Aufgaben mit dem TreeEditor

1. Installation

Auf dem Datei-Server finden Sie das Programmier-Projekt *ch.fhnw.algd2.treeeditor*. Importieren Sie dieses in Eclipse.

Der *TreeEditor* ist eine Java-Applikation, die binäre Bäume darstellen und manipulieren kann. Sie können beim Start der Applikation als Parameter angeben, welche Tree-Klasse benutzt werden soll. Beim Starten der Applikation über eine Kommandozeile z.B. so:

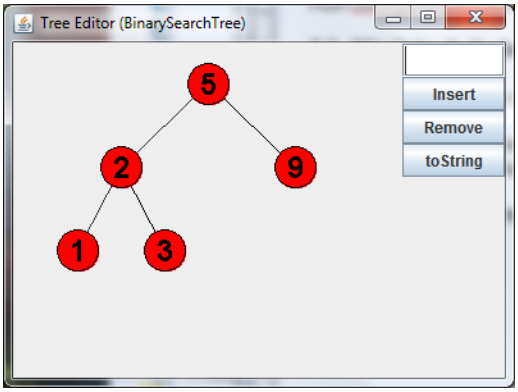
```
java TreeEditor BinarySearchTree
```

Der *Gradle* Task `run` enthält dieses Start-Kommando für die ebenfalls vorbereitete Klasse `BinarySearchTree`.

Sie können auch eine eigene Tree-Klasse programmieren. Damit sie vom *TreeEditor* verwendet werden kann, muss sie das Interface `ch.fhnw.algd2.treeeditor.base.Tree` implementieren. Beim Start des *TreeEditors* muss der vollständige Name als Parameter angegeben werden. Unsere vorbereiteten Klasse `BinarySearchTree` ist deswegen der Einfachheit halber im Default Package.

2. Aufgaben für Binäre Suchbäume

Ergänzen Sie die Klasse `BinarySearchTree` wie folgt.



a) Implementieren Sie die noch leere Methode `insert`. Eingefügt wird nach Set-Semantik.

   Sie können die Methode mit dem Editor selbst ausprobieren. Darüber hinaus finden Sie ein paar vorbereitete Tests im *test*-Ordner in der Klasse `Test_Insert`.

b) Implementieren Sie die noch leere Methode `search`.

   Sie finden ein paar vorbereitete Tests im *test*-Ordner in der Klasse `Test_Search`.

c) Implementieren Sie die noch leere Methode `remove`.

   Der *TreeEditor* ruft `remove` auf, wenn Sie einen Baumknoten mit der Maus selektieren und dann den *Remove*-Knopf drücken. Der Schlüssel im selektierten Knoten wird als Parameter übergeben.

   Sie können die Methode mit dem Editor selbst ausprobieren. Darüber hinaus finden Sie ein paar vorbereitete Tests im *test*-Ordner in der Klasse `Test_Remove`.

d) Implementieren Sie die noch leere Methode `toString`. Dabei soll eine String-Repräsentation des ganzen Baumes erzeugt werden.

   Beispiel: Für den rechts abgebildeten Baum sähe die String-Repräsentation so aus:

   `[[[1]2[3]]5[9]]`

   Teilbäume sind also jeweils von eckigen Klammern eingeschlossen. Welche Art der Baumtraversierung verwenden Sie?

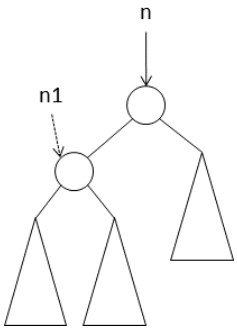
e) Implementieren Sie die noch leere Methode `height`

   Sie finden ein paar vorbereitete Tests im *test*-Ordner in der Klasse `Test_Height`.

