

Von: Sebastian Schöner
Fiete Botschen

f)
e)
d)
c)

[illegible]

Ablauf: $a + 2\text{er-Komplement}(b) = c$
(b' Komplement, b'' 2er-Komplement)

[illegible][illegible]

c) $-c = 2er\text{-Komplement}(c)$

a) Sowohl durch ein eindimensionales Array mit $n * m$ Stellen als auch durch ein zweidimensionales Array mit m Zeilen die jeweils n Stellen beinhalten.

Weiter denkbar wären alle weiteren linearen Datenstrukturen

b)

Annahme: Funktion wird nur auf n x m Matrizen aufgerufen!

```
public static int[][] transponiere(int[][] m)
{
    if(m.length > 0 && m[0].length > 0)
    {
        int[][] res = new int[m[0].length][m.length];

        for(int zeile = 0; zeile < m.length; zeile++)
        {
            for(int spalte = 0; spalte < m[0].length; spalte++)
            {
                res[spalte][zeile] = m[zeile][spalte];
            }
        }
        return res;
    }
    else
    {
        return null;
    }
}
```

- c) Sei n die Länge/Breite der Matrix (eine n-x-n Matrix). Dann gibt es n^2 viele Felder, die alle einmal besucht werden. Also ist die Laufzeit $O(n^2)$.

Aufgabe 4)

a)

// in ListElement

```
public void helper()
{
    if(this.next != null)
    {
        this.next.helper();
        TextIO.put(this.first);
    }
    else
    {
        TextIO.put(this.first);
    }
}
```

#in LineareListe

```
public void printListReverse()
{
    if(this.head != null)
    {
        this.head.helper();
    }
}
```

b) Der Anfang der linearen Liste zeigt nach Durchlauf der Funktion auf das vormals sechste Element der Liste oder, falls die Liste kürzer war, auf das entsprechend letzte Element der Liste. Falls die Liste leer war, zeigt der Anfang ins nichts(null).

Die Schleife wird genau dann beendet, wenn 4 Durchläufe möglich waren oder falls vorher das nächste Listenelement null ist.

c) Die Methode ist sowohl deterministisch als auch determinierend als auch terminierend.

Der Ablauf ist vorhersagbar, das Ergebnis ebenfalls.

Aufgabe 5)

a)

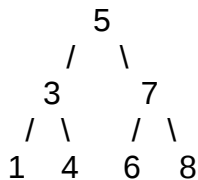
```
public class Btree
{
    int elem;
    Btree l;
    Btree r;
}
```

```
public class Broot
{
    Btree root;
}
```

b)

Kleineres Element als aktueller Knoten => wird nach links weitergegeben.

Größeres Element als aktueller Knoten => wird nach rechts weitergegeben.



c)

// in Broot

```
public int min()
{
    if(this.root != null)
    {
        return this.root.min();
    }
    else
    {
        return Integer.MAX_VALUE;
    }
}
```

```
// in Btree
public int min()
{
    if(this.l == null)
    {
        return this.elem;
    }
    else {
        return this.l.min();
    }
}
```

d)

```
# in Broot
public boolean binarySearch(int x)
{
    return (this.root != null) ? this.root.binarySearch(x) : false;
}
```

```
#in Btree
public boolean binarySearch(int x)
{
    if(this.elem == x)
    {
        return true;
    }
    else if(x < this.elem && this.l != null)
    {
        return this.l.binarySearch(x);
    }
    else if(x > this.elem && this.r != null)
    {
        return this.r.binarySearch(x);
    }
    else
    {
        return false;
    }
}
```

Sei der Baum

```

      2
     / \
    1   3
```

- BinarySearch(1) auf 2 => BinarySearch auf linken Baum
- BinarySearch(1) auf 1 => return true
- BinarySearch(5) auf 2 => BinarySearch auf rechten Baum
- BinarySearch(5) auf 3 => rechter Teilbaum null => return false.

e) Im schlechtesten Fall sind die Elemente linear angeordnet, beispielsweise aufsteigend. Dann entspricht der Aufbau einer Liste und die Suche ist von linearer Laufzeit. Geht man davon aus, dass die Elemente balanciert verteilt sind, halbieren sich in jedem Schritt „eine Ebene weiter runter“ die Anzahl der potentiell zu besuchenden Knoten, daraus resultiert eine Laufzeit von $O(\log n)$.

Aufgabe 6)

a)

23, 1, 6, 19, 14, 18, 8, 24, 15

23

1 23

1 23 6

1 19 6 23

1 14 6 23 19

1 14 6 23 19 18

1 14 6 23 19 18 8

1 14 6 23 19 18 8 24

1 14 6 15 19 18 8 24 23

b) Zu erst alle Zahlen in den Heap einfügen, dann jeweils das erste Element des Heaps (= das kleinste) entfernen und an die Ausgabelist anhängen (bzw. in das Ausgabearray schreiben).

Aufgabe 7)

1. ja

2. ja

3. nein

4. ja

5. ja

6. ja

7. nein

8. ja

Aufgabe 8)

a)

```
int countNumber(int[] arr, int num)
{
    int count = 0;
    for(int i = 0; i < arr.length; i++)
    {
        if(arr[i] == num)
        {
            count ++;
        }
    }
    return count;
}
```

b)

```
int findNumberPos(int[] arr, int num)
{
    int numCount = countNumber(arr, num);
    int res = new int[numCount];
    for(int i = 0; i < arr.length; i++)
    {
        if(arr[i] == num)
        {
            res[numCount] = i;
            numCount++;
        }
    }
    return res;
}
```

Aufgabe 9)

a)

```
double dotProduct(double[] x, double[] y) {
    return dotProductHelper(x, y, 0)
}

double dotProductHelper(double[] x, double[] y, int idx) {
    if (idx >= x.length) {
        return 0;
    } else {
        return x[idx] * y[idx] + dotProductHelper(x, y, i+1);
    }
}
```

b) Linear-rekursiv

c) Alle Typen von Rekursion lassen sich in Iterationen umwandeln. In diesem Falle ist bspw. die Angabe einer iterativen Methode eine ausreichende Begründung:

```
double dotProduct(double[] x, double[] y) {  
    double sum = 0.0;  
    for (int i = 0; i < x.length; i++)  
        sum += x[i] * y[i];  
    return sum;  
}
```

Aufgabe 10)

a) todo

b) Es wird in den beiden Zeilen zu erst eine Variable mit Namen 'maier' vom Typ 'TurnierSpieler' deklariert. In der zweiten Zeile wird 'maier' mit einer neuen Instanz von 'TurnierSpieler' initialisiert.

Die zweite Anweisung wird benötigt, damit 'maier' überhaupt einen Wert hat; sonst würde nämlich 'maier == null' gelten. Lediglich deklarierte Variablen werden nicht automatisch initialisiert.

c)

maier.name = „Maier“.