


Algoritmer og datastrukturer
Eksamen – 02.03.2016

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, men hvis et bokstavpunkt er delt opp, kan en krevende del telle mer enn en enkel del. Hvis du skulle trenge en hjelpestruktur (liste, stakk, kø o.l.) fra *java.util* eller fra kompendiet, kan det brukes fritt uten at du må kode det selv. Men det må brukes på en korrekt måte. Du bør si ifra hva du bruker (en kommentar i besvarelsen).

Resultater skal begrunnes. Det er spesielt viktig der det spørres etter tegninger eller utskrifter. Også i kodeoppgaver der det brukes spesielle idéer, er det viktig med forklaringer.

Oppgave 1


 **1A.** Grensesnittet **Stakk** står i vedlegget. Metoden *toString()* gir verdiene i rekkefølge fra toppen av stakken og ned til bunnen (og innrammet med [og] og adskilt med komma og mellomrom). Et kall på *toString()* gjør ingen endringer internt i en stakk. Gitt flg. kode:

```
Stakk<Character> A = new TabellStakk<>(); // en tabellstakk
Stakk<Character> B = new LenketStakk<>(); // en lenket stakk

char[] bokstaver = "ABCDEFGF".toCharArray(); // en bokstavtabell
for (char c : bokstaver) A.leggInn(c);      // legger inn i A
System.out.println(A + " " + B);           // skriver ut innholdet i A og B

while (!A.tom()) B.leggInn(A.taUt());        // flytter fra A til B
System.out.println(A + " " + B);           // skriver ut innholdet i A og B
```

Koden over er kjørbart og vil ikke gi feilmeldinger. Den har to utskriftssetninger. Hva blir skrevet ut hvis koden blir kjørt? Gi begrunnelse! Husk at der hvor en stakk inngår som parameter i en *print*-setning, er det stakkens *toString()*-metode som kalles.

 **1B.** Lag metoden **public static <T> int indeks(Stakk<T> s, T verdi)**. Den skal returnere *indeksen* til *verdi* i stakken *s*. Vi definerer at det øverste elementet (toppen) på en stakk har indeks 0, det neste øverste indeks 1, osv. nedover. Hvis *verdi* forekommer flere ganger i *s*, skal indeksen til den av dem som ligger øverst, returneres. Hvis *verdi* ikke ligger der, skal metoden returnere -1. Etter et kall på metoden skal stakken være som den var. Siden parametertypen er *Stakk*, er det kun metodene i grensesnittet *Stakk* (se **vedlegget**) som kan brukes i kodingen. Flg. kodebit viser hvordan metoden skal fungere:

```
Stakk<String> s = new LenketStakk<>(); // en lenket stakk
String[] navn = {"Ole", "Kari", "Ali", "Eli", "Per", "Pia"};
for (String n : navn) s.leggInn(n);    // legger inn i s

System.out.println(s);                // skriver ut innholdet
System.out.println("Pia har indeks " + indeks(s, "Pia"));
System.out.println("Kari har indeks " + indeks(s, "Kari"));
System.out.println("Petter har indeks " + indeks(s, "Petter"));

// Utskrift:
// [Pia, Per, Eli, Ali, Kari, Ole]
// Pia har indeks 0
// Kari har indeks 4
// Petter har indeks -1
```

1C. De fleste har opplevd at det i en kø kommer beskjed om at køen skal stenges og at alle må gå over i en annen kø. I denne oppgaven skal det lages en metode som gjør dette på en «sivilisert» måte. I metoden **public static <T> void flytt(Kø<T> A, Kø<T> B)** skal alle i kø B flyttes over i kø A. Den første i A (hvis A ikke er tom) skal fortsatt være først. Den første i B skal inn som nummer to i A. Den som var nummer to i A (hvis det var to der) skal inn som nummer tre, osv. Når dette er ferdig vil annenhver i A komme fra den opprinnelige køen A og annenhver fra B. Hvis det var flere i B enn i A, vil de gå inn bakerst.