Aufgabe 3 s. Rückseite

3. Aufgaben für AVL-Bäume

a) Fügen Sie der Klasse `BinarySearchTree` eigene Methoden für die AVL-Rotationen hinzu:




```
private Node<K, E> rotateR(Node<K, E> n) { ... }
private Node<K, E> rotateL(Node<K, E> n) { ... }
```

Hinweise zum Vorgehen:

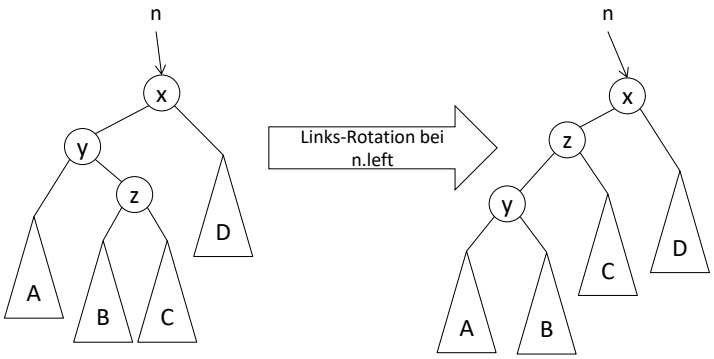
- Beginnen Sie mit `rotateR`.
- Sie müssen eine Folge von Referenz-Zuweisungen programmieren. Beginnen Sie damit, eine Referenz auf den linken Nachfolger von `n` an eine neue lokale Variable `n1` zuzuweisen.
- Nun wird die Referenz auf den linken Teilbaum in `n` (vorübergehend) nicht mehr benötigt und Sie können diesem Speicherort eine neue Referenz zuweisen, nämlich die, welche dort nach Abschluss der Rotation gespeichert sein soll.
- Danach ist wieder ein Speicherort frei geworden, den Sie neu beschreiben können. Fahren Sie so fort, bis die neue Struktur erreicht ist. Das Verfahren funktioniert also ähnlich, wie die *swap*-Operation von zwei Array-Elementen mit einer Hilfsvariablen.
- Der Rückgabewert der Methode ist die (neue) Wurzel des rotierten Teilbaums.
- Die Implementierung von `rotateL` lässt sich spiegelsymmetrisch aus jener von `rotateR` herleiten. Sie müssen nur konsequent `left` und `right` vertauschen.

b) Fügen Sie der Klasse weitere eigene Methoden für die AVL-Doppel-Rotationen hinzu:

```
private Node<K, E> rotateRL(Node<K, E> n) { ... }
private Node<K, E> rotateLR(Node<K, E> n) { ... }
```

Hinweise zum Vorgehen:

- Diese *Doppelrotationen* lassen sich – dem Namen entsprechend – als eine Folge von zwei der bereits für a) programmierten Einzelrotationen programmieren. Z.B. führt eine Links-Rotation bei `n.left` zur Vorbedingung dafür, dass eine Rechts-Rotation bei `n` selbst zum gewünschten Ziel führt. Diese Methoden lassen sich also mit lediglich zwei Aufrufen programmieren.



c) Modifizieren Sie die bestehende Methode `insert` so, dass der Baum ausgeglichen bleibt.

Einfachere (langsame) Variante: Die Höhe eines beliebigen Teilbaums können Sie auch mit der vorhandenen Methode `height(Node<K, E> t)` berechnen lassen.[1]

Um den Baum nach dem Einfügen ausgeglichen zu halten, muss auf dem „Rückweg der Rekursion“ jeweils die Ausgeglichenheit geprüft werden. Ist der Teilbaum nicht mehr ausgeglichen, kann dies mit einer der vier zuvor programmierten Rotationsoperationen behoben werden.

Herausfordernde (effiziente) Variante: Erweitern Sie die Klasse Node um ein Feld balance. Schreiben Sie eine rekursive Hilfsmethode `upin(Node<K, E> p)`, um die Balancefaktoren zu aktualisieren und Rotationen aufzurufen.

[1] Für am Detail Interessierte: Dieses Verfahren funktioniert, könnte aber effizienter implementiert werden, indem man in den `Node`-Objekten die Höhe des „in ihnen verwurzelten“ Teilbaums speichert. Diese Werte können beim rekursiven Wiederaufstieg in `insert` und `remove` jeweils leicht aktualisiert werden. Die Methode `height` kann dann einfach diesen Wert zurückgeben und muss nicht jedes Mal den Teilbaum traversieren (Aufwand $O(1)$ anstelle von $O(\log n)$).

d) (*herausfordernd*) Ändern Sie die Methode `remove` so, dass der Baum ausgeglichen bleibt. Hier muss man vor allem im Fall 3 der im Leitprogramm diskutierten Situationen aufpassen. Die zur Verfügung gestellten Tests funktionieren dann nicht mehr, weil sie von fixen Baumstrukturen ausgehen.