Skriftlig eksamen i Datalogi

Modul 1

Sommer 2000

Opgavesættet består af 6 opgaver, der ved bedømmelsen tillægges følgende vægte:

Opgave 1	10%
Opgave 2	10%
Opgave 3	30%
Opgave 4	12,5%
Opgave 5	12,5%
Opgave 6	25%

Alle sædvanlige hjælpemidler er tilladt. I tilfælde af unøjagtigheder i opgaveteksterne forventes det, at deltagerne selv præciserer besvarelsernes forudsætninger.

Opgavesættet består af 11 paginerede sider (incl. forsiden). Kontroller at din kopi er fuldstændig.

Opgave 1: Rekursion (10%)

En algoritme til konvertering af et positivt decimalt heltal (grundtal 10) til et binært tal (grundtal 2) er følgende:

(1) Lad d være det decimale tal.
(2) Så længe d > 0:
- Sæt r lig med d%2 (r er resten ved heltalsdivision af d med 2).
- Sæt r til venstre for sekvensen af tidligere værdier af r.
- Sæt d lig med d/2 (hvor / betegner heltalsdivision).

(3) Det binære tal udgøres af den konstruerede sekvens af r-værdier.

Spørgsmål 1.1 Benyt denne algoritme til at programmere en ikke-rekusiv metode

```
String toBinary(int d)
```

der, givet et positivt decimalt heltal, `d`, som argument, returnerer det tilsvarende binære tal repræsenteret som en tekststreng. For eksempel skal `toBinary(13)` returnere strengen `"1101"`.

Spørgsmål 1.2 Programmér en rekursiv udgave af metoden `toBinary`.

Opgave 2: Generelle træer (10%)

Nedenfor ses en skitse til en klasse, `EQ`, der benytter en union-find-algoritme til at vedligeholde en ækvivalensrelation på mængden af heltal i intervallet [0 ; `size`-1].

Klassen repræsenterer en opdeling af mængden i ækvivalensklasser. For eksempel kan mængden {0, 1, 2, .., 9} være opdelt i følgende 4 ækvivalensklasser {1, 3, 7}, {0, 5, 8}, {9}, {2, 4, 6}.

Hver ækvivalensklasse repræsenteres som et træ, i hvilket knuderne er objekter af klassen `Node`. For hver knude angiver `dad` knudens far i træet (`null`, hvis knuden er rod).

Den offentlige metode `differ` benyttes til at afgøre, om to heltal tilhører den samme ækvivalensklasse.

Den offentlige metode `union` benyttes til at forene ækvivalensklasserne svarende til to heltal.

```
class EQ {
    public EQ(int size) {
        array = new Node[size];
        for (int i = 0; i < size; i++)
            array[i] = new Node();
    }

    private class Node {
        Node dad;
    }

    private Node[] array;

    public boolean differ(int i, int j) {
        return find(array[i]) != find(array[j]);
    }

    public void union(int i, int j) {
        /* indsæt kode (se spørgsmål 2.2) */
    }

    private Node find(Node n) {
        return n.dad == null ? n : find(n.dad);
    }
}
```

Spørgsmål 2.1 Programmér en ikke-rekursiv udgave af metoden `find`.

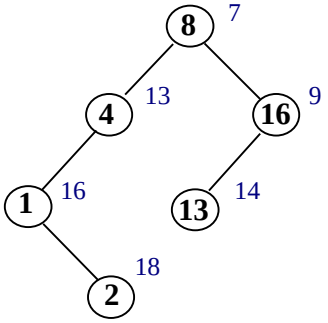
Spørgsmål 2.2 Programmér metoden `union`.

Opgave 3: Binære søgetræer og binære hobe (30%)

Denne opgaver omhandler en datastruktur, der kaldes en *treap*.

En treap er et binært træ, hvor hver knude indeholder en nøgle, en hægte til sit venstre undertræ, en hægte til sit højre undertræ samt en prioritet. En treap er et binært søgetræ, men med den ekstra betingelse, at knudernes prioriteter opfylder betingelsen for hob-orden: enhver knudes prioritet er større end eller lig med sin fars prioritet.

