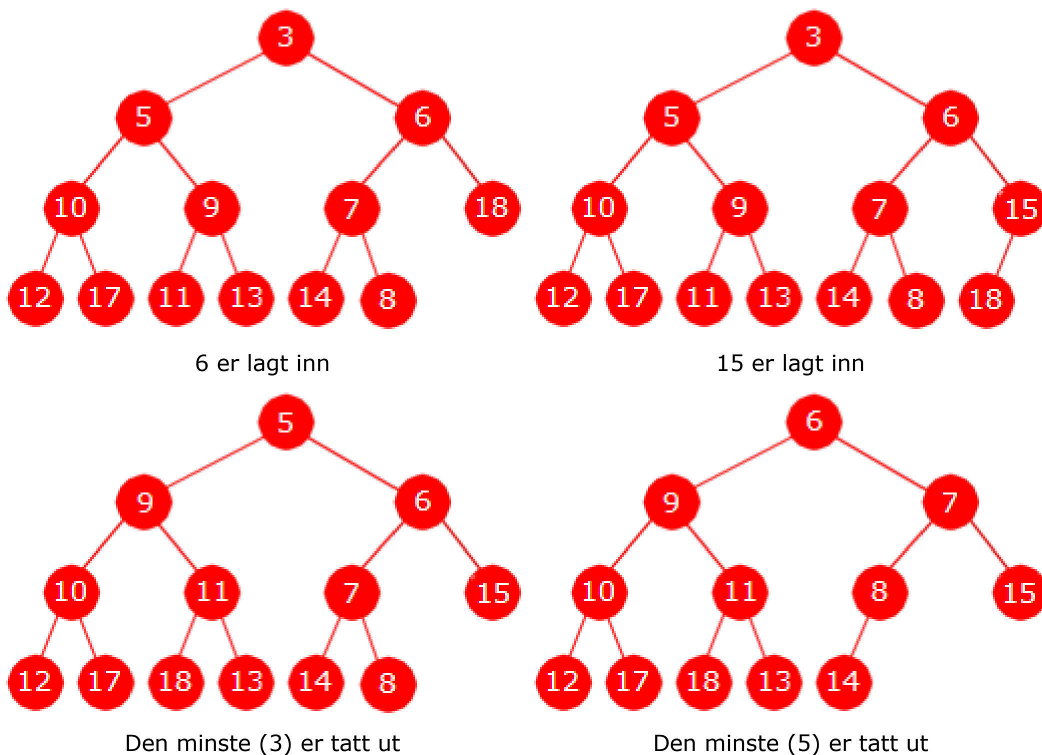
Det er mulig å få til det samme med våre listeklasser, men da på en mer primitiv eller «brutal»måte. Vi kan traversere listen ved hjelp av en iterator (lineær orden) og legge verdiene fortløpende over i en hjelpestruktur (f.eks. en stakk). Så «tømmer» vi listen ved hjelp av metoden`nullstill` ((lineær orden) og til slutt bygger vi opp listen på nytt ved å legge inn verdiene i motsatt rekkefølge (lineær orden). Sammenlagt blir dette av lineær orden:

```java
public static <T> void snu(Liste<T> liste)
{
  Stakk<T> s = new TabellStakk<>();
  for (T t : liste) s.leggInn(t);  // for-alle-løkke
  liste.nullstill();
  while (!s.tom()) liste.leggInn(s.taUt());
}
```

Hvis vi ikke vil bruke metoden `nullstill`, kan vi få til samme effekt ved hjelp av`fjern`-metoden. Det å fjerne den siste i listen er av konstant orden for begge listeklassene.

```java
public static <T> void snu(Liste<T> liste)
{
  Kø<T> kø = new TabellKø<>();
  while (!liste.tom()) kø.leggInn(liste.fjern(liste.antall()-1));
  while (!kø.tom()) liste.leggInn(kø.taUt());
}
```

