Algoritmer og datastrukturer
Eksamen – 25.02.2015

Eksamensoppgaver

Råd og tips: Bruk ikke for lang tid på et punkt. Gå isteden videre til neste punkt og eventuelt tilbake hvis du får god tid. Resultatet fra et punkt som du ikke har løst, kan brukes senere i en oppgave som om det var løst. Prøv alle punktene. **De 10 bokstavpunktene teller likt!**

Hvis du skulle ha bruk for en datastruktur fra *java.util* eller fra kompendiet, kan du fritt bruke det uten å måtte kode det selv. Men du bør si fra dette i en kommentar.

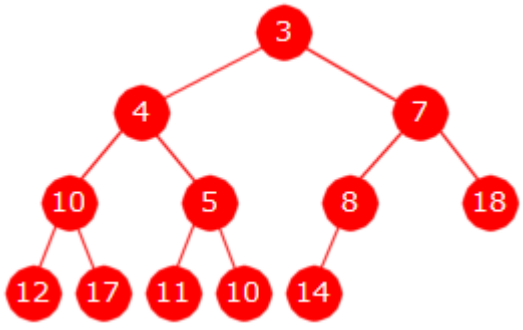
Oppgave 1

1A.
1) Gitt navnene: Per, Anders, Ole, Ali, Jens, Azra, Jasmin. Navnene skal ordnes først etter lengde (et kort navn kommer foran et som er lengre). Hvis navnene er like lange (samme antall bokstaver), gjelder vanlig alfabetisk ordning. Skriv opp navnene i den rekkefølgen som denne ordningen gir.

2) Lag et *lamda*-uttrykk for typen ***String*** som er slik at når det brukes som en komparator, vil en sortering gi den rekkefølgen som er beskrevet i punkt 1).

1B. Definisjonen av et 2-3-4 tre og en algoritme for innlegging av verdier er gitt i vedlegget. Sett inn flg. verdier i den gitte rekkefølgen i et på forhånd tomt 2-3-4 tre:
3, 10, 6, 5, 8, 12, 15. Tegn treet etter at 3, 10 og 6 lagt inn. Tegn det så etter at 5 er lagt inn. Tegn treet til slutt når alle verdiene er lagt inn.

1C.



Figur 1: En binær minimumsheap

Figur 1 til venstre viser en binær minimumsheap, dvs. et binært og komplett minimumstre. 1) Legg inn verdien 5 i treet i *Figur* 1 ved å bruke regelen for innlegging i en minimumsheap. Tegn treet.
2) Legg så inn 2 på samme måte. Tegn treet. Gå så tilbake til treet slik det er i *Figur* 1. 3) Skriv opp de verdiene som ligger i minimumsgrenen. 4) Ta ut den minste verdien ved å bruke regelen for å ta ut fra en minimumsheap. Tegn treet. Husk at treet hele tiden (etter en innlegging og etter et uttak) skal være en komplett minimumstre.

Oppgave 2

2A. Lag metoden **public static** String ***toString***(**int**[] a). Den skal returnere en tegnstreng med tallene fra heltallstabellen *a*. Strengen skal starte med [og ende med] og ha komma og mellomrom mellom tallene. Java har en slik metode i klassen Arrays. Den kan du ikke bruke. Du skal kode metoden selv. Flg. eksempel viser hvordan den skal virke:

```
int[] a = {};
int[] b = {5};
int[] c = {1,2,3,4,5,6,7,8,9,10};

System.out.println(toString(a) + " " + toString(b) + " " + toString(c));
```

```
// Utskrift: [] [5] [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

2B. Lag metoden **public static int**[] ***snitt***(**int**[] a, **int**[] b). Den skal returnere en *int*-tabell som inneholder de tallene som tabellene *a* og *b* har felles, dvs. snittet mellom dem. Det skal tas som gitt at både *a* og *b* er sortert stigende og at ingen av dem har like verdier. Målet et å få metoden så effektiv som mulig. Oppgi hvilken orden din metode har!

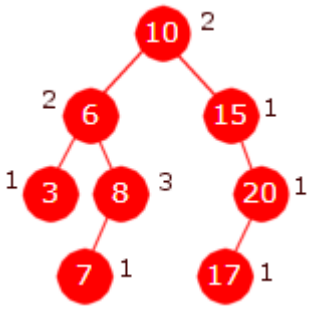
I flg. eksempel har *a* og *b* tallene 2, 9 og 12 felles. Videre har *b* og *c* ingenting felles, mens *a* og *c* har tallet 4 felles:

```
int[] a = {1,2,4,5,8,9,12};
int[] b = {2,6,9,12,15};
int[] c = {4,7,10};