Lag metoden! Siden parametertypen er Kø er det kun metodene i grensesnittet Kø (se **vedlegget**) som kan brukes i kodingen. Du får best score om du løser dette uten bruk av hjelpestrukturer. Flg. kodebit viser hvordan metoden skal fungere:

```
Kø<String> A = new LenketKø<>(); // oppretter kø A
Kø<String> B = new TabellKø<>(); // oppretter kø B

String[] navn1 = {"Per", "Kari", "Elin", "Ali", "Jens"};
String[] navn2 = {"Åse", "Ole", "Kjersti"};

for (String n : navn1) A.leggInn(n); // legger inn i kø A
for (String n : navn2) B.leggInn(n); // legger inn i kø B

System.out.println(A + " " + B); // skriver ut A og B
flytt(A,B); // B flyttes over i A
System.out.println(A + " " + B); // skriver ut A og B

// Utskrift: [Per, Kari, Elin, Ali, Jens] [Åse, Ole, Kjersti]
//           [Per, Åse, Kari, Ole, Elin, Kjersti, Ali, Jens] []
```

Oppgave 2

2A. I undervisningen (+ Oblig 1) har seks forskjellige sorteringsmetoder blitt gjennomgått. Oppgi navnet på så mange av dem som mulig. Bruk gjerne engelsk navn hvis du husker det best. Oppgi hvilken orden de har. Velg så to av dem og forklar i grove trekk hvordan de to virker og hvilken hovedidé de er basert på. Bruk ord, tegninger eller andre ting. Det er ikke meningen du skal lage kode, men hvis du ønsker, kan du også gjøre det.

2B. En binær minimumsheap er et binært og komplett minimumstre. Der har nodene posisjoner fra 1 til n der n er antall noder. Det gjør at treet kan representeres ved hjelp av en tabell der indeks 0 i tabellen ikke er i bruk. Tegn den binære minimumsheapen (dvs. treet) som er representert ved flg. tabell. Hvilke verdier har minimumsgrenen?

	3	5	7	8	10	8	18	17	12	11	10	14	17	22	18	21	15	18	20
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Oppgave 3

3A. Legg tallene 7, 4, 18, 1, 6, 14, 30, 12, 15, 9, 25, 22, 27, 10 og 20 (i den gitte rekkefølgen) inn i et på forhånd tomt *binært søketre*. Skriv ut trets verdier i *inorden* og *preorden*. Hvis verdiene ikke kommer sortert i din inordenutskrift, har du en feil.

Vedlegget har et skjelett for klassen SBinTre – et binært søketre der det denne gangen ikke skal være tillatt med like verdier. Det skal ikke settes inn andre variabler – hverken i treklassen eller i nodeklassen.

3B. Metoden **public boolean leggInn(T verdi)** (se **vedlegget**) er ferdigkodet med standardkoden for et binært søketre. Men denne gangen skal det ikke være tillatt med like verdier. Hvis metoden kalles med en verdi som ikke finnes fra før, skal den legges inn på vanlig måte. Men hvis den finnes fra før, skal metoden

returnere false (og ingen innlegging). Gjør de endringene som trengs for å få til dette. Det holder at du i din besvarelse kun setter opp den delen av koden der det er endringer.

3C. Lag metoden **public int dybde(T verdi)** (se [vedlegget](#)). Den skal returnere dybden til den noden i treet som inneholder *verdi*. Det er entydig siden treet ikke har like verdier. Dybden er avstanden til roten. Det betyr spesielt at roten har dybde 0. Hvis *verdi* ikke ligger i treet, skal metoden returnere -1.

3D. En *gren* i et binærtre består av nodene fra og med roten og ned til og med en bladnode. Gitt to forskjellige noder *p* og *q* som ligger på samme gren. Da sier vi at den av dem som ligger nærmest roten, er en *forgjenger* til den andre. Omvendt sier vi at den av dem som ligger lengst ned, er en *etterkommer* av den andre. Deres *avstand* er antall kanter på veien mellom dem. Hvis det ikke finnes en gren som inneholder både *p* og *q*, så må de to ha en nærmeste felles forgjenger *f*. I så fall sier vi at avstanden mellom *p* og *q* er lik avstanden fra *p* til *f* pluss avstanden fra *q* til *f*.

Eksempel: Er treet du laget i Oppgave 3A korrekt, vil avstanden mellom 18-noden (noden med verdi 18) og 10-noden være lik 4. Nærmeste felles forgjenger til 15-noden og 27-noden er 18-noden. Dermed blir avstanden mellom dem (15-noden og 27-noden) lik $2 + 3 = 5$.

i) Lag metoden **public int avstand(T verdi1, T verdi2)** (se [vedlegget](#)). Den skal returnere avstanden mellom nodene som inneholder *verdi1* og *verdi2*. Hvis en av dem (eller begge) ikke ligger i treet, skal metoden kaste et passelig unntak. Du bestemmer selv hvilken teknikk du vil bruke og om du vil bruke hjelpemetoder.

ii) Den største avstanden mellom to noder kalles treet's *diameter*. Spesielt sier vi at et tomt tre har diameter -1 og et tre med én node har diameter 0. Hva er diameter til treet du laget i Oppgave 3A? Hva blir diameter hvis du i tillegg legger inn 2 og 3? Begrunn svarene!

iii) Lag metoden **public int diameter()** (se [vedlegget](#)). Den skal returnere treet's diameter. Du bestemmer selv hvilken teknikk du vil bruke og om du vil bruke hjelpemetoder. Målet er å få en mest mulig effektiv metode. Hvilken orden vil metoden din få?

