Algoritmer og datastrukturer
Eksamen – 27.11.2012

Eksamensoppgaver

Råd og tips: Bruk ikke for lang tid på et punkt i en oppgave hvis du ikke får det til innen rimelig tid. Gå isteden videre til neste punkt. Hvis du i et senere punkt får bruk for det du skulle ha laget i et tidligere punkt, kan du fritt bruke resultatet som om det var løst og at løsningen virker slik som krevd i oppgaven. Prøv alle punktene. Det er ikke lurt å la noen punkter stå helt blanke. Til og med det å demonstrere i en eller annen form at du har forstått hva det spørres etter og/eller at du har en idé om hvordan det kunne løses, er bedre enn ingenting. Det er heller ikke slik at et senere punkt i en oppgave nødvendigvis er vanskeligere enn et tidlig punkt. **Alle 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 kommentere at du gjør det.

Oppgave 1

1A. Lag metoden **`public static void halvbytt(int[] a)`**. Den skal gjøre at verdiene i første og andre halvdel av tabellen *a* bytter plass. Rekkefølgen internt i halvdelene skal ikke endres. Dette vil kun virke hvis lengden til *a* er et partall. Metoden skal derfor kaste en **`IllegalArgumentException`** hvis lengden til *a* ikke er et partall. Se flg. eksempel:

```
int[] a = {1,2,3,4,5,6,7,8,9,10};
halvbytt(a);
System.out.println(Arrays.toString(a));
// Utskrift: [6, 7, 8, 9, 10, 1, 2, 3, 4, 5]
```

1B. De åtte navnene Per, Kari, Anton, Ali, Jasmin, Hansine, Knut og Anders skal sorteres etter flg. ordning: Hvis et navn x er kortere (har færre tegn) enn et navn y, skal x komme foran y. Hvis de er like lange, skal de ordnes alfabetisk.

1. Sett opp de åtte navnene i sortert rekkefølge med hensyn på den gitte ordningen.
2. Lag metoden **`compare`** i klassen **`StringKomparator`** (se vedlegget). Den skal ordne tegnstrenger etter samme type ordning som den som er satt opp for navn. Dvs. hvis en tegnstreng s er kortere enn en tegnstreng t, skal s komme foran t. Hvis de er like lange, skal de ordnes alfabetisk.

1C. Lag metoden **`public static <T> void kopier(Stakk<T> A, Stakk<T> B)`**. Metoden skal gjøre at stakken *B*, gitt at den på forhånd er tom, får de samme verdiene i samme rekkefølge som det *A* har. Stakken *A* skal etterpå være som den opprinnelig var. Grensesnittet **`Stakk`** står i vedlegget. Se flg. eksempel:

```
Stakk<Integer> A = new TabellStakk<>();  // A er tom
Stakk<Integer> B = new LenketStakk<>();  // B er tom

int[] tall = {1,2,3,4,5};
for (int k : tall) A.leggInn(k);      // legger inn tallene i A

kopier(A,B);                          // B blir en kopi av A
B.leggInn(6);                         // legger inn 6 i B
System.out.println(A + " " + B);      // bruker toString-metoden

// Utskrift: [5, 4, 3, 2, 1] [6, 5, 4, 3, 2, 1]
```

Oppgave 2

2A. La **a** være gitt ved **int[] a = {5, 3, 4, 1, 2}**. Hvordan vil **a** se ut etterpå hvis koden under anvendes på **a**? Hva kalles denne teknikken? Hvilken orden har den?

```
for (int i = 1; i < a.length; i++)
{
  int temp = a[i], j = i-1;
  while(j >= 0 && temp < a[j])
  {
    a[j+1] = a[j]; j--;
  }
  a[j+1] = temp;
}
```

2B. Hva er et komplett binærtre? Tegn et komplett binærtre med plass til 10 verdier. Legg inn tallene fra 1 til 10 slik at de kommer i sortert rekkefølge i inorden.

2C. Definisjonen av et 2-3-4 tre og en algoritme for innlegging av verdier er gitt i vedlegget. Legg verdiene 7, 5 og 12 inn i et på forhånd tomt 2-3-4 tre. Lag en tegning av treet! Legg så inn 10 i treet. Lag en tegning av hvordan treet nå ser ut. Legg så inn 3, så 8 og til slutt 9. Lag en tegning av hvordan treet nå ser ut.

Oppgave 3

3A. Gitt tallene: 7, 2, 9, 1, 5, 8, 10, 3, 6, 4. Legg dem inn, i den gitte rekkefølgen, i et på forhånd tomt binært søketre. Tegn treet. Skriv ut verdiene i nivåorden og preorden.

Vedlegget inneholder et «skjelett» for klassen **SBinTre** - et binært søketre. Av de metodene som klassen inneholder, er noen ferdigkodet. Resten skal kodes.

3B. Lag metoden **public boolean inneholder(T verdi)**. Den skal returnere **true** hvis treet inneholder parameterverdien **verdi** og returnere **false** ellers.

3C. Skriv opp første og andre verdi i postorden i treet fra *Oppgave* 3A. Lag metoden **public T andrePostorden()**. Den skal returnere den andre verdien (den som kommer som nummer to) i postorden. Hvis treet er tomt eller har kun én verdi, skal metoden kaste en **NoSuchElementException** med en passelig tekst. Lag metoden så effektiv som mulig.

3D. Lag metoden **`public static <T> SBinTre<T> kopi(SBinTre<T> tre)`**. Den skal returnere en kopi av parametertreet **`tre`**, dvs. et nytt tre, men med samme form, innhold og komparator som **`tre`**. Du bestemmer selv om du vil bruke rekursjon eller iterasjon og om du vil bruke hjelpemetoder eller ikke. Flg. eksempel viser hvordan metoden skal virke:

```
int[] a = {2,1,4,3,5};

Comparator<Integer> c = Komparator.naturlig();
SBinTre<Integer> tre = new SBinTre<>(c);
for (int k : a) tre.leggInn(k);     // bygger treet

SBinTre<Integer> kopitre = SBinTre.kopi(tre);  // lager en kopi

tre.leggInn(1);         // legger inn 1 i treet
kopitre.leggInn(5);     // legger inn 5 i kopitreet
System.out.println(tre + " " + kopitre);

// Utskrift: [1, 1, 2, 3, 4, 5] [1, 2, 3, 4, 5, 5]
```

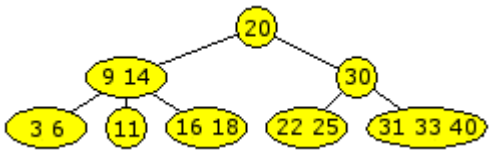
Vedlegg

```
public class StringKomparator implements Comparator<String>
{
  public int compare(String s, String t)
  {
    // kode mangler - skal lages
  }

} // class StringKomparator


public interface Stakk<T>
{
  public void leggInn(T verdi);   // legger verdi på toppen
  public T kikk();                // ser på den øverste
  public T taUt();                // tar ut den øverste
  public int antall();            // antall på stakken
  public boolean tom();           // er stakken tom?
  public void nullstill();        // tømmer stakken
}
```

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 class SBinTre<T>  // et binært søketre
{
  private static final class Node<T>      // en indre nodeklasse
  {
    private T verdi;                      // nodens verdi
    private Node<T> venstre, høyre;       // venstre og høyre barn

    private Node(T verdi) // konstruktør
    {
      this.verdi = verdi;
      venstre = høyre = null;
    }

  } // class Node

  private Node<T> rot;                          // peker til rotnoden
  private int antall;                           // antall noder
  private final Comparator<? super T> comp;     // komparator

  public SBinTre(Comparator<? super T> c)       // konstruktør
  {
    rot = null;
    antall = 0;   // et tomt tre har 0 noder
    comp = c;
  }

  public int antall()
  {
    return antall;
  }

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

  public void leggInn(T verdi)
  {
    if (verdi == null)
      throw new NullPointerException("Nullverdier er ulovlig!");
```

```
  Node<T> p = rot, q = null;              // pekere
  int cmp = 0;                            // hjelpevariabel

  while (p != null)                       // while-løkke
  {
    q = p;                                // q skal være forelder til p
    cmp = comp.compare(verdi, p.verdi);   // sammenligner
    if (cmp < 0) p = p.venstre;           // går til venstre
    else  p = p.høyre;                    // går til høyre
  }

  p = new Node<>(verdi);                  // en ny node

  if (q == null) rot = p;                 // tomt tre
  else if (cmp < 0) q.venstre = p;        // blir venstre barn
  else  q.høyre = p;                      // blir høyre barn

  antall++;                               // en ny node i treet
}

public String toString()
{
  StringBuilder s = new StringBuilder();   // StringBuilder
  s.append('[');                           // starter med [
  if (!tom()) toString(rot,s);             // den rekursive metoden
  s.append(']');                           // avslutter med ]
  return s.toString();                     // returnerer
}

private static <T> void toString(Node<T> p, StringBuilder s)
{
  if (p.venstre != null)                     // p har et venstre subtre
  {
    toString(p.venstre, s);                  // komma og mellomrom etter
    s.append(',').append(' ');               // den siste i det venstre
  }                                          // subtreet til p

  s.append(p.verdi);                         // verdien i p

  if (p.høyre != null)                       // p har et høyre subtre
  {
    s.append(',').append(' ');               // komma og mellomrom etter p
    toString(p.høyre, s);                    // siden p ikke er den siste
  }                                          // noden i inorden
}

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

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

```
  public static <T> SBinTre<T> kopi(SBinTre<T> tre)
  {
    // kode mangler - skal lages
  }

} // class SBinTre
```