String ab = toString(snitt(a, b));
String bc = toString(snitt(b,c));
String ac = toString(snitt(a, c));

System.out.println(ab + " " + bc + " " + ac);
// Utskrift: [2, 9, 12] [] [4]
```

2C. Lag metoden **public static** <T> T ***maks***(Kø<T> kø, Comparator<? **super** T> c). Den skal returnere den største verdien i *kø*. Etter et metodekall skal køen være nøyaktig slik den var. Grensesnittet Kø inngår (se vedlegget). Vi vet ikke hva slags type kø (TabellKø, LenketKø eller noe annet) som vil inngå i et metodekall. Det betyr at det er kun metodene fra grensesnittet som kan brukes. Målet er å bruke minst mulig av hjelpestrukturer - helst ingen. Her er et eksempel på hvordan metoden skal virke:

```
Integer[] a = {3,9,6,2,8,1,5,10,7,4};          // en heltallstabell
Kø<Integer> kø = new LenketKø<>();             // en Integer-kø
for (int tall : a) kø.leggInn(tall);           // legger inn i køen
Comparator<Integer> c = Comparator.naturalOrder();  // en komparator
Integer maksverdi = maks(kø, c);               // kaller metoden
System.out.println(maksverdi);                 // skriver ut
// Utskrift: 10
```

Oppgave 3



Figur 2

Her skal vi se på et binært søketre som er litt annerledes enn det normale. Se SBinTre i vedlegget. Klassen Node har en ekstra variabel med navn *forekomster*. Innlegging av en verdi skal skje som før, men hvis verdien allerede er i treet (et duplikat), skal det ikke opprettes en ny node. Isteden skal variabelen *forekomster* økes med 1. Første gang en verdi legges inn, settes *forekomster* til 1. *Figur* 2 til venstre viser hvordan treet vil bli hvis verdiene 10, 6, 8, 15, 6, 3, 20, 8, 7, 10, 17, 8 legges inn i den gitte rekkefølgen. Ved siden av hver node står det hvor mange forekomster det er av verdien. Treet har 8 noder, men 12 verdier.

3A. Gitt tallene: 8, 10, 4, 7, 8, 2, 5, 9, 12, 6, 2, 8, 10. Sett dem inn, i den oppgitte rekkefølgen, i et på forhånd tomt binært søketre av den typen som er beskrevet over. Tegn treet og skriv ved siden av hver node hvor mange forekomster det er av nodens verdi.

Et poeng med dette er å gjøre det enklere å fjerne en *verdi* fra treet. Variabelen *forekomster* i noden som inneholder *verdi*, sier hvor mange forekomster det er. Hvis *forekomster* er 0, er noden «tom». Det skal tolkes som at *verdi* ikke er i treet. Hvis *forekomster* > 0, reduseres den med 1. Dermed har treet mistet én forekomst av *verdi*. Hvis *forekomster* da blir 0, skal vi ikke fjerne noden. Den blir isteden en «tom» node (*verdi* skal fortsatt ligge der). Treet har *int*-variablen *tommeNoder*. Den økes med 1 når treet får en ny «tom» node. Metoden *fjern(T verdi)* er ferdigkodet og laget slik som beskrevet.

3B. Lag metoden **public int** inneholder(T verdi). Den skal returnere antallet forekomster av *verdi* i treet. Hvis treet ikke har noen node som inneholder *verdi* eller kun har en «tom» node med *verdi*, skal metoden returnere 0.

3C. Lag metoden **public boolean** leggInn(T verdi). Den skal legge inn *verdi* på rett sortert plass i treet. Hvis *verdi* ligger i en node i treet fra før, skal variabelen *forekomster* i noden økes med 1. Hvis det var en «tom» node, skal variabelen *tommeNoder* reduseres med 1 siden det har blitt en «tom» node mindre i treet. Hvis det ikke finnes noen node i treet som inneholder *verdi*, skal det opprettes en ny node. Til slutt skal treets *antall*-variabel økes med 1 siden treet har fått en en verdi til.

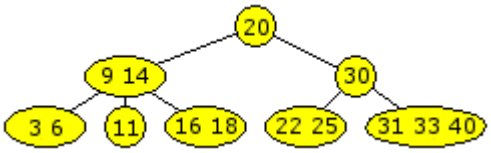
3D. Klassen SBinTre har en *iterator*. I den indre klassen InordenIterator er *p* satt opp som en instansvariabel. Den er av typen Node<T>. Den skal flytte seg fra node til node i inorden. Metoden *hasNext()* er ferdigkodet og skal ikke endres. Den returnerer *true* så lenge som *p* ikke er null. Du bestemmer selv om det er nødvendig med flere instansvariabler, hjelpestrukturer eller hjelpemetoder.

1. Lag konstruktøren for iteratorklassen. Den skal lages slik at første kall på *next()* gir verdien i den noden som kommer først i inorden.
2. Lag metoden *next()*. Den skal returnerer én og én verdi i inorden. Den skal virke slik som i kodebiten under:

```
int[] a = {8,10,4,7,8,2,5,9,12,6,2,8,10};
SBinTre<Integer> tre = new SBinTre(Comparator.naturalOrder());
for (int verdi : a) tre.leggInn(verdi);

System.out.println(tre.iterator().next());

for (Iterator<Integer> i = tre.iterator(); i.hasNext(); )
  System.out.print(i.next() + " ");

// Utskrift:
// 2
// 2 2 4 5 6 7 8 8 8 9 10 10 12
```

Vedlegg

Et **2-3-4** tre har flg. egenskaper:



Et 2-3-4 tre

- En node kan ha 1, 2 eller 3 verdier.
- En indre node har 2, 3 eller 4 barn.
- Antallet verdier i en indre node er alltid én mindre enn antallet barn som noden har.
- Alle bladnoder ligger på samme nivå i treet.
- Det tillates ikke duplikatverdier.

Algoritme for innlegging av verdi i et 2-3-4 tre:

1. Hvis verdi er den første, legg den i rotnoden (som da blir en bladnode). Hvis ikke, legg verdi på rett sortert plass i den bladnoden som den hører til ut fra sorteringen.
2. Hvis noden da får tre eller færre verdier, er innleggingen ferdig.
3. Hvis noden får fire verdier, del den i to slik at de to første verdiene kommer i den ene (første) noden og den siste verdien i den andre noden.
4. Flytt den tredje verdien (av de fire) oppover, dvs. legg den på rett sortert plass i foreldernoden. De to nodene blir venstre og høyre barn med hensyn på verdien som ble flyttet til foreldernoden. Hvis det ikke er noen foreldernode, må den opprettes først.
5. Hvis foreldernoden får 3 eller færre verdier, er innleggingen ferdig. Hvis ikke, fortsett fra punkt 3.

```
public interface Kø<T>
{
  public void leggInn(T t);   // legger inn bakerst
  public T kikk();            // ser på det som er først
  public T taUt();            // tar ut det som er først
  public int antall();        // antall i køen
  public boolean tom();       // er køen tom?
  public void nullstill();    // tømmer køen
}

public class SBinTre<T> implements Iterable<T>
{
  private static final class Node<T>       // en indre nodeklasse
  {
    private T verdi;                    // nodens verdi
    private Node<T> venstre, høyre;     // venstre og høyre barn
    private int forekomster;            // antall av denne verdien

    private Node(T verdi)               // nodekonstruktør
    {
      this.verdi = verdi;
      venstre = høyre = null;
      forekomster = 1;
    }
  }  // class Node

  private Node<T> rot;                  // peker til rotnoden
  private int antall;                   // antall verdier i treet
  private int tommeNoder;               // antall tomme noder
```

```
private final Comparator<? super T> comp;  // komparator

public SBinTre(Comparator<? super T> c)   // konstruktør
{
  rot = null;
  antall = tommeNoder = 0;
  comp = c;
}

public int antall()
{
  return antall;
}

public boolean tom()
{
  return antall == 0;
}

public int inneholder(T verdi)
{
  // kode mangler - skal lages
}

public boolean leggInn(T verdi)
{
  // kode mangler - skal lages
}

public boolean fjern(T verdi)
{
  Node<T> p = rot;                          // starter i roten
  while (p != null)
  {
    int cmp = comp.compare(verdi, p.verdi);     // sammenligner
    if (cmp < 0) p = p.venstre;                 // til venstre
    else if (cmp > 0) p = p.høyre;              // til høyre
    else break;                               // verdi ligger i noden
  }

  if (p == null) return false;              // verdi ligger ikke i treet
  if (p.forekomster == 0) return false;     // en tom node regnes ikke med

  p.forekomster--;                          // reduserer nodens antall
  if (p.forekomster == 0) tommeNoder++;     // en ny tom node

  antall--;         // én verdi mindre i treet
  return true;      // en forekomst av verdi er fjernet
}



public Iterator<T> iterator()
{
  return new InordenIterator();
}

private class InordenIterator implements Iterator<T>  // en iteratorklasse
{
  private Node<T> p;  // hjelpevariabel
```

```
    // andre aktuelle variabler skal inn her

    // aktuelle hjelpestrukturer skal inn her

    // aktuelle hjelpemetoder skal inn her

    private InordenIterator()   // konstruktør
    {
      // kode mangler - skal lages
    }

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

    public T next()
    {
      // kode mangler - skal lages
    }

  } // InordenIterator

} // SBinTre
```