

Algoritmer og datastrukturer

Eksamen – 19.12.2016

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 du fritt bruke den uten å måtte kode den selv. Men den må brukes på en korrekt måte. Men du bør si fra om dette i en kommentar. Hvis du har idéer om hvordan ting skal løses, men likevel ikke klarer å få det til, kan du demonstrere idéene dine med ord, tegninger o.l.

Oppgave 1

1A. I metoden **public static int omorganiser(char[] c)** skal det tas som gitt at tabellen *c* kun inneholder bokstaver fra *a* til *z* (små eller store). Den skal omorganisere *c* slik at alle de små bokstavene kommer først og så alle de store. Hvilken rekkefølge de små og store bokstavene får innbyrdes, spiller ingen rolle. Tabellen inneholder generelt en blanding, men kan også inneholde kun store eller kun små bokstaver. Metoden skal returnere antallet små bokstaver. Målet (for å få best score) er å få metoden mest mulig effektiv med minst mulig bruk av hjelpestrukturer.

- Lag metoden!
- Hvilken orden får metoden din? Gi en begrunnelse!

Et eksempel på hvordan en slik metode kan virke:

```
char[] c = "AbaAcBbAAaCCbcAB".toCharArray();
int antall = omorganiser(c);
System.out.println(antall + " " + Arrays.toString(c));
// Utskrift: 7 [c, b, a, b, c, a, b, A, A, B, C, C, A, A, A, B]
```

1B. Klassen **LenketHashTabell** (se vedlegget) bruker «lukket adressering med separat lenking». Den inneholder en *tabell* med *nodereferanser* der alle i utgangspunktet er null. Et *objekt* legges inn på objektets *tabellindeks* (objektets hashverdi modulo tabellengden). Dvs. en *node* (med objektet) legges først i den (eventuelt tomme) lenkede nodelisten som hører til tabellindeksen. Se metoden *leggInn()* i **LenketHashTabell**.

En samling navn (tegnstrenger) skal legges inn. Det er en jobb å regne ut hashverdier og indekser for hånd. Dette er derfor allerede gjort for noen lengder/dimensjoner:

navn	Evj	Bo	Ali	Per	Eli	Siri	Ola	Mari	Ann	Åse
hashverdi	70088	2157	65918	80125	69762	2577197	79364	2390763	65985	192983
hashverdi % 11	7	1	6	1	0	7	10	1	7	10
hashverdi % 17	14	15	9	4	11	14	8	2	8	16
hashverdi % 23	7	18	0	16	3	1	14	5	21	13

Setningen **LenketHashTabell<String> hash = new LenketHashTabell<>(11);** oppretter en instans av klassen der den interne tabellen får dimensjon (lengde) lik 11. Legg inn én og én verdi (*leggInn*-metoden med et

navn som verdi) i den gitte rekkefølgen (dvs. Evy, Bo, Ali, osv). En node skal ha både *verdi* og *hashverdi*, men på en tegning holder det med *verdi*.

- Lag en tegning av datastrukturen når de åtte første navnene er lagt inn.
- Lag så en ny tegning som viser hvordan datastrukturen ser ut når alle er lagt inn.

Oppgave 2

2A. Metoden

public static <T> **int** *compare*(Liste<T> a, Liste<T> b, Comparator<? **super** T> comp) skal (ved hjelp av komparatoren comp) sammenligne to generiske lister leksikografisk. Den skal returnere 0 hvis de er like, et negativt tall hvis a er «mindre enn» b og et positivt tall hvis a er «større enn» b. Leksikografisk sammenligning er på samme måte som alfabetisk sammenligning av ord (bokstavene sammenlignes parvis - f.eks. er Ola «mindre enn» Ole). Men nå er det en generisk type T og ikke tegn. En tom liste er mindre enn en som ikke er tom og to tomme lister er like. Listene a og b skal ikke endres. Grensesnittet **Liste** er i vedlegget.

- Lag metoden! Målet er en metode som er effektiv for alle listetyper.

En DobbeltlenketListe (fra Oblig 2) brukes i flg. eksempel. Men metoden skal virke for enhver klasse som implementerer Liste:

```
Character[] tegn1 = {'A','B','C'}, tegn2 = {'A','B','D'};  
Integer[] tall1 = {1,2,3,4,5}, tall2 = {1,2,3,4};
```

```
Liste<Character> a = new DobbeltLenketListe<>(tegn1); // A,B,C  
Liste<Character> b = new DobbeltLenketListe<>(tegn2); // A,B,D
```

```
Liste<Integer> c = new DobbeltLenketListe<>(tall1); // 1,2,3,4,5  
Liste<Integer> d = new DobbeltLenketListe<>(tall2); // 1,2,3,4
```

```
int cmp1 = compare(a, b, Comparator.naturalOrder()); // cmp1 skal bli negativ  
int cmp2 = compare(c, d, Comparator.naturalOrder()); // cmp2 skal bli positiv
```

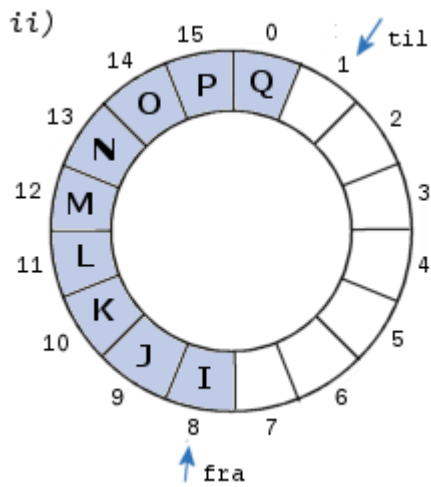
2B.

i) En TabellKø oppfører seg som en vanlig kø. Dvs. *leggInn()* legger en verdi bakerst i køen og *taUt()* tar ut den første verdien i køen. Hva blir utskriften i flg. programbit:

```
Kø<Character> kø = new TabellKø<>();  
char[] c1 = "ABCDEFGHJKLMN".toCharArray(), c2 = "NOPQ".toCharArray();  
  
for (char bokstav : c1) kø.leggInn(bokstav);  
for (int i = 0; i < 8; i++) kø.taUt();  
for (char bokstav : c2) kø.leggInn(bokstav);  
System.out.println(kø); // skriver ut køen
```

Klassen TabellKø bruker internt en såkalt sirkulær tabell. Se figuren til venstre. Der refererer indeks *fra* til den første i køen og *til* til den første ledige plassen (en bak den bakerste).

- Legg inn R, S, T og U i den sirkulære køen til venstre og ta så ut fem verdier. Tegn så køen (du behøver ikke bruke grå bakgrunn)! Hvor mange verdier har køen?
- La tabellen hete a. Vis hvordan man generelt kan (så sant tabellen ikke er full) finne antallet i en slik kø kun ved hjelp av



verdiene til *a.length*, *fra* og *til*.

Oppgave 3

3A. Gitt tallene 4, 10, 5, 12, 6, 8, 9 og 7. Legg dem inn, i den gitte rekkefølgen, i et på forhånd tomt *rød-svart binærtre*. Tegn treet etter at a) tre verdier er lagt inn, så etter at b) syv verdier er lagt inn og til slutt etter at c) alle de åtte verdiene er lagt inn.

3B. Lag metoden **public static** *<T> void omvendtkopi*(Stakk<T> a, Stakk<T> b). Den skal sørge for at stakken b blir en omvendt kopi av a (samme innhold, men i

omvendt rekkefølge). Stakken a skal etterpå være som før. Det kan tas som gitt at b på forhånd er tom. Grensesnittet **Stakk** står i vedlegget. Metoden skal virke for alle klasser som implementerer Stakk. Flg. eksempel viser hvordan metoden skal virke:

```
Stakk<String> a = new TabellStakk<>(), b = new LenketStakk<>();
a.leggInn("C"); a.leggInn("B"); a.leggInn("A");
System.out.println(a + " " + b); // utskrift: [A, B, C] []
omvendtkopi(a,b);
System.out.println(a + " " + b); // utskrift: [A, B, C] [C, B, A]
```

Oppgave 4

4A. Gitt tallene 11, 3, 25, 10, 5, 2, 15, 13, 20, 8, 22 og 16. Legg dem inn, i den gitte rekkefølgen, i et på forhånd tomt *binært søketre*. Tegn treet! Skriv ut treets verdier i *preorden* og i *postorden*!

Vedlegget har et «skjelett» av klassen **SBinTre** - et binært søketre. Der har hver node i tillegg til variablene *verdi*, *venstre* og *høyre*, også en heltallsvariabel *vAntall*. Den skal ha som verdi det antallet noder som det er i nodens venstre subtre (og dermed 0 hvis det venstre subtre er tomt). Det skal ikke være flere instansvariabler - verken i Node eller i SBinTre. Metoden *leggInn()* (som setter verdi på *vAntall*) er ferdigkodet.


4B. i) Lag en kopi av tegningen du laget i Oppgave 4A. Skriv på venstre side av hver node det antallet noder det er i nodens venstre subtre (dvs. *vAntall*). Legg så inn 4 og så 12. Deretter oppdaterer du *vAntall* for de nodene der det trengs.

ii) Legg merke til at klassen SBinTre ikke har en antallvariabel. Dermed må metodene *tom()* og *antall()* i SBinTre kodes annerledes enn normalt. Idé: Gå langs treets høyre kant (fra roten og ned til den siste i inorden). For hver node vil *vAntall* inneholde antallet noder i venstre subtre. Ved hjelp av dette kan du finne antallet noder i treet.

- Lag metoden *antall()* ved å bruke idéen beskrevet over!
- Lag metoden *tom()*!

4C. Anta nå at *vAntall* i nodene ikke har fått verdi, dvs. at den er 0 i alle noder.

- Lag metoden **public void** *settVAntall()* i SBinTre. Den skal sette korrekt verdi på variabelen *vAntall* i hver node i treet. Du bestemmer selv om du vil lage hjelpemetoder (rekursive eller iterative) eller ikke.
- Hvilken orden får metoden din? Gi en begrunnelse!

 **4D.** i) Ta utgangspunkt i treet du tegnet (med 4 og 12 lagt inn) i Oppgave 4B. Noder kan indekseres i preorden. Dvs. den første i preorden har indeks 0, den neste har indeks 1, osv. La p være noden med indeks 6 og q den med indeks 13 i preorden.

- Skriv opp nodeverdi til nodene på veien fra roten og ned til p
- Skriv opp nodeverdi til nodene på veien fra roten og ned til q

ii) Lag metoden **public** `T preorden(int indeks)` i `SBinTre`. Den skal returnere verdien i den noden som har parameterverdien `indeks` som indeks i preorden. Hvis `indeks` er utenfor treet, skal det kastes en `NoSuchElementException`. Du kan ta som gitt at variabelen `vAntall` har korrekt verdi i hver node. Målet er å få metoden så effektiv som mulig. Du bestemmer selv om du vil lage hjelpemetoder. Hvilken orden får metoden din? Gi en begrunnelse!

