Prof. Dr. Christian Kohls
Alexander Dobrynin, M.Sc.

May 30, 2022
SoSe 2022

Algorithmen und Programmierung 2
Praktikum 4 - Bäume und Testing

Dieses Übungsblatt beschäftigt sich mit den theoretischen Grundlagen und der Implementierung eines binären Baums (*binary tree*). Sie werden die Basistruktur, grundlegende Methoden und komplexere Methoden implementieren. Anschließend schreiben Sie automatisierte Tests für Ihren binären Baum mit JUnit.

Für die Aufgaben sind die folgenden Videos aus der Playlist auf YouTube relevant:

- "54. Bäume - Überblick" bis "59. Funktionen höherer Ordnung".
- "41. Automatisiert Testen / JUnit"

Bzw. folgende Kapitel aus dem Buch Programmieren lernen mit Kotlin:

- Kapitel "29 Testen und Optimieren".
- Kapitel "12.13 Funktionen höherer Ordnung" bis inklusive Kapitel "12.13.1 Funktionen, die Funktionen als Parameter akzeptieren". Hinweis: Hier werden Funktionen höherer Ordnung als solches beschrieben.
- Kapitel "25 Funktionen höherer Ordnung für Datensammlungen". Hinweis: Hier werden Funktionen höherer Ordnung u.a. auf Listen beschrieben.

Contents

1 Grundlagen zu binären Bäumen **2**
1.1 Theorie . . . 3
1.2 Binären Baum erzeugen . . . 3

2 Methoden auf binären Bäumen implementieren **4**
2.1 Basismethoden . . . 4
2.2 Die map Funktion . . . 4
2.3 Die add Funktion . . . 5

3 Testen mit JUnit **7**
3.1 Der erste Test . . . 7
3.2 Alle Methoden testen . . . 7

4 Anhang: Anleitung für JUnit 5 in IntelliJ **8**

1 Grundlagen zu binären Bäumen

Gegeben sei die Basistruktur eines binären Baums, der generisch über den Typparameter `A` ist.:

```
// Basisstruktur
sealed class Tree<A>

private data class Node<A>(
  val value: A,
  val left: Tree<A>,
  val right: Tree<A>
) : Tree<A>()

private object Empty : Tree<Nothing>()

// Hilfsfunktionen zum Erstellen von Bäumen
fun <A> emptyTree(): Tree<A> = Empty as Tree<A>

fun <A> treeNode(
  value: A,
  left: Tree<A> = emptyTree(),
  right: Tree<A> = emptyTree()
): Tree<A> = Node(value, left, right)
```

Zunächst ein paar Anmerkungen zu diesem Code:

- Eine `sealed class` ist wie eine abstrakte Klasse, mit der Einschränkung, dass alle Unterklassen in einer Datei definiert werden müssen. Somit kann man kontrollieren, welche und wie viele Unterklassen überhaupt möglich sind. In diesem Fall sind es nur zwei: `Node` und `Empty`.
- `Node` beschreibt einen Knoten mit einem Element vom Typ `A` und einen rechten und linken Teilbaum. Ein Teilbaum kann wiederum ein Knoten oder ein leerer Baum sein.
- `Empty` beschreibt einen leeren Baum.
- Die Klassen `Node` und `Empty` sollen **niemals direkt** verwendet werden. Verwenden Sie die Funktionen `emptyTree`, um einen leeren Baum und `treeNode`, um einen Knoten mit entsprechenden Elementen zu erzeugen. Das hat ähnliche Gründe wie bei der Listenklasse in Kotlin. Hier verwenden Sie auch die Hilfsfunktion `listOf`, `mutableListOf` oder `emptyList`, um eine entsprechende Liste zu erzeugen.

1.1 Theorie

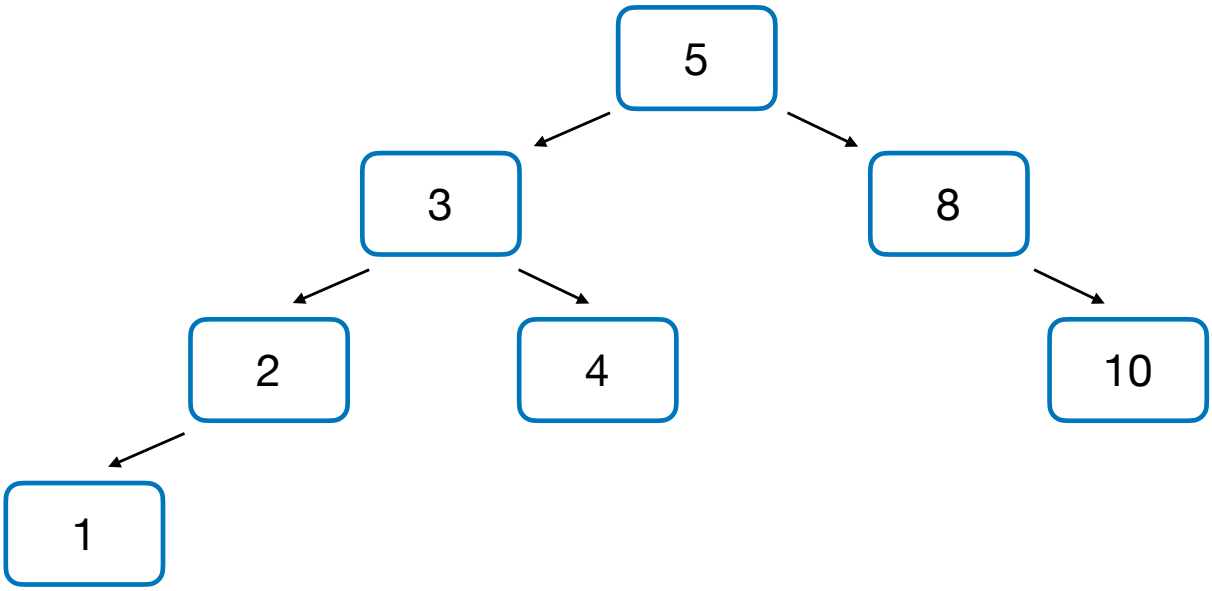
Machen Sie sich mit dem Konzept eines binären Baums vertraut, indem Sie folgende Fragen beantworten:

