Übungsblatt 11

Aufgabe 1 - Lambda-Ausdrücke

Gegeben sind die Klasse `IntValues` und die beiden Interfaces `IntIntFunction` und `IntBoolFunction`.

```
public class IntValues {
	private int[] data;
	public IntValues( int[] p ) { data = p; }
	public int compute( IntIntFunction f, IntBoolFunction b ) {
		int comp = 0;
		for ( int i=0; i<data.length && b.test( data[i] ); i++ ) {
			comp += f.apply( data[i] );
		}
		return comp;
	}
}
```

```
public interface IntIntFunction {
    int apply( int p );
}
```

```
public interface IntBoolFunction {
    boolean test( int p );
}
```

Die Referenz `iv` verweist jeweils auf ein `IntValues`-Objekt. Ergänze jeweils einen Lambda-Ausdruck als Argument für den Aufruf von `compute`.

- Die Summe aller Werte im Attribut `data` bis zu ersten Auftreten des Wertes `0` soll der Variablen `result` zugewiesen werden.

```
int result = iv.compute( [          ] );
```

- Die Anzahl aller Werte im Attribut `data` bis zu ersten Auftreten eines ganzzahligen Vielfachen von `10` soll der Variablen `result` zugewiesen werden.

```
int result = iv.compute( [          ] );
```

- Die Anzahl aller positiven Werte im Attribut `data` bis zu ersten Auftreten eines negativen Wertes soll der Variablen `result` zugewiesen werden.

```
int result = iv.compute( [          ] );
```

- Die Anzahl aller geraden Werte im Attribut `data` bis zu ersten Auftreten eines Wertes aus dem Intervall zwischen einschließlich 3 und einschließlich 17 soll der Variablen `result` zugewiesen werden.

```
int result = iv.compute( [          ] );
```

Aufgabe 2 - Klasse FlexibleTree

Die aus der Vorlesung bekannte generische Klasse `BinarySearchTree<T ...>` implementiert einen einfachen binären Suchbaum, der für jedes über die `compareTo`-Methode von `T` unterscheidbare Objekt nur registrieren kann, ob es im Baum vorhanden ist oder nicht. Der ebenfalls aus der Vorlesung bekannte und im Rahmen Komprimierung eingesetzte Suchbaum der Klasse `CharacterSearchTree` ist so angelegt, dass er mit einem einem `HuffmanTriple`- Objekt kooperiert und dadurch die Häufigkeit der im Baum abgelegten Zeichen zählen und deren Kodierung ablegen und zurückgeben kann. In dieser Aufgabe sollst Du nun eine allgemeinere Implementierung eines generischen binären Suchbaums erstellen, mit dessen Hilfe auch die Anforderungen erfüllt werden können, die sich aus der Komprimierung nach dem Algorithmus von Huffman ergeben. Nimm dazu die unten beschriebene Implementierung vor.

Vorgegeben sind die inneren Interfaces `Counter` und `CounterStrategy`.

Das Interface `Counter` gibt drei Methoden vor, die sich auf einen Zähler beziehen:

- `void increment()` erhöht den Zähler.
- `void decrement()` vermindert den Zähler.
- `int getValue()` liefert den Wert des Zählers.

Das Interface `CounterStrategy` gibt nur eine Methode vor:

- `Counter generateCounter()` liefert ein zur Klasse `Counter` kompatibles Objekt.

Diese beiden Klassen sollen dazu genutzt werden, das Zählen der gleichartigen Objekte innerhalb des Baums zu organisieren. Alle dem Baum hinzugefügten Knoten sollen über das gleiche `CounterStrategy`-Objekt eigene `Counter`-Objekte erzeugen, die innerhalb des Knotens das Zählen übernehmen. Über die `increment`- und `decrement`-Methoden kann dann implementiert werden, wie mit dem mehrfachem Einfügen des gleichen Inhalts oder dem Entfernen eines Inhalts umgegangen werden soll.

Vervollständigen Sie nun die generische Klasse `FlexibleTree<T ...>` mit den folgende Methoden.
Beachten Sie, dass der Vergleich von Inhalten immer mit der `compareTo`-Methode von `T` erfolgen soll.

- `FlexibleTree( CounterStrategy s )` legt einen leeren Baum an und weist ihm eine Strategie zum Zählen zu, die dann benutzt wird, um einen passenden Zähler zu erzeugen.
- `void add( T t )` nimmt einen Inhalt `t` neu im Baum auf oder erhöht – sofern `t` schon vorhanden ist – den entsprechenden Zähler.
- `boolean isEmpty()` gibt `true` zurück, wenn der ausführende Knoten keinen Inhalt besitzt.
- `boolean isLeaf()` gibt `true` zurück, wenn der ausführende Knoten ein Blatt ist.
- `boolean contains( T t )` gibt `true` zurück, wenn der aktuelle Stand des Zählers für den Inhalt `t` positiv ist.
- `int getQuantity( T t )` gibt den aktuellen Stand des Zählers für den Inhalt `t` zurück.
- `T get( T t )` gibt dasjenige im Baum gespeicherte Objekt zurück, das gemäß der `compareTo` -Methode gleich `t` ist.
- `void delete ( T t )` vermindert den entsprechenden Zähler, löscht aber keinen Knoten aus dem Baum.
- `void show()` zeigt alle Inhalte an, deren Zähler größer als 0 sind.
- `int size()` gibt die Anzahl der Knoten zurück, deren Zähler größer als 0 sind.

Testen Sie Ihre Implementierung mit vier Zählstrategien:

- `MultiCount` ändert für jedes Einfügen eines Inhalts den Zähler für diesen Inhalt. Beim Löschen soll reduziert, aber der Wert 0 nie unterschritten werden. Dieses Verfahren würde das Nachbilden des `CharacterSearchTree` erlauben.
- `DeficitCount` ändert für jedes Einfügen eines Inhalts den Zähler für diesen Inhalt. Beim Aufruf von `delete` können auch negative Werte erzeugt werden, die dann ein *Defizit* anzeigen.
- `SimpleCount` soll gleiche Inhalte nicht unterscheiden, es werden also beim Zählen nur die Werte 0 und 1 unterschieden. Mit dieser Strategie verhält sich ein `FlexibleTree` wie ein `BinarySearchTree`.
- `HistoryCount` soll wie `SimpleCount` gleiche Inhalte nicht unterscheiden und nur die Werte -1 oder 1 annehmen. Die Methode `getQuantity` liefert dadurch nur dann den Wert 0, wenn der Knoten noch nie enthalten war, und -1, wenn er vorhanden war und anschließend gelöscht wurde.