Übung 5: Sharing Objects: Immutable TreeSet

In dieser Übung lernen Sie, wie Sie immutable Datenstrukturen implementieren können. Im Kontext der funktionalen Programmierung bezeichnet man diese Strukturen auch als persistent [1].

Wir beginnen indem wir eine immutable Singly Linked List analysieren. Wenn Sie bereits funktional programmieren können, dürfen Sie die folgenden Erklärungen gerne überspringen.

```
public interface IList<E> {

  E head();
  IList<E> tail();
  boolean isEmpty();

  default IList<E> prepend(E e) {
    return new IList<E>() {
      public E head() { return e; }
      public IList<E> tail() { return IList.this; }
      public boolean isEmpty() { return false; }
    };
  }

  public static <E> IList<E> empty() {
    return new IList<E>() {
      public E head() { throw new IllegalStateException("head of Nil"); }
      public IList<E> tail() { throw new IllegalStateException("tail of Nil"); }
      public boolean isEmpty() { return true; }
    };
  }

}
```

Eine Liste besteht aus einem Head (dem vordersten Element) und dem Tail (dem Rest, wiederum eine Liste). Entsprechend kann man auf einer Liste head() aufrufen um an das vorderste Element zu kommen. Mit dem Aufruf von tail() bekommt man den hinteren Teil der List (alles ausser head).

Bei einer rekursiven Definition benötigt man einen Anfang (Base Case). Mittels IList.empty() erzeugt man eine solche leere Liste. Mit der Methode prepend kann nun ein Element vorne in die Liste eingefügt werden.

Hinzufügen von Elementen:

```
IList<Integer> l_e = IList.empty();              // Nil
IList<Integer> l_1e = l_e.prepend(1);            // [1 -> Nil]
IList<Integer> l_21e = l_1e.prepend(2);          // [2 -> [1 -> Nil]]
```

Abfragen von Elementen:

```
l_21e.head() == 2
l_21e.tail() == l_1e
l_21e.tail().head() == 1
```

Hinweis: Die obige Implementierung verzichtet auf explizite Subklassen. Es wäre auch möglich gewesen, die zwei Subklassen (Nil und Cons [2]) explizit zu modellieren.

[1] http://en.wikipedia.org/wiki/Persistent_data_structure
[2] http://en.wikipedia.org/wiki/Cons

Nun sind Sie an der Reihe. Implementieren Sie ein ImmutableTreeSet. Ein TreeSet ist eine Menge (jedes Element kommt nur einmal vor) implementiert als Binary Search Tree.

Hier ist das Interface:

```
public interface ITreeSet<E extends Comparable<E>> {
    ITreeSet<E> insert(E e);
    boolean contains(E e);
    boolean isEmpty();
}
```

Mit insert wird ein neues Element eingefügt, aber nur falls es noch nicht drin ist. Der Return Wert ist entsprechend entweder ein neues TreeSet, das zusätzlich das eingefügte Element nun enthält oder immer noch das ursprüngliche TreeSet, falls das Element schon drin war.

Aufgabe:
Implementieren Sie das obige Interface. Die Lösung muss strikt immutable sein. Nur final Referenzen sind erlaubt.

Hinweise:

- Es ist nicht notwendig jeweils den ganzen Tree zu kopieren!
- Da die Elemente verglichen werden darf das Element null nicht in den Baum eingefügt werden.
- Mit Java fühlen sich derartige Datenstrukturen wenig elegant an. Das ändert sich aber, sobald wir im Unterricht auf Scala umsteigen.
- Falls Sie mehr wissen möchten, empfehle ich Purely Functional Data Structures von Chris Okasaki (http:// www.cs.cmu.edu/~rwh/theses/okasaki.pdf)

Abgabe: 1./2. April 2019