Oppgave 1D



Oppgave 2A

Her er det viktig å legge merke til at parameteren til konstruktøren i klassen `EnkelPrioritetsKø` er av typen `Kø`. Grensesnittet `Kø` var satt opp i vedlegget. Det betyr at det kun er de metodene som grensesnittet har, som kan brukes i kodingen. Når klassen senere skal brukes, må det finnes en instans av en kø-klasse som skal gå inn som parameter. Men hva slags kø-klasse det er, er selvfølgelig helt ukjent når metodene i klassen `EnkelPrioritetsKø` kodes. Men selv om det skulle være kjent hvilken kø-klasse det er, er det likevel kun klassen offentlige metoder som kan benyttes og det er de som er satt opp i grensesnittet `Kø`. Med andre ord er det umulig å komme i kontakt med klassens indre, dvs. de private delene.

```
public T kikk()
{
  if (tom()) throw new NoSuchElementException();
  return kø.kikk();   // kaller en metode i kø
}

public T taUt()
{
  if (tom()) throw new NoSuchElementException();
  return kø.taUt();   // kaller en metode i kø
}
```

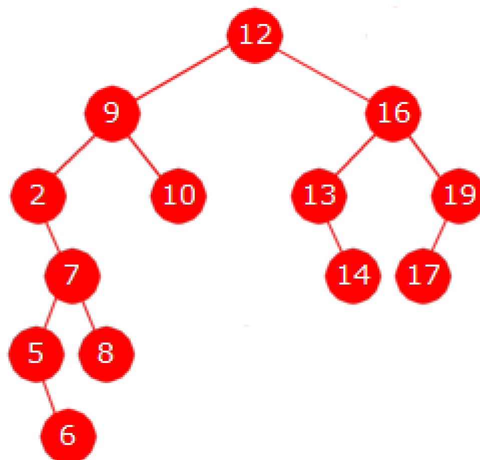
Oppgave 2B

Her må vi kikke (så ta ut og legge inn bakerst) så lenge som den verdien vi ser, på er mindre enn den nye verdien. Så legges den nye verdien inn (bakerst) og deretter resten av køen. Her vil det være helt unødvendig og dermed ikke optimalt å bruke hjelpestrukturer (det vil medføre trekk).

```
public void leggInn(T verdi)
{
  int n = kø.antall(), i = 0;

  while (i < n && c.compare(verdi,kø.kikk()) > 0)
```

```
    {
      kø.leggInn(kø.taUt());
      i++;
    }

    kø.leggInn(verdi);

    while (i < n)   // tar med resten
    {
      kø.leggInn(kø.taUt());
      i++;
    }
  }
```

Oppgave 3A



Verdiene i postorden: 6, 5, 8, 7, 2, 10, 9, 14, 13, 17, 19, 16, 12
Treet har 5 bladnoder. Sortert rekkefølge: 6, 8, 10, 14, 17

Oppgave 3B

For å finne første bladnode går vi vekselsvis mot venstre og høyre. Dvs. mot venstre hvis det er et venstre barn og ellers mot høyre. Hvis noden hverken har venstre eller høyre barn, er det en bladnode. Legg merke til at første bladnode er lik første node i postorden:

```
  public T førsteBladverdi()
  {
    if (tom()) return null;
    Node<T> p = rot;

    while (true)
    {
      if (p.venstre != null) p = p.venstre;
      else if (p.høyre != null) p = p.høyre;
      else return p.verdi;   // p er første bladnode
    }
  }
```

Det er også mulig å bruke rekursjon selv om det hverken gir mer effektiv eller enklere kode:

```
  public T førsteBladverdi()
  {
    if (tom()) return null;
    return førsteBladverdi(rot);
  }
```

```
private static <T> T førsteBladverdi(Node<T> p)
{
  if (p.venstre != null) return førsteBladverdi(p.venstre);
  else if (p.høyre != null) return førsteBladverdi(p.høyre);
  else return p.verdi;
}
```

Oppgave 3C

Denne oppgaven kan løses på flere måter. Vi kan f.eks. bruke rekursjon med «splitt og hersk».Da bruker vi generelt at antall bladnoder i et binærtre er lik summen av antallene i rotnodens to subtrær. Basistilfellene er at et tomt tre har ingen bladnoder og et tre med én node har én bladnode:

```
public int antallBladnoder()
{
  return antallBladnoder(rot);
}

private static int antallBladnoder(Node<?> p)
{
  if (p == null) return 0;  // ingen bladnoder i et tomt tre
  else if (p.venstre == null && p.høyre == null) return 1;
  else
    return antallBladnoder(p.venstre) + antallBladnoder(p.høyre);
}
```

