Algoritmer og datastrukturer
Eksamen – 02.12.2009

Eksamenstid: 5 timer

Hjelpemidler: Alle trykte og skrevne + håndholdt kalkulator som ikke kommuniserer.

Faglærer: Ulf Uttersrud

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. Metoden med navn **`ukjent`** er satt opp i vedlegget. Hva blir utskriften fra følgende kodebit? Gi en forklaring!

```
int[] a = {1,2,3,4,5};
int[] b = {3,4,5,6,7};
int[] c = new int[a.length + b.length];

int k = ukjent(a,b,c);

for (int i = 0; i < k; i++) System.out.print(c[i] + " ");
```

1B. En kø har de metodene som er satt opp i grensnittet **`Kø`** (se vedlegget). Hva blir utskriften etter at flg. kodebit er utført? Fortell hva som har foregått. Bruk for eksempel tegninger.

```
Kø<String> kø = new TabellKø<String>();

kø.leggInn("A");
kø.leggInn("B");
kø.leggInn("C");

String s = kø.taUt(); kø.leggInn(s);
s = kø.taUt(); kø.leggInn(s);
s = kø.taUt(); kø.leggInn(s);

while (!kø.tom()) System.out.print(kø.taUt() + " ");
```

1C. Vi sier at to køer *A* og *B* er like hvis de har samme antall og verdiene i køene er parvis like. Dvs. at den første i *A* må være lik den første i *B*, den andre i *A* må være lik den andre i *B*, osv. Lag metoden **`public static <T> boolean erLike(Kø<T> A, Kø<T> B)`**. Den skal returnere *true* hvis køene *A* og *B* er like og *false* hvis de ikke er like. Køene *A* og *B* skal være nøyaktig som de var etter at metoden er ferdig. Det er kun de metodene som grensnittet **`Kø`** (se vedlegget) har som kan benyttes i kodingen. To generiske objekter (datatypen *T*) sammenlignes ved hjelp av metoden **`equals`**. Målet er å kode dette med minst mulig bruk av hjelpevariabler og hjelpestrukturer.