3E. Lag metoden **public static <T> SBinTre<T> kopi(SBinTre<T> tre)**. Det er en generisk metode som vi tenker oss ligger i en annen klasse enn SBinTre. Den skal returnere en kopi av parameterreet *tre*, dvs. et tre med samme form og innhold. Det er kun de offentlige metodene i SBinTre som kan brukes i konstruksjonen av kopien siden vi befinner oss utenfor klassen SBinTre. Med andre ord har vi ikke tilgang til noe av klassens indre. Det er ikke tillatt å gjøre endringer SBinTre for å få til dette.

Klassen har imidlertid en iterator siden klassen implementerer Iterable (se [vedlegget](#)) og den kan gi oss alle verdiene i *tre*. En mulig fremgangsmåte er derfor å opprette et tomt tre med samme komparator som parameterreet *tre* (klassen har en metode som returnerer den - se [vedlegget](#)). Deretter hentes alle verdiene fra *tre*. Men de kan ikke legges inn i samme rekkefølge siden iteratoren gir dem sortert (inorden). Men hver verdi i *tre* har en bestemt dybde (se Oppgave 3C). Dermed kan verdiene sorteres med hensyn på dybden slik at de kommer i nivåorden (minst nivå først). Hvis verdiene så legges inn, får vi en kopi. Gjør dette!

Vedlegg

```
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
    public String toString();       // fra toppen til bunnen
}
```

////////////////////////////////////

```
public interface Kø<T>
{
    public void leggInn(T verdi);    // legger inn bakerst i køen
    public T kikk();                // ser på den første i køen
    public T taUt();                // tar ut den første i køen
    public int antall();            // antall i køen
    public boolean tom();           // er køen tom?
    public void nullstill();        // tømmer køen
    public String toString();      // fra den første til den siste
}
```

////////////////////////////////////

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

        private Node(T verdi) // konstruktør
        {
            this(verdi, null, 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;
        comp = c;
    }

    public Comparator<? super T> comparator()
    {
        return comp;                // returnerer treets komparator
    }

    public int antall()
    {
        return antall;            // returnerer antall verdier i treet
    }

    public boolean tom()
    {
        return antall == 0;        // sjekker om treet er tomt
    }

    public boolean leggInn(T verdi)
```

```

{
    Objects.requireNonNull(verdi, "Ikke tillatt med null-verdier!");

    Node<T> p = rot, q = null;
    int cmp = 0;

    while (p != null)
    {
        q = p; // q blir forelder til p
        cmp = comp.compare(verdi, p.verdi);
        p = cmp < 0 ? p.venstre : p.høyre;
    }

    p = new Node<>(verdi);

    if (tom()) rot = p;
    else if (cmp < 0) q.venstre = p;
    else q.høyre = p;

    antall++;
    return true; // vellykket innlegging
}

public int dybde(T verdi)
{
    // skal kodes
}

public int avstand(T verdi1, T verdi2)
{
    // skal kodes
}

public int diameter()
{
    // skal kodes
}

private class InordenIterator implements Iterator<T>
{
    private Stakk<Node<T>> stakk = new TabellStakk<>();
    private Node<T> p = null;

    private Node<T> først(Node<T> q)
    {
        while (q.venstre != null)
        {
            stakk.leggInn(q);
            q = q.venstre;
        }
        return q;
    }

    private InordenIterator()
    {
        if (!tom()) p = først(rot);
    }

    public boolean hasNext()
    {

```

```
    return p != null;
}
```

```
public T next()
{
    if (!hasNext()) throw new NoSuchElementException();
```

```
    T verdi = p.verdi;
```

```
    if (p.høyre != null) p = først(p.høyre);
    else if (!stakk.tom()) p = stakk.taUt();
    else p = null;
```

```
    return verdi;
}
```

```
} // InordenIterator
```

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

```
} // class SBinTre
```