Allgemeines Programmierpraktikum **Sommersemester 2023**
Dr. Henrik Brosenne Georg-August-Universität Göttingen
Institut für Informatik

Übung 03

~~**Das tagging muss vor dem Abgabetermin Mo., 18.05., 8 Uhr** erfolgen.~~
Das *tagging* muss vor dem Abgabetermin Mo., 22.05., 8 Uhr erfolgen.

Am 18.05. ist **Himmelfahrt**.

Testate ab Mo., 22.05.

Vorbereitung

Prüfsumme und Testat

Bitte verwenden Sie ihr GitLab Projekt, das Sie für die vorherigen Übungen erstellt haben, auch für diese und alle weiteren Übungen. Folgen Sie der gleichen Abgabeprozedur (*tagging*).

Java

Informieren Sie sich in der *Java Platform, Standard Edition 11 API Specification* `https://docs.oracle.com/en/java/javase/11/docs/api/index.html` über die typisierte Schnittstelle `java.lang.Comparable<T>`, die typisierte abstrakte Klasse `java.util.AbstractCollection<E>`, die typisierte Klasse `java.util.Stack<E>` und die Klasse `java.util.Random`.

Allgemein

1. Verwenden Sie `ant` zum automatisierten Übersetzen des Quelltext und zum Erzeugen der Dokumentation. Nach dem Übersetzen befinden sich die `java`-, die `class`-Dateien und die Dokumentation in unterschiedlichen Verzeichnissen.

   Beim Übersetzen sollten möglichst wenig Hinweise *unchecked or unsafe operations* auftreten.

2. Kommentieren Sie alle Klassen ausführlich. Erstellen Sie für die Klassen selbst und alle Attribute Dokumentationskommentare für `javadoc`. Erzeugen Sie mit `javadoc` eine API-Dokumentation, inklusive `private`-Attribute, der Klasse.

Aufgabe 1

Schnittstelle `Comparable<T>`

Implementieren Sie im Package `app.exercise.algebra` eine Klasse `CompRational`, die von der Klasse `Rational` von *Übung 02* erbt und die typisierte Schnittstelle `java.lang.Comparable<T>` implementiert.

Implementieren Sie die Methode `compareTo` der Schnittstelle `Comparable` wie in der API beschrieben. Stellen Sie sicher, dass `(x.compareTo(y)==0) == (x.equals(y))` für zwei beliebige Objekte `x`, `y` der Klasse `CompRational` gilt.

Aufgabe 2

Typisierter binärer Suchbaum

Weitere Erläuterungen finden Sie unten im Abschnitt *Binäre Suchbäumen*.

1. Programmieren Sie eine typisierte Klasse `BSTree`, die einen binären Suchbaum (siehe unten *Binäre Suchbäume*) realisiert und eine typisierte Klasse `BSTreeIterator` für einen zugehörigen Iterator.

   a) Die Knoten-Klasse des binären Suchbaums implementiert die folgende Schnittstelle `DrawableTreeElement`. Siehe im GitLab *uebung→uebung03-data*.

   ```
   package app.exercise.visualtree;

   public interface DrawableTreeElement<T> {
       public DrawableTreeElement<T> getLeft();
       public DrawableTreeElement<T> getRight();
       public boolean isRed();
       public T getValue();
   }
   ```

   Lassen Sie `isRed` einen beliebigen Wert zurückliefern.

   b) `BSTree` befindet sich im Package `app.exercise.adt` (*ADT = Abstract Data Type*), erbt von `java.util.AbstractCollection<E>` und speichert Referenzen auf Objekte von Klassen, die `java.lang.Comparable<T>` implementieren.

   Legen Sie die die Methoden so an, wie in der *API Specification* für `java.util.AbstractCollection<E>` gefordert.

   Implementieren Sie mindestens folgende nicht optionale Methoden.

   - `int size()`
   - `boolean contains(Object o)`

Die Methode `contains` benutzt die Struktur des binären Suchbaums, um zu prüfen ob das gesuchte Objekt enthalten ist. Siehe *Suchen/Einfügen* im Abschnitt *Binäre Suchbäumen*.

Implementieren Sie folgende optionale Methoden sinnvoll.

- `boolean add(E e)`
- `boolean remove(Object o)`

Alle weiteren erforderlichen, nicht implementierten Methoden lösen eine `UnsupportedOperationException` aus.

c) `BSTreeIterator` befindet sich im Package `app.exercise.adt` und implementiert die Schnittstelle `Iterator<E>`. Legen Sie die Methoden so an, wie in der *API Specification* für `java.util.Iterator<E>` gefordert.

Implementieren Sie folgende Methoden, sodass der Iterator den Baum in *inorder* Reihenfolge durchläuft, d.h. die Elemente in aufsteigender Ordnung zurückliefert.

- `boolean hasNext()`
- `E next()`
- `void remove()`

Implementieren Sie die Methode `Iterator<E> iterator()` in `BSTree`, die ein passendes Objekt von `BSTreeIterator` zurückliefert.

2. Schreiben Sie im Package `app.exercise.testing` einen ausführbare Klasse `BSTreeTester`, die eine Liste von rationalen Zahlen auf der Kommandozeile übergeben bekommt.

   - Enthält die Kommandozeile keine gerade Anzahl ganzer Zahlen wird eine passende Fehlermeldung ausgegeben.
   - Zwei, auf der Kommandozeile, aufeinander folgende Zahlen werden als Zähler und Nenner einer vergleichbaren rationalen Zahl (`CompRational`) interpretiert.
   - Speichern Sie alle übergebenen rationalen Zahlen in einem primären binären Suchbaum.
   - Erzeugen Sie aus dem primären binären Suchbaum mit `toArray()` ein Feld und geben Sie das Feld komponentenweise aus.
   - Speichern Sie die übergebenen rationalen Zahlen abwechselnd in zwei sekundären binären Suchbäumen.
   - Geben Sie die beiden sekundären binären Suchbäumen mit `toString()` aus.

- Geben Sie die Elemente des primären und der sekundären Bäume mit Hilfe des *in-order*-Iterators aus.
- Testen Sie mit `containsAll`, ob die sekundären Bäume im primären Baum enthalten sind.
- Testen Sie mit `contains`, ob die erste und letzte auf der Kommandzeile übergebene rationale Zahl im primären Bäume enthalten ist.

  Entfernen Sie mit `remove` die erste und letzte auf der Kommandzeile übergebene rationale Zahl aus dem primären Baum.
- Erzeugen Sie zufällig (`java.util.Random`) hundert rationale Zahlen, die zwischen der kleinsten und größten in der Liste enthaltenden Zahl liegen.

  Überprüfen Sie mit `contains`, ob die Zahlen jeweils im primären Baum enthalten sind. Entfernen Sie die enthalten Zahlen aus dem primären Baum.

  Hinweis

  Sei a/b das kleinste, c/d das größte Elemente und $0 < n < 1000$ eine zufällige ganze Zahl. Dann ist $a/b + n \cdot [(cb - ad)/(db \cdot 1000)]$ eine zufällige rationale Zahl, die zwischen dem kleinsten und größten Element liegt.
- Integrieren Sie die Visualisierung mit den `RedBlackTreeDrawer` in die Testklasse (siehe unten *Visualisierung*).

  Rufen Sie die Methode `RedBlackTreeDrawer.draw` nach jeder Veränderung (Einfügen, Löschen, etc.) am primären Baum auf.

Hinweis

Die Methoden

- `boolean containsAll(Collection<?> c)`
- `Object[] toArray()`
- `String toString()`