- Ein binärer Baum ist eine rekursive Datenstruktur. Was bedeutet das?
- Wofür werden binäre Bäume verwendet? Nennen Sie einen typischen Anwendungsfall.

1.2 Binären Baum erzeugen

Erzeugen Sie den in Abbildung 1 dargestellten Binärbaum im Code. Verwenden Sie hierfür die Hilfsfunktionen `treeNode` und `emptyTree`.

Figure 1: Visualisierung eines Binärbaums.



2 Methoden auf binären Bäumen implementieren

In dieser Aufgabe werden Sie Methoden auf binären Bäumen implementieren. Definieren Sie alle Methoden als abstrakte Funktionen innerhalb der `sealed class Tree<A>`. Die Implementierung machen Sie dann in den jeweiligen Unterklassen (wie in den Screencasts)[1].

2.1 Basismethoden

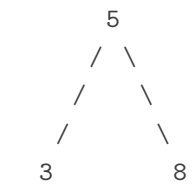
- `fun isEmpty(): Boolean` => Gibt zurück, ob der Baum leer ist oder nicht.
- `fun size(): Int` => Gibt die Anzahl der Knoten zurück. Der Baum in Aufgabe 1.2 hat beispielsweise 7 Knoten.
- `fun depth(): Int` => Gibt die Tiefe des Baums zurück. Der Baum in Aufgabe 1.2 hat beispielsweise eine Tiefe von 4.

2.2 Die map Funktion

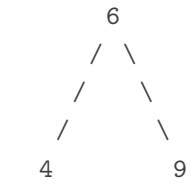
Die map Funktion hat die folgende Signatur:

```
fun <B> map(transform: (A) -> B): Tree<B>
```

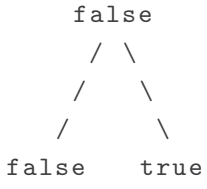
Und gibt einen neuen Baum zurück, bei dem jeder Wert mit der `transform` Funktion verändert wurde.
Gegeben sei beispielsweise folgender Baum:



Wenn der Baum mit einer Funktion gemappt wird, die jeden Wert um 1 erhöht, dann gibt die `map` Funktion folgenden Baum zurück:



Wenn der ursprüngliche Baum mit einer Funktion gemappt wird, die zurückgibt, ob der Wert gerade ist oder nicht, dann gibt die `map` Funktion folgenden Baum zurück:



[1] Alternativ können Sie die Implementierung auch direkt in der `sealed class Tree<A>` machen. Hierfür müssen die Methoden dann nicht mehr abstrakt sein. Verwenden Sie hierfür das pattern matching mit dem `when` Ausdruck.

2.3 Die add Funktion

In dieser Aufgabe sollen sie es ermöglichen, Elemente dem Baum hinzuzufügen.
Achtung: Implementieren Sie die add Funktion direkt in der Klasse `Tree`. Die Unterscheidung zwischen `Node` und `Empty` machen Sie über den `when` Ausdruck. Hier die Vorlage dazu:

```
sealed class Tree<A> {
  // ... andere Methoden

  fun add(value: A, compare: (A, A) -> Int): Tree<A> =
    when (this) {
      Empty -> TODO("Implementierung für Empty")
      is Node -> TODO("Implementierung für Node")
    }
}
```

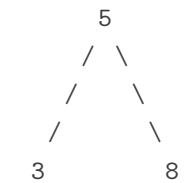
Die `add` Funktion gibt einen neuen binären Baum zurück, bei dem der Wert `value` an der **richtigen Stelle** (sortiert) eingefügt wurde. Die Regeln hierfür sind wie folgt:

- Wenn einem leeren Baum ein Element hinzugefügt wird, wird ein Knoten mit dem Wert `value` zurückgegeben.
- Wenn `value` kleiner oder gleich dem Wert des aktuellen Knotens ist, wird im linken Teilbaum eingefügt.
- Wenn `value` größer als der Wert des aktuellen Knotens ist, wird im rechten Teilbaum eingefügt.

Die `compare` Funktion wird verwendet, um zwei Werte vom Typ `A` zu vergleichen[2]. Diese Funktion sollen Sie verwenden, um den aktuellen Wert im Knoten mit dem neu einzufügenden Wert zu vergleichen. Auf dieser Grundlage entscheiden Sie, ob weiter links oder rechts eingefügt werden soll. