Nicolai Schönberg, 9314806, Einsendeaufgaben 2, Datenstrukturen I

Aufgabe 1

a)

```
public class Tree {
	Node root;

	public Tree(Node rootNode) {
		this.root = rootNode;
	}

	public void printAllKeys() {
		printKeys(this.root);
	}

	public void printKeys(Node startNode) {
		if (startNode != null) {
			System.out.println(startNode.getKey());

			if (startNode.rightSibling != null) {
				printKeys(startNode.rightSibling);
			}

			if (startNode.leftmostChild != null) {
				printKeys(startNode.leftmostChild);
			}
		}
	}
}
```

b)

Ich durchlaufe den Tree in gleicher Weise wie bei a), allerdings starte ich nicht beim root Node, sondern beim leftmostChild des root Nodes. Aufruf mit getTreeGrade()

```
	public int getTreeGrade() {
		this.gradeOfTree = 0;
		this.getGradeOfNode(this.root.leftmostChild);
		return this.gradeOfTree;
	}

	private void getGradeOfNode(Node startNode) {
		if (startNode != null) {
			// System.out.println(startNode.getKey());

			if (startNode.rightSibling != null) {
				getGradeOfNode(startNode.rightSibling);
			}

			if (startNode.leftmostChild != null) {
				getGradeOfNode(startNode.leftmostChild);
			}
```

```
                Node currentNode = startNode;
                int grade = 1;
                while (currentNode.rightSibling != null) {
                        grade++;
                        currentNode = currentNode.rightSibling;
                }

                if (grade > this.gradeOfTree) {
                        gradeOfTree = grade;
                }

        }
    }
```

c) Klasse ArrayNode zur Verdeutlichung der eigentlichen Konvertierung

```
public class ArrayNode {
    public Elem key;
    public ArrayNode[] sons;

    public ArrayNode(Elem value, int sonCount) {
        sons = new ArrayNode[sonCount];
    }

    public void setKey(Elem newValue) {
        this.key = newValue;
    }
}
```

Eigentliche Konvertierung:

```
    public ArrayNode ConvertToArrayTree() {
        return ConvertToArrayNode(this.root);
    }

    public ArrayNode ConvertToArrayNode(Node node) {
        ArrayNode anode = null;
        int nodeIndex = 0;
        if (node != null) {
            anode = newnode_arr(this.getTreeGrade());
            anode.setKey(node.getKey());

            Node currentNode = node.leftmostChild;
            while (currentNode != null) {
                anode.sons[nodeIndex++] = ConvertToArrayNode(currentNode);
                currentNode = currentNode.rightSibling;
            }
        }
        return anode;
    }
```

```
	private ArrayNode newnode_arr(int grade) {
		return new ArrayNode(new ElemInteger(0), grade);
	}
```

Aufgabe 2

a)

Sie können den Code auch in meinem Git-hub einsehen:
https://github.com/Grrbrr404/studium_ea/tree/master/src

```
public interface SetElement {

	/**
	 * @return Eindeutiger Schlüssel des Elements
	 */
	int getKey();

	/**
	 * @return Einen Namen
	 */
	String getName();
}

/**
 * Diese Basisklasse stellt durch den Konstruktor sicher, das jedes Element
 * seinen eigenen eindeutigen Schlüssel aus dem KeyPoolSingleton erhält
 *
 * Elemente die in das Set eingefügt werden sollen, sollten von dieser Klasse
 * abgeleitet werden.
 */

public abstract class BaseElement implements SetElement {
	private int key;

	public BaseElement() {
		key = KeyPoolSingleton.getInstance().getNextKeyFor(this);
	}

	public int getKey() {
		return key;
	}
}
```

```java
/**
 * Diese Klasse dient dazu eindeutige Schlüssel zur Programmlaufzeit zu vergeben.
 * Diese eindeutigen Schlüssel werden benötigt um die Laufzeit von O(1) bei den
 * Methoden insert, contains und remove zu ermöglichen
 */
class KeyPoolSingleton {
	public static KeyPoolSingleton instance = null;
	private int nextKey = 0;


	/**
	 * Merkt sich für jede Klasse einen eindeutigen Schlüssel
	 */
	private Dictionary<Object, Integer> keyMemory;

	public KeyPoolSingleton() {
		keyMemory = new Hashtable<Object, Integer>();
	}


	/**
	 * Gibt die einzige Instanz des KeyPoolSingleton zurück
	 * @return Singleton instance
	 */
	public static KeyPoolSingleton getInstance() {
		if (instance == null) {
			instance = new KeyPoolSingleton();
		}
		return instance;
	}


	/**
	 * Gibt für ein Element einen eindeutigen Schlüssel zurück
	 * @param element
	 * @return key
	 */
	public int getNextKeyFor(SetElement element) {
		int key;
		if (keyMemory.get(element.getClass()) == null) {
			key = nextKey++;
			keyMemory.put(element.getClass(), key);
		}
		else {
			key = keyMemory.get(element.getClass());
		}

		return key;
	}

}
```

```java
/**
 * Das eigentliche Set / Menge. Verwaltung von Objekten
 */
public class Set {

	/**
	 * Speicher für alle Objekte die das Interface SetElement implementieren
	 */
	private SetElement[] memory;

	/**
	 * Gibt die Anzahl der Elemente im Set an
	 */
	private int elementCount = 0;

	/**
	 * Die maximale Größe des Sets
	 */
	private int sizeOfSet;

	/**
	 * Vorhandene Elemente werden zusätzlich auf dem remove stack abgelegt
	 */
	private Stack<SetElement> removeStack;

	public Set(int size) {
		this.memory = new SetElement[size];
		this.sizeOfSet = size;
		this.removeStack = new Stack<SetElement>();
	}
}
```

b)

```java
/**
 * Es wird geprüft ob das angegebene Element bereits an der
 * vorgesehenen Stelle (durch element.getKey()) vorhanden ist. Laufzeit O(1)
 * @param element
 * @return true falls das Element bereits in memory vorhanden ist
 */
public boolean contains(SetElement element) {
	boolean result = false;
	if (element.getKey() < sizeOfSet) {
	  result = memory[element.getKey()] != null
			&& memory[element.getKey()].getKey() == element.getKey();
	}
	return result;
}
```

```java
/**
 * Fügt ein neues Element in das Set ein, sofern es noch nicht vorhanden ist
 * Laufzeit O(1)
 * @param element
 */
public void insert(SetElement element) {
	if (!contains(element)) {
		if (element.getKey() < sizeOfSet) {
			memory[element.getKey()] = element;
			removeStack.push(element);
			elementCount++;
		}
		else {
			System.out.println("Error: Element " + element.getName()
					+ " does already exist.");
		}
	}
	else {
		System.out.println("Error: Element " + element.getName()
				+ " does already exist.");
	}
}


/**
 * Löscht mit hilfe des remove Stack das Element was zuletzt in das Set
 * eingefügt wurde.Ich habe die Aufgabe "ein beliebiges Element löschen"
 * so verstanden, das ich mir aussuchen darf welches ich löschen und
 * zurückgeben möchte. Laufzeit O(1)
 * @return Element falls Set nicht leer. Sonst null
 */
public SetElement remove() {
	SetElement result = null;
	if (elementCount > 0) {
		result = removeStack.pop();
		memory[result.getKey()] = null;
		elementCount--;
	}
	return result;
}
```

Aufgabe 3

a)

Eingabefolge:

Für die interne Darstellung der Eingabefolge würde ich eine Queue verwenden. Sie bietet den Vorteil das ich die einzelnen Zeichen der Eingabefolge hintereinander in der richtigen Reihenfolge verarbeiten kann. Ein Stack würde auch funktionieren, hätte aber den Nachteil das die Eingabe in umgekehrter reihenfolge verarbeitet werden müsste.

Übersicht:

Für die Übersicht würde ich ein Zweidimensionales-Array verwenden. Dann kann ich alle Tripel wie folgt zusammenfassen:

Nichtterminal	**Terminal**	**Anwendung Regel Nr**
Stmt	Id	1
Stmt	If	2
Stmt	While	3
Stmt	Then	Error
Stmt	Do	Error
Stmt	Else	Error
...	...	...

Regeln:

Die Regeln würde ich als Baum implementieren. Dabei würde jede Regel ein eigener Knoten sein und alle erlaubten nachfolger als Söhne implementiert werden. Mann kann sich dann bei der Abarbeitung der Eingabe von oben nach unten durch den Baum „hangeln“ und weiß sofort ob ein Syntaxproblem vorliegt oder nicht.

Zustand der Regelabarbeitung:

Nachdem eine Regel erfolgreich abgearbeitet wurde, kann sie in einem Stack abgelegt werden. Dadurch ist es möglich immer die zuletzt abgearbeitete Regel zu erhalten um im Regel-Baum nach zu sehen ob auch der nächste Durchlauf gültig ist.

b)

- Zu beginn müssen die Zeichen mit enqueue der Reihenfolge nach in die Queue geladen werden
- Sobald die Eingabe geladen ist, kann das erste Element (E1) mit dequeue aus dem Queue entfernt werden. Es muss sich vom Programm intern gemerkt werden.
- Die Eingabe-Queue enthält nun ein Element weniger, an erster Stelle in der Queue steht nun das Element 2 (E2)
- Durch den Befehl front der Queue, kann das oberste Elemente E2 angesehen werden und bildet mit E1 ein Tupel (E1, E2)
- Nachdem eine Regel erfolgreich auf das Tupel angewendet worden ist, wird die angewandte Regel mit dem Befehl push oben auf den Stack gelegt
- Um beim nächsten Durchlauf die zuletzt verwendete Regel zu bekommen, wird die Methode peek des Stacks verwendet. Wir wollen die letzte Regel nur ansehen und nicht entfernen, daher peek statt pop.

Aufgabe 4

(Aufgabe 4 … bitte scrollen)

algebra line

sorts int, real, bool, point, segment, line

ops

makePoint:	real × real	→ point
makeSegment:	point × point	→ segment
createLine:	~~line × segment~~	→ line
append:	line × segment	→ line
prepend:	segment × line	→ line
update:	line × int × bool × point	→ line
getLength:	segment	→ real
getLength:	line	→ real
getStartEndDistance	line	→ real
getAngle	segment × segment	→ real

sets $point = \{(x, y) \mid x, y \in real\}$

$segment = \{S \mid S \in \{[start\ Ende] \mid start, Ende \in point \wedge start \neq Ende\}$

$line = \{S_1 \cup S_2 \ldots \cup S_n \mid n \in \mathbb{N},\ S \in segment \text{ mit } S_i.Ende = S_{i+1}.Start\}$

functions

makePoint(x, y) $= (x, y)$

makeSegment$(point_1, point_2)$ $= [point_1\ point_2] = S$

createLine$()$ $= \{\}$

append$(line, S)$ $= \{S\}$ falls $line = \emptyset$

$= \{S_1 \cup \ldots \cup S_n \cup S_{n+1}\}$ falls $S_n.Ende = S_{n+1}.Start$ wobei $S = S_{n+1}$

$= \{S_1 \cup \ldots \cup S_n \cup makeSegment(S_n.Ende, S_{n+1}.Start) \cup S_{n+1}\}$ falls $S_n.Ende \neq S_{n+1}.Start$, wobei $S_{n+1} = S$

$= \{ S_1 \cup \dots S_n \cup makeSegment(S_{n+1}.Ende, S_{n+1}.Start$
$\}$ falls $S_n.Ende = S_{n+1}.start$ wobei
$S_{n+1} = S$

prepend$(S, line)$ $= \{S\}$ falls $Line = \emptyset$
$= \{S_1 \cup S_2 \cup \dots \cup S_n\}$ falls $S_1.Ende = S_2.Start$
wobei $S_1 = S$; Indizes des Segments die bereits in
Line enthaltes sind müssen um 1 erhöht werden
$= \{S_1 \cup makeSegment(S_1.Ende, S_2.Start) \cup S_2 \dots \cup S_n\}$
falls $S_1.Ende \neq S_2.Start$, wobei $S_1 = S$; Indizes
hinter dem eingefügten um 1 erhöhen
$= \{makeSegment(S_1.Ende, S_2.Start) \cup \dots \cup S_n)\}$
falls $S_1.start = S_2.start$, wobei $S_1 = S$; Indizes
müssen erhöht werden

getLength(S) $= \sqrt{(S.Start.x - S.Ende.x)^2 + (S.start.y - S.Ende.y)^2}$

getLength$(Line)$ $= getLength(S_1) + \dots + getLength(S_n)$, wobei S_1
das erste und S_n das letzte Segment des Linien-
zuges ist

getStartEndDistance $= 0$ falls $Line = \emptyset$, sonst
$= getLength(makeSegment(S_1.Start, S_n.Ende))$
wobei S_1 erstes und S_n letztes Element der
Linie

end line.

getAngle(S_1, S_2) = $\perp$ falls $S_1.\text{Ende} \neq S_2.\text{Start}$
$\vee$ getLength(S_1) = 0
$\vee$ getLength(S_2) = 0

sonst:

$$= \cos^{-1}\left(\frac{\left| \begin{pmatrix} S_1.\text{Ende}.x - S_1.\text{Start}.x \\ S_1.\text{Ende}.y - S_1.\text{Start}.y \end{pmatrix} \cdot \begin{pmatrix} S_2.\text{Ende}.x - S_2.\text{Start}.x \\ S_2.\text{Ende}.y - S_2.\text{Start}.y \end{pmatrix} \right|}{\left\| \begin{pmatrix} S_1.\text{Ende}.x - S_1.\text{Start}.x \\ S_1.\text{Ende}.y - S_1.\text{Ende}.y \end{pmatrix} \right\| \cdot \left\| \begin{pmatrix} S_2.\text{Ende}.x - S_2.\text{Start}.x \\ S_2.\text{Ende}.y - S_2.\text{Start}.y \end{pmatrix} \right\|} \right)$$

wegen: Winkelberechnung zwischen 2 Vektoren

update(line, x, bool, point) = line falls $x < 1$ oder $x > n$

$= \{ S_1 \cup \dots \cup [S_x.\text{Start Point}] \cup \dots \cup S_n \}$ falls bool = true

$= \{ S_1 \cup \dots \cup [\text{Point } S_x.\text{Ende}] \cup \dots \cup S_n \}$ falls bool = false