Oppgave 3C kan også løses ved å bruke en vanlig traversering og så telle opp de bladnodene som passeres. Da kan vi f.eks. bruke en heltallstabell med med lengde 1 til opptellingen. Flg. metode traverserer i postorden. Legg merke til at bladnodene kommer i samme rekkefølge enten vi traverserer i preorden, inorden eller postorden:

```
public int antallBladnoder()
{
  int[] ant = {0};
  if (!tom()) antallBladnoder(rot, ant);
  return ant[0];
}

private static void antallBladnoder(Node<?> p, int[] ant )
{
  if (p.venstre != null) antallBladnoder(p.venstre, ant);
  if (p.høyre != null) antallBladnoder(p.høyre, ant);
  if (p.venstre == null && p.høyre == null) ant[0]++;
}
```

OBS: Noen forsøker å bruke flg. type rekursiv traversering (som ikke vil virke):

```
public int antallBladnoder()
{
  int antall = 0;
  if (!tom()) antallBladnoder(rot, antall);
  return antall;
}

private static void antallBladnoder(Node<?> p, int antall)
{
  if (p.venstre != null) antallBladnoder(p.venstre, antall);
  if (p.høyre != null) antallBladnoder(p.høyre, antall);
```

```
    if (p.venstre == null && p.høyre == null) antall++;
  }
```

Hvis man tror at koden over virker, har man en fundamental misforståelse av hvordan parameteroverføring skjer i Java. Parameteren `antall` i den rekursive metoden har det som kalles verdioverføring. Det betyr at det opprettes en lokal variabel med navnet `antall` og med verdien til parameteren som verdi. En oppdatering av denne får da kun lokal effekt, mens den variabelen som inngår som parameter når metoden kalles, blir helt uberørt.

Vi kan også traversere og telle opp iterativt ved hjelp av en stakk. Flg. metode traverserer i preorden:

```
  public int antallBladnoder()
  {
     if (tom()) return 0;

     int antallBladnoder = 0;
     Stakk<Node<T>> stakk = new TabellStakk<>();  // en stakk
     stakk.leggInn(rot);      // starter med å legge rot på stakken

     while (!stakk.tom())
     {
       Node<T> p = stakk.taUt();          // henter fra stakken
       if (p.venstre == null && p.høyre == null) antallBladnoder++;

       if (p.høyre != null) stakk.leggInn(p.høyre);     // først høyre
       if (p.venstre != null)stakk.leggInn(p.venstre);  // så venstre
     }

     return antallBladnoder;
  }
```

Oppgave 3D

```
  private class BladnodeIterator implements Iterator<T>
  {
    private Stakk<Node<T>> s = new TabellStakk<>();
    private Node<T> p;

    // Flg. hjelpemetode finner den første bladnoden i det treet
    // som har p som rotnode. Hvis en node som passeres på veien
    // dit, har et høyre barn og vi går mot venstre, legges dette
    // barnet på stakken. Det betyr at når vi tar en node p fra
    // stakken, vil neste bladnode være den første bladnoden i
    // treet med p som rot

    private Node<T> førsteBladnode(Node<T> p)
    {
      while (true)
      {
        if (p.venstre != null)
        {
          if (p.høyre != null) s.leggInn(p.høyre);
          p = p.venstre;
        }
        else if (p.høyre != null) p = p.høyre;
        else return p;
      }
    }
```

```java
  private BladnodeIterator()
  {
    if (tom()) return;

    p = førsteBladnode(rot);  // p er første bladnode i treet
  }

  public boolean hasNext()
  {
    return p != null;
  }

  public T next()
  {
    if (!hasNext()) throw
      new NoSuchElementException("Ingen flere verdier");

    T tempverdi = p.verdi;
    p = s.tom() ? null : førsteBladnode(s.taUt());

    return tempverdi;
  }

  public void remove()
  {
    throw new UnsupportedOperationException();
  }
}
```