Übung 6 – Binärbäume

Arbeiten Sie im Skript das *Kapitel Bäume* durch. Suchen Sie sich weitere Informationen, wo Sie die Ausführungen nicht verstehen. Sprechen Sie mit Ihren Kommilitoninnen und Kommilitonen. Fragen Sie im Forum.
Lesen Sie die Aufgaben vollständig und markieren Sie sich zentrale Aspekte. Verwenden Sie **keine Klassen der Java API** außer als Test für Ihre Implementationen und beachten Sie konsistent die Zugriffrechte.

Aufgabe 1

Wir implementieren im Folgenden die Datenstruktur "Binärer Suchbaum" für `Integer`-Werte (d.h. die Klasse ist vorerst nicht generisch). Erstellen Sie dafür eine Klasse `IntSuchbaum`. Die Klasse soll folgende Methoden bereitstellen:

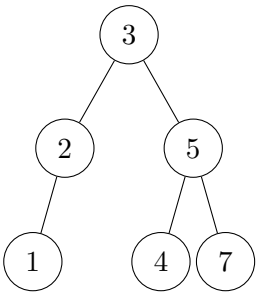
- `IntSuchbaum()`:
  Der Konstruktor erzeugt einen leeren Baum.
- `isEmpty()`:
  Die Methode gibt einen Wahrheitswert zurück, ob der Baum leer ist.
- `insert(Integer i)`:
  Die Methode fügt einen Wert in den Baum ein. Dabei wird darauf geachtet, dass die Regeln des Suchbaumes nicht verletzt werden und der Wert an der richtigen Stelle im Baum einfügt wird. Sollte das Element bereits im Baum enthalten sein wird der Wert nicht eingefügt.
- `contains(Integer i)`:
  Liefert einen Wahrheitswert, ob `i` im Suchbaum vorkommt.
- `toString()`:
  Die Methode liefert eine Darstellung des Baums als Zeichenkette der Form `((<links>)<wurzel>(<rechts>))` (ohne Leerzeichen). Darin stehen `<links>` und `<rechts>` für entsprechende Darstellungen des linken und des rechten Teilbaums und `wurzel` für die Darstellung des Inhaltes an der Wurzel. Ein leerer Baum wird durch einen leeren String dargestellt.
- `hoehe()`:
  Die Methode berechnet die Höhe des Baumes und gibt den Wert zurück. Sie sollten sich hierbei eine rekursive Hilfsmethode `hoehe(BaumEl e)` definieren, die die Höhe eines Baumelementes (`BaumEl`) berechnet.
- `size()`:
  Die Methode gibt die Anzahl der Werte im Baum als Ganzzahl zurück. Lösen Sie das Problem rekursiv ohne Instanzvariable.

Testen Sie Ihre Methoden mit JUnit-Tests.

Hinweis: Die Methoden außer `isEmpty()` sollten rekursiv implementiert werden, i.d.R. mittels einer rekursiven Hilfsmethode.

Beispiel: Die folgenden Anweisungen erzeugen den daneben skizzierten Baum.

```
IntSuchbaum baum;
baum = new IntSuchbaum();
baum.insert(3);
baum.insert(2);
baum.insert(5);
baum.insert(4);
baum.insert(1);
baum.insert(7);
```



Ein Aufruf von `toString()` liefert entsprechend den String `(((1)2)3((4)5(7)))`.

Aufgabe 2 (Theorie)

Wir betrachten im Folgenden zwei verschiedene Integer-Suchbäume `baum1` und `baum2` bzw. deren Aufbau und Inhalt. Skizzieren Sie wie die Bäume, die aus den folgenden Anweisungen enstehen, schematisch aussehen.

```
//baum1:

    baum1.insert(33);
    baum1.insert(7);
    baum1.insert(43);
    baum1.insert(1);
    baum1.insert(20);
    baum1.insert(31);
    baum1.insert(67);
```

```
//baum2:

    baum2.insert(1);
    baum2.insert(7);
    baum2.insert(20);
    baum2.insert(31);
    baum2.insert(33);
    baum2.insert(43);
    baum2.insert(67);
```

Nehmen Sie nun an, dass für jeden Suchbaum `contains(67)` aufgerufen wird. Was fällt Ihnen bezüglich Laufzeiten (zählen Sie die Schritte, die in contains durchgeführt werden müssten) auf?

Aufgabe 3

Erweitern Sie die Klasse `IntSuchbaum` um folgende Traversierungmethoden. Alle diese Methoden sollen eine neue `Folge<Integer>` zurückgeben. In der zurückgegebenen Folge stehen die Werte entsprechend der Reihenfolge der Traversierung.

- `preorder()`:
  Bei dieser Methode wird der Baum in preorder traversiert und die entsprechenden Inhalte in die Folge eingefügt.
- `inorder()`:
  Bei dieser Methode wird der Baum in inorder traversiert und die entsprechenden Inhalte in die Folge eingefügt.
- `postorder()`:
  Bei dieser Methode wird der Baum in postorder traversiert und die entsprechenden Inhalte in die Folge eingefügt.
- `breitensuche()`:
  Bei dieser Methode wird der Baum in einer Breitensuche traversiert und die entsprechenden Inhalte in die Folge eingefügt. Machen Sie sich zunächst mit Hilfe einer Skizze bewusst, wie Breitensuche in Binärbäumen funktioniert. Überlegen Sie sich im Anschluss daran welcher *abstrakte Datentyp*, den wir bereits kennengelernt haben, hier verwendet werden sollte um den Suchalgorithmus zu implementieren (Zwischenspeichern von noch zu bearbeitenden Teilbäumen - *FIFO*-Abfolge). Innerhalb der Methode können Sie eine konkrete Implementierung (Beispiel `<ADT>Mit<Datenstruktur>`) ihres abstrakten Datentypen aus der Vorwoche verwenden.

Beispiel: Für den Baum aus dem Beispiel in Aufgabe 1 liefern die Traversierungsmethoden Folgen mit folgendem Inhalt:

- `preorder`: `3 2 1 5 4 7`
- `inorder`: `1 2 3 4 5 7`
- `postorder`: `1 2 4 7 5 3`
- `breitensuche`: `3 2 5 1 4 7`

Erweitern Sie Ihre Testklasse für den `IntSuchbaum` um Tests für die Traversierungsmethoden. Schreiben Sie sich eine Hilfsmethode in der Sie testen, ob zwei `Folge<Integer>` gleich sind. Prüfen Sie, ob die Folgen gleich lang sind und iterieren Sie über die Elemente und prüfen ob diese gleich sind.

Aufgabe 4

Implementieren Sie in Iherer Klasse `IntSuchbaum` eine Methode `remove(Integer i)`, die den Wert `i` im Baum löscht. Damit keine Löcher entstehen, müssen die Werte aus den unteren Teilbäumen hochgezogen werden (**Hinweis**: Fallunterscheidung), dabei dürfen die Regeln des Suchbaums nicht verletzt werden. Orientieren Sie sich hierzu an den Erläuterungen aus dem Skript. Testen Sie Ihre Methode sinnvoll mit JUnit-Tests für die möglichen Fälle und überprüfen Sie die Ergebnisse mit Hilfe der `toString()`-Methode.

Hinweis: Machen Sie sich eine Skizze der verschiedenen möglichen Fälle *bevor* Sie die Methode implementieren!

Zusatzaufgabe (Programmierübung Projekt Euler)

Die Primfaktoren von 13195 sind $5, 7, 13$ und 29.

Was ist der größte Primfaktor der Zahl 600851475143?

`https://projekteuler.de/problems/3`
Lösen Sie das Problem mit einer Klassenmethode `maxPrimfactor()` in einer Klasse `Euler3`. Geben Sie den größten Primfaktor zurück und testen Sie Ihre Implementierung für verschiedene Werte. Welchen Datenyp verwenden Sie? Warum eignet sich Integer nicht?