Nedenstående figur viser et eksempel på en treap. Nøglerne er i dette tilfælde heltal og er angivet inde i knuderne. Prioriteterne er angivet ved siden af knuderne.



Spørgsmål 3.1 Tegn en treap, der indeholder følgende 7 nøgler med tilknyttede prioriteter:

Nøgle	Prioritet
3	8
8	10
5	16
9	1
1	4
13	3
18	14

Vi ønsker nu at implementere en klasse `Treap` i Java, der benytter denne datastruktur. På næste side er vist de fundamentale dele af en sådan implementation.

Træets knuder repræsenteres som objekter af klassen `TreapNode`. Træets rod udpeges ved hjælp af referencen `root` i klassen `Treap`.

Nøglesammenligninger foretages ved hjælp af metoden `compareTo`. Hvis `lhs` og `rhs` er to `Comparable`-objekter, skal kaldet `lhs.compareTo(rhs)` returnere `-1`, `0` eller `1`, alt efter om `lhs` er mindre end, lig med eller større end `rhs`.

En knudes prioritet tildeles tilfældigt, når knuden skabes.

```
public interface Comparable {
    int compareTo(Comparable rhs);
}

public class Treap {
    public Comparable find(Comparable k) {
        return root != null ? root.find(k) : null;
    }

    public void insert(Comparable k) {
        root = root != null ? root.insert(k) : new TreapNode(k);
    }

    public void remove(Comparable k) {
         if (root != null)
             root = root.remove(k);
    }

    private TreapNode root;
}

class TreapNode {
    TreapNode(Comparable k) {
        key = k;
        left = right = null;
        priority = rand.nextInt();
    }

    Comparable find(Comparable k)     { /* se spørgsmål 3.2 */ }
    TreapNode rotateWithLeftChild()   { /* se spørgsmål 3.3 */ }
    TreapNode rotateWithRightChild()  { /* ---------------- */ }
    TreapNode insert(Comparable k)    { /* se spørgsmål 3.4 */ }
    TreapNode remove(Comparable k)    { /* se spørgsmål 3.5 */ }

    Comparable key;
    TreapNode left, right;
    int priority;
    static Random rand = new Random();
}
```

Metoden `find` i klassen `TreapNode` søger efter en specificeret nøgle, `k`, i det træ, der har knuden som rod. Hvis der findes en knude med denne nøgle, returneres den pågældende knudes nøgle, `key`; ellers returneres `null`. Nedenfor ses en rekursiv udgave af denne metode.

```
Comparable find(Comparable k) {
    if (k.compareTo(key) < 0)
        return left != null ? left.find(k) : null;
    if (k.compareTo(key) > 0)
       return right != null ? right.find(k) : null;
    return key;
}
```

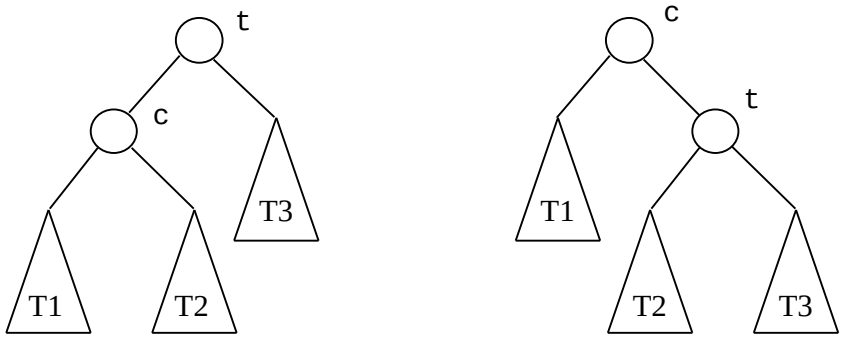
Spørgsmål 3.2 Programmér en ikke-rekursiv udgave af `find`, der ikke benytter stak.

Metoden `insert` i klassen `TreapNode` benyttes til at indsætte et ny nøgle i træet. Et udkast til implementation ses nedenfor.

```
TreapNode insert(Comparable k) {
    if (k.compareTo(key) < 0)
        left = left != null ? left.insert(k) : new TreapNode(k);
    else if (k.compareTo(key) > 0)
        right = right != null ? right.insert(k) : new TreapNode(k);
    return this;
}
```

Denne udgave af `insert` vil sikre, at en treap efter indsættelse fortsat vil være et binært søgetræ. Men det vil ikke nødvendigvis længere være en treap. For at sikre at prioriteterne efter indsættelsen overholder betingelsen for hob-orden, vil vi foretage passende rotationer i træet undervejs.