Vedlegg

```
public interface Liste<T> extends Beholder<T>
{
    public boolean leggInn(T verdi);           // verdi bakerst
    public void leggInn(int indeks, T verdi); // verdi på plass indeks
    public boolean inneholder(T verdi);       // er verdi i listen?
    public T hent(int indeks);                // hent verdi på plass indeks
    public int indeksTil(T verdi);            // hvor ligger verdi?
    public T oppdater(int indeks, T verdi);   // oppdater på plass indeks
    public boolean fjern(T verdi);            // fjern verdi
    public T fjern(int indeks);               // fjern verdi på plass indeks
    public int antall();                      // antallet i listen
    public boolean tom();                    // er listen tom?
    public void nullstill();                 // listen nullstilles (og tømmes)
    public Iterator<T> iterator();           // en iterator
    public String toString();                // innholdet - først til bakerst
}
```

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

```
public class LenketHashTabell<T>
{
    private static final class Node<T> // en indre nodeklasse
    {
        private final T verdi;          // nodens verdi
        private final int hashverdi;    // lagrer hashverdien
        private Node<T> neste;         // referanse til neste node

        private Node(T verdi, int hashverdi, Node<T> neste) // konstruktør
        {
            this.verdi = verdi;
            this.hashverdi = hashverdi;
            this.neste = neste;
        }
    } // class Node

    private Node<T>[] hash;           // en nodetabell
    private final float tetthet;      // eng: loadfactor
    private int grense;               // eng: threshold (norsk: terskel)
    private int antall;                // antall verdier

    @SuppressWarnings({"rawtypes","unchecked"})
    public LenketHashTabell(int dimensjon) // konstruktør
    {
        hash = new Node<T>[dimensjon];
        antall = 0;
    }
}
```

```
tetthet = 0.75f;
grense = (int)(tetthet * hash.length);
}
```

```
private void utvid()
{
    @SuppressWarnings({"rawtypes","unchecked"}) // bruker raw type
    Node<T>[] nyhash = new Node[2*hash.length + 1]; // dobling + 1

    for (int i = 0; i < hash.length; i++) // den gamle tabellen
    {
        Node<T> p = hash[i]; // listen til hash[i]
        while (p != null) // går nedover
        {
            Node<T> q = p.neste; // hjelpevariabel
            int nyindeks = p.hashverdi % nyhash.length; // indeks i ny tabell
            p.neste = nyhash[nyindeks]; // p skal legges først
            nyhash[nyindeks] = p;
            p = q; // flytter p til den neste
        }
        hash[i] = null; // nuller i den gamle
    }
    hash = nyhash; // bytter tabell
    grense = (int)(tetthet * hash.length); // ny grense
}
```

```
public boolean leggInn(T verdi)
{
    Objects.requireNonNull(verdi, "verdi er null!");

    if (antall >= grense) utvid(); // utvider

    int hashverdi = verdi.hashCode() & 0x7fffffff; // fjerner fortegn
    int indeks = hashverdi % hash.length; // finner tabellindeks

    hash[indeks] = new Node<>(verdi, hashverdi, hash[indeks]); // først i listen

    antall++;
    return true;
}

/* De øvrige metodene er ikke tatt med. */
}
```

```
////////////////////////////////////
```

```
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 og nedover
}
```

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

```
public class SBinTre<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 vAntall;        // antall noder i venstre subtre

        private Node(T verdi) // konstruktør
        {
            this.verdi = verdi;
            venstre = høyre = null;
            vAntall = 0;
        }
    } // class Node

    private Node<T> rot;
    private final Comparator<? super T> comp;

    public SBinTre(Comparator<? super T> c) // konstruktør - lager et tomt tre
    {
        rot = null; // rot er null i et tomt tre
        comp = c;   // komparatoren får verdi
    }

    public boolean leggInn(T verdi)
    {
        Node<T> p = rot, q = null;
        int cmp = 0;

        while (p != null)
        {
            q = p;
            cmp = comp.compare(verdi, p.verdi);

            if (cmp < 0)
            {
                p.vAntall++;
                p = p.venstre;
            }
            else p = p.høyre;
        }

        p = new Node<>(verdi);

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

        return true;
    }
}
```

```
public int antall()
{
    // kode mangler - skal lages
}

public boolean tom()
{
    // kode mangler - skal lages
}

public void settvAntall()
{
    // kode mangler - skal lages
}

public T preorden(int indeks)
{
    // kode mangler - skal lages
}

} // SBinTre
```