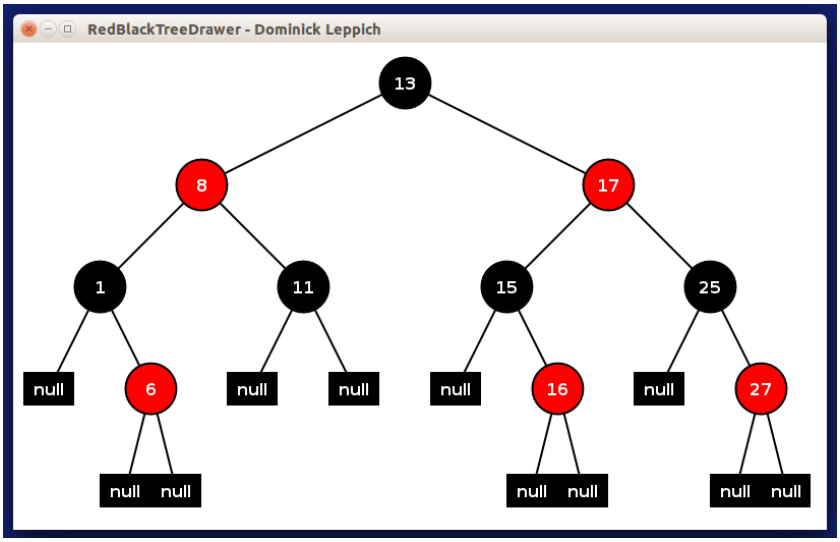
werden aus `java.util.AbstractCollection<E>` geerbt.

Alternative

Falls Sie die Klasse `CompRational` nicht implementiert haben, können Sie den Wert jeder rationalen Zahl als Gleitkommazahl (`Double`) bestimmen und damit arbeiten.

Visualisierung

Dominick Leppich hat ein Werkzeug zur Visualisierung von Rot-Schwarz-Bäumen programmiert, den `RedBlackTreeDrawer`. Dieses Werkzeug kann eingeschränkt auch für die Visualisierung von binären Suchbäumen verwendet werden. Zum Testen und für die Fehlersuche kann diese Visualisierung sehr hilfreich sein.



Lassen Sie die Knoten-Klasse Ihres binären Suchbaums die Schnittstelle `DrawableTreeElement` implementieren.

Erzeugen Sie in Ihrer Test-Klasse eine Instanz der Klasse `RedBlackTreeDrawer`, dabei muss die gleich Typisierung wie für die Knoten-Klasse Ihres binären Suchbaums verwendet werden, z.B. `Double`. Dann können Sie, indem Sie der Methode `draw` von `RedBlackTreeDrawer` einen beliebigen Knoten Ihres binären Suchbaums übergeben, den Baum, dessen Wurzel der übergebene Knoten ist, anzeigen lassen. Sie können die Methode auch mehrfach aufrufen.

```
RedBlackTreeDrawer<Double> visual = new RedBlackTreeDrawer<Double>();
...
visual.draw(...);
...
visual.draw(...);
```

Die Schnittstelle `DrawableTreeElement` und die Klasse `RedBlackTreeDrawer` sind im jar-Archive `RedBlackTreeDrawer.jar`. Siehe im Git *uebung→uebung03-data*.

Zum Übersetzen müssen Sie das Archiv `RedBlackTreeDrawer.jar` dem Klassenpfad (*classpath*) hinzufügen.

Beispiel

Im Ant-Buildfile kann der Klassenpfad mit dem Attribute `classpath` im Task `javac` gesetzt werden.

```
<property name="src" location="src"/>
<property name="build" location="build"/>
<property name="jarfile" location="RedBlackTreeDrawer.jar"/>
...
<javac srcdir="${src}"
       destdir="${build}"
       classpath="${jarfile}"
       includeantruntime="false"/>
```

Auch zum Ausführen muss der Klassenpfad angepasst werden.

Beispiel

Das Archiv `RedBlackTreeDrawer.jar` ist im aktuellen Verzeichnis hinterlegt und die `class`-Dateien sind im Verzeichnis `build` gespeichert. Dann kann die ausführbaren Klasse `BSTreeTester` im Packet `app.exercise.testing` wie folgt gestartet werden.

```
java -cp build:RedBlackTreeDrawer.jar
     app.exercise.testing.BSTreeTester
```

Binäre Suchbäume

In einem binären Suchbaum gelten folgende Eigenschaften.

- Jeder Knoten enthält ein Element der gespeicherten Menge und hat genau zwei, möglicherweise leere, Unterbäume. Einen leerer Unterbaum (`null`) nennt man Blatt.
- Es gibt einen ausgezeichneten Knoten, die Wurzel, der zu keinem Unterbaum eines Knotens gehört. Alle anderen Knoten gehören zu genau einem Unterbaum.
- Die Knoten des linken Unterbaums eines Knotens enthalten nur Elemente, die kleiner sind als das im Knoten selbst gespeicherte Element.
- Die Knoten des rechten Unterbaum eines Knotens enthalten nur Elemente, die größer sind als das im Knoten selbst gespeicherte Element.

Diese Eigenschaften gelten rekursiv für die Unterbäume und müssen z.B. bei Einfügeoperationen erhalten bleiben.

Suchen/Einfügen

Vorraussetzung, um eine Element in einen binären Suchbaum einzufügen, ist, das der Suchbaum dieses Element (bzw. ein identisches Element) nicht bereits enthält. Deshalb laufen die Suchen nach einem (vorhandenen) Element und das Finden der Einfügeposition für ein (neues) Element im wesentlichen gleich ab.

1. Der Laufknoten wird mit der Wurzel des Baumes initialisiert.
2. Es werden folgende Fälle und Unterfälle unterschieden.
   - Das gesuchte/einzufügende Element ist identisch mit dem Element am Laufknoten.

     Suchen Suche war erfolgreich und ist beendet.

     Einfügen Element ist bereits enthalten. Einfügen ist beendet.
   - Das gesuchte/einzufügende Element ist kleiner als das Element am Laufknoten.
     - Linker Unterbaum des Laufknotens ist leer.

       Suchen Suche war nicht erfolgreich und ist beendet.

       Einfügen Erzeuge einen neuen Knoten mit dem einzufügenden Element und zwei leeren Unterbäumen, der zum linken Unterbaums des Laufknotens wird. Einfügen ist beendet.
     - Linker Unterbaum des Laufknotens nicht ist leer. Setze den Laufknoten auf die Wurzel des linken Unterbaums. Führe die Suche bei 2. fort.
   - Das gesuchte/einzufügende Element ist größer als das Element am Laufknoten.
     - Rechter Unterbaum des Laufknotens ist leer.

       Suchen Suche war nicht erfolgreich und ist beendet.

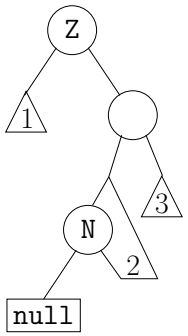
       Einfügen Erzeuge einen neuen Knoten mit dem einzufügenden Element und zwei leeren Unterbäumen, der zum rechten Unterbaums des Laufknotens wird. Einfügen ist beendet.
     - Rechter Unterbaum des Laufknotens ist nicht leer. Setze den Laufknoten auf die Wurzel des rechten Unterbaums. Führe die Suche bei 2. fort.

Entfernen

Man betrachtet den zu entfernenden Knoten (`Z`) und, falls er existiert, dessen Nachfolger (`N`). Bezogen auf das in `Z` gespeicherte Element ist `N` der Knoten, der das nächst größere Element enthält.

Hinweis

`N` findet man im rechten Unterbaum vom `Z`. Die Suche beginnt bei der Wurzel dieses Unterbaums und es wird in jedem durchlaufenen Knoten zum linken Unterbaum verzweigt, bis ein Knoten erreicht wird, dessen linker Unterbaum leer, d.h. ein Blatt ist.



Es werden folgende Fälle unterschieden.

1. Die Kinder von `Z` sind Blätter (`null`). Der Knoten `Z` wird gelöscht und durch ein Blatt ersetzt.

2. Genau ein Kind von `Z` ist ein Blatt. Der Knoten wird gelöscht und durch das Kind ersetzt, das kein Blatt ist.

3. Keins der Kinder von `Z` ist ein Blatt. Das Element von `N` wird in `Z` kopiert und `N` wird entfernt. Da mindestens eins der Kinder von `N` ein Blatt ist, kann `N` wie unter 1 oder 2 entfernt werden.