Nedenfor er skitseret en rotation af en knude `t` med sit venstre barn `c`.



Spørgsmål 3.3 Programmér metoden `rotateWithLeftChild` i klassen `TreapNode`, så den udfører denne form for rotation på det undertræ, der har den pågældende knude som rod. Metoden skal returnere roden i det nye undertræ.

På tilsvarende vis kan vi definere en rotation af en knude med sit højre barn. Bemærk at disse rotationer opretholder træet som et binært søgetræ. Vi kan derfor udnytte dem til at sørge for, at betingelsen for hob-orden opretholdes.

Spørgsmål 3.4 Færdiggør den udgave af `insert`, der er vist nedenfor, således at knudernes prioriteter opretholder betingelsen for hob-orden efter et kald.

```
TreapNode insert(Comparable k) {
    if (k.compareTo(key) < 0) {
        left = left != null ? left.insert(k) : new TreapNode(k);
        /* A: indsæt kode her */
    }
    else if (k.compareTo(key) > 0) {
        right = right != null ? right.insert(k) : new TreapNode(k);
        /* B: indsæt kode her */
    }
    return this;
}
```

Metoden `remove` i klassen `TreapNode` benyttes til at fjerne en nøgle fra træet. Også her kan vi ved hjælp af rotationer sørge for, at hob-ordenen bevares.

Spørgsmål 3.5 Færdiggør den udgave af `remove`, der er vist nedenfor, således at knudernes prioriteter opretholder betingelsen for hob-orden efter et kald.

```
TreapNode remove(Comparable k) {
    if (k.compareTo(key) < 0) {
        if (left != null)
            left = left.remove(k);
    } else if (k.compareTo(key) > 0) {
        if (right != null)
            right = right.remove(k);
    } else {
        if (left == null)
            return right;
        if (right == null)
            return left;
        /* C: indsæt kode her */
    }
    return this;
}
```

Vink. Foretag en passende rotation og fortsæt søgningen rekursivt i det roterede undertræ.

Opgave 4 (12,5%)

Betragt følgende klasse:

```
abstract class Tree {
    private Tree left, right;
    Tree(Tree actualLeft, Tree actualRight) {
        left = actualLeft;
        right = actualRight;
    }
    boolean isLeaf() {
        return (left == null && right == null);
    }
    abstract void visit();
    void traverse() {
        if (left != null) left.traverse();
        visit();
        if (right != null) right.traverse();
    }
}
```

Klassen `Tree` kan bruges til at repræsentere et såkaldt træ. En reference til et træ er i dette tilfælde en reference til træets rod. Derfra er der referencer til træets venstre og højre undertræ. Et tomt træ repræsenteres af `null`. Et "blad" er et ikke-tomt træ hvor begge undertræer er tomme.

Betragt nu følgende udvidelse af `Tree`:

```
class IntTree extends Tree {
    int data;
    ...
}
```

Klassen `IntTree` skal bruges til at repræsentere er træ hvortil der er knyttet et heltal.

Bemærk i forbindelse med spørgsmålene 4.1 og 4.2 at variablerne `left` og `right` i klassen `Tree` er private.

Spørgsmål 4.1 Tilføj til klassen `IntTree` en konstruktør der som parametre har et heltal samt to træer.

Spørgsmål 4.2 Tilføj til `IntTree` en metode `visit()` således at `traverse()` metoden udskriver alle de tal der er knyttet til træets blade. Tallene skal udskrives fra venstre mod højre (svarende til deres rækkefølge i træet).

Spørgsmål 4.3 Forklar hvordan `traverse()` metoden kan modificeres på en sådan måde at tallene i træets blade udskrives i modsat rækkefølge, dvs. fra højre mod venstre. Det skal bemærkes at dette spørgsmål kan besvares uden kendskab til en besvarelse af spørgsmål 4.2.

Opgave 5 (12,5%)

Betragt følgende klasse:

```
class Thermostate {
    private int threshold;
    private boolean on;
    void setThreshold(int actualThreshold) {
        threshold = actualThreshold;
    }
    boolean isOn() {
        return on;
    }
}
```

Klassen `Thermostate` skal bruges til at repræsentere en termostat; f.eks. en der sidder på en radiator. Variablen `threshold` indeholder den temperatur under hvilken termostaten skal være on; hvis f.eks. `threshold` indeholder værdien 21, så skal variablen `on` have værdien `true` hvis den aktuelle temperatur er mindre end 21, og den skal have værdien `false` hvis den aktuelle temperatur er større end eller lig med 21. Men kan "skrue på" termostaten vha. metoden `setThreshold` og aflæse den vha. `isOn`.

Spørgsmål 5.1 Tilføj til klassen `Thermostate` en metode `update()` der på passende vis sætter termostatens variabel `on` til `true` eller `false` givet en aktuel temperatur.

Betragt nu følgende udvidelse af `Thermostate`:

```
class EconomyThermostate extends Thermostate {
    private int maxThreshold;
}
```

En økonomitermostat har den egenskab at den maksimalt kan skrues op på den temperatur der er indeholdt i variablen `maxThreshold`.

Spørgsmål 5.2 Tilføj til klassen `EconomyThermostate` en konstruktør der tager eet argument, nemlig den temperatur den konstruerede økonomitermostat maksimalt skal kunne skrues op på.

Spørgsmål 5.3 Tilføj til klassen `EconomyThermostate` en metode `setThreshold` der givet en ønsket temperatur "skruer op eller ned" på termostaten, dvs. sætter økonomitermostatens variabel `threshold` til en passende værdi.

Bemærk i forbindelse med spørgsmål 5.3 at variablerne `threshold` og `on` i klassen `Thermostate` er private.

Opgave 6 (25%)

Betragt klassen:

```
public class Dato {
    int dag,maaned,aar;
}
```

Spørgsmål 6.1 Tilføj til klassen en konstruktor der kaldes med tre argumenter: dag, maaned og aar:

```
 Dato(int dag, int maaned, int aar)
```

Spørgsmål 6.2 Tilføj en `toString`-metode til klassen. Metoden skal returnere datoen på en form som f.eks. "26/6-2000".

Spørgsmål 6.3 Tilføj en metode `foer` der kaldes med en anden dato som argument. For to datoer `d1` og `d2` skal kaldet `d1.foer(d2)` returnere sand (`true`) hvis `d1` kommer før `d2` og ellers returnere falsk (`false`).

Et projekt repræsenteres som et objekt af klassen:

```
public class Projekt {
  String titel;
  Dato start,slut;
}
```

Spørgsmål 6.4 Tilføj en metode `status` der kaldes med en dato som argument og returnerer projektets status den pågældende dato. Status returneres som en af tekststrengene "ej påbegyndt", "undervejs", "færdigt".

(Det kan i dette og senere spørgsmål antages at metoden `foer` fra spørgsmål 6.3 er til rådighed - også selv om man ikke har besvaret spørgsmålet).

Vi ønsker at arbejde med projekter med en midtvejs milepæl. Milepælen er en dato der ligger mellem start og slutdato.

```
class ProjektMedMilepael extends Projekt{
  Dato Milepael;
}
```

Spørgsmål 6.5 Tilføj konstruktorer til `Projekt` og `ProjektMedMilepael`. Konstruktorerne skal kaldes med argumenter der bruges til at initialisere objekternes felter.

Spørgsmål 6.6 Tilføj en metode `status` til klassen `ProjektMedMilepael.` Metoden kaldes som den tilsvarende metode i projekt, men for projekter undervejs returneres "før milepæl" eller "efter milepæl".

Spørgsmål 6.7 Udvid metoden `status` i klassen `ProjektMedMilepael` med kontrol for at datoerne kommer i rigtig rækkefølge (startdato før milepæl før slutdato) og at ingen dato er `null`. Såfremt en af disse betingelser er tilstede skal metoden kaste en undtagelse. Man kan her benytte Javas standard klasse `Exception`.