Oppgave 2

2A. Klassen **`Arrays`** i Java har en metode som lager en tegnstreng av innholdet i en heltallstabell. I denne oppgaven skal du lage en metode som gjør det samme. Lag metoden **`public static String toString(int[] a)`**. Den skal virke slik som utskriften viser:

```
int[] a = null; int[] b = {};
int[] c = {3}; int[] d = {1,2,3,4,5};

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

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

2B. Vi skal komprimere en sekvens med tegn ved hjelp av Huffman-teknikken. Sekvensen inneholder kun tegnene *A*, *B*, *C*, *D*, *E*, *F* og *G* med frekvenser på henholdsvis 30, 20, 3, 18, 42, 25 og 10. Tegn det Huffman-treet dette gir. Sett så opp for hvert av de 7 tegnene den bitkoden som treet bestemmer. Da en liten del av sekvensen ble komprimert ved hjelp av disse bitkodene, ble resultatet: 001100100011101110101. Hvilken delsekvens var det?

2C. Huffman-teknikken går ut på å bygge et tre ved hjelp av frekvensfordelingen for de aktuelle tegnene slik som i **2B**. Treet gir oss bitkodene. Men det er også mulig å gå motsatt vei. Hvis vi kjenner bitkodene, er det mulig å bygge opp det opprinnelige Huffmantreet.

Klassen **`Huffman`** er satt opp i vedlegget. I dens nodeklasse er kun variabelen **`tegn`** tatt med i tillegg til to pekere. Lag metoden **`private static Node byggHuffmanTre(String[] koder)`**. Tabellen **`koder`** inneholder bitkoder i form av tegnstrenger der tegnene er enten '0' eller '1'. Tegnene i en tegnstreng hentes ut ved metoden **`charAt`**. Tegnstrengen **`koder[i]`** er bitkoden til tegnet med ascii-verdi lik *i*. Hvis **`koder[i]`** er *null*, betyr det at tilhørende tegn ikke er med. Lag først en rotnode. Bruk standarkonstruktøren. En bitkode hører til en bladnode og bitene forteller hvordan en kommer dit ('0' til venstre og '1' til høyre). Hvis det mangler noder på veien dit, må de opprettes underveis. I bladnoden skal det tilhørende tegnet legges inn. Metoden skal returnere roten til det ferdige treet.

Oppgave 3

3A. Legg tallene 12, 8, 10, 17, 14, 2, 5, 20, 4, 7, i den oppgitte rekkefølgen, inn i et på forhånd tomt **binært søketre**. Tegn treet og sett et **kryss** ved siden av den noden som inneholder den nest minste verdien (dvs. verdien 4). Gjenta dette, dvs. du skal lage et nytt

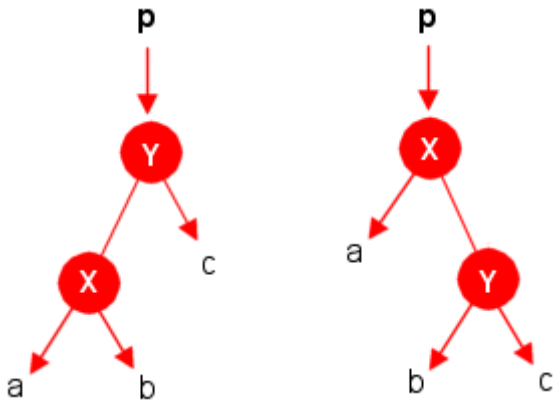
tre der tallene 8, 12, 10, 17, 5, 7, 4, 20, 2, 14 legges inn, i den oppgitte rekkefølgen, i et på forhånd tomt **binært søketre**. Tegn treet og sett et **kryss** ved siden av den noden som inneholder den nest minste verdien (dvs. verdien 4).

I vedlegget er det satt opp et «skjelett» for klassen **`SBinTre`** – et binært søketre. Hver node har en verdi og pekere til venstre og høyre barn. Klassen **`SBinTre`** har de vanlige instansvariablene. Det skal **ikke** legges inn flere instansvariabler eller statiske variabler hverken i den indre **`Node`**-klassen eller i **`SBinTre`**-klassen. Noen av metodene i «skjelettet» er ferdigkodet. Klassen skal ikke ha flere offentlige metoder enn de som allerede er satt opp.

3B. Lag metoden **`public T nestMinst()`**. Den skal returnere den nest minste verdien i treet. Hvis treet har færre enn to noder/verdier, skal det kastes en **`NoSuchElementException`** siden det da ikke finnes noen nest minste verdi. Husk at for å finne den nest minste må du først finne den minste.

3C. Et binært søketre kalles *fullstendig høyreskjevt* hvis ingen noder har venstrebarn. Som spesialtilfelle sier vi at et tomt tre er fullstendig høyreskjevt. Lag en tegning av et fullstendig høyreskjevt binært søketre som kun inneholder verdiene 2, 4, 5, 7, 8, 10. Lag så metoden **`public boolean erFullstendigHøyreskjevt()`**. Den skal returnerer *true* hvis et tre er fullstendig høyreskjevt og returnere *false* ellers.

3D. I et binært søketre kan det utføres en enkel rotasjon med hensyn på en node *p*. Lag metoden **`private static <T> Node<T> hRotasjon(Node<T> p)`**. Det kan tas som gitt at *p* har et venstre barn. Metoden skal gjøre en enkel høyrerotasjon på *p*. Hint: Opprett en hjelpevariabel *q* og la den peke på *p.venstre*.



Figuren over viser hvordan resulatet av kallet **`p = hRotasjon(p);`** skal bli. Til venstre slik treet, med *p* som rot, var før kallet og til høyre treet etter kallet. Bokstavene *X* og *Y* står for nodeverdier og **a**, **b** og **c** står for subtrær.

Gjør en enkel høyrerotasjon på rotnoden (**`rot = hRotasjon(rot);`**) i hvert av de to binære søketrærne du tegnet i punkt **3A**. Tegn resultatene.

Lag så metoden **`public void gjørFullstendigHøyreskjevt()`**. Et kall på denne metoden skal føre til at treet blir fullstendig høyreskjevt. Dette kan gjøres f.eks. ved en serie rotasjoner, men også på mange andre måter. Men det skal ikke lages noen nye noder og ingen noder skal forsvinne. Metoden skal omorganisere de nodene som treet allerede har.

```
public static int ukjent(int[] a, int[] b, int[] c)
{
  int i = 0, j = 0, k = 0;
  while( i < a.length && j < b.length)
  {
    if (a[i] < b[j]) c[k++] = a[i++];
    else if (a[i] == b[j]) { i++; j++; }
    else  c[k++] = b[j++];
  }
  while (i < a.length) c[k++] = a[i++];
  while (j < b.length) c[k++] = b[j++];
  return k;
}
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

public interface Kø<T>        // eng: Queue
{
  public void leggInn(T t);   // eng: offer/push  legger inn bakerst
  public T kikk();            // eng: peek        ser på det som er først
  public T taUt();            // eng: poll/pop    tar ut det som er først
  public int antall();        // eng: size        antall i køen
  public boolean tom();       // eng: isEmpty     er køen tom?
  public void nullstill();    // eng: clear       tømmer køen
} // grensesnittet Kø
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

public static <T> boolean erLike(Kø<T> A, Kø<T> B)
{
  return false;  // foreløpig kode - skal kodes
}
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

public class Huffman  // klasse for komprimering
{
  private static final class Node
  {
    private char tegn;     // et tegn
    private Node venstre;  // peker til venstre barn
    private Node høyre;    // peker til høyre barn

    private Node()  // standardkonstruktør
    {
      tegn = 0;
      venstre = høyre = null;
    }

  } // slutt på class Node

  private static Node byggHuffmanTre(String[] koder)
  {
    return null;  // foreløpig kode - skal kodes
  }

} // Huffman
```

```java
public class SBinTre<T> // binært søketre
{
  private static final class Node<T>    // en indre nodeklasse
  {
    private T verdi;                    // nodens verdi
    private Node<T> venstre;            // peker til venstre barn/subtre
    private Node<T> høyre;              // peker til høyre barn/subtre

    private Node(T verdi, Node<T> v, Node<T> h)  // nodekonstruktør
    {
      this.verdi = verdi;
      venstre = v;
      høyre = h;
    }

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

  } // class Node

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

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

  public void leggInn(T verdi) // skal ligge i class SBinTre
  {
    Node<T> p = rot, q = null;  // p starter i roten
    int cmp = 0;  // hjelpevariabel

    while (p != null)  // fortsetter til p er "ute av" treet
    {
      q = p;  // q skal være forelder til p
      cmp = comp.compare(verdi,p.verdi);  // bruker komparatoren
      p = cmp < 0 ? p.venstre : p.høyre;  // flytter p
    }

    p = new Node<T>(verdi);  // lager en ny node med gitt verdi

    if (rot == null) rot = p;    // treet var opprinnelig tomt
    else if (cmp < 0) q.venstre = p;
    else q.høyre = p;

    antall++;  // én verdi mer i treet
  }
```

```
  public boolean inneholder(T verdi)
  {
    Node<T> p = rot;

    while (p != null)
    {
      int cmp = comp.compare(verdi,p.verdi);
      if (cmp < 0) p = p.venstre;
      else if (cmp > 0) p = p.høyre;
      else return true;  // finner første forekomst av verdi
    }
    return false;
  }

  public int antall()
  {
    return antall;  // returnerer antallet
  }

  public boolean tom()
  {
    return antall == 0;  // tomt tre?
  }

  public T nestMinst()
  {
    return null;  // foreløpig kode - skal kodes
  }

  public boolean erFullstendigHøyreskjevt()
  {
    return false;  // foreløpig kode - skal kodes
  }

  private static <T> Node<T> hRotasjon(Node<T> p)
  {
    return null;  // foreløpig kode - skal kodes
  }

  public void gjørFullstendigHøyreskjevt()
  {
    // foreløpig kode - skal kodes
  }

} // class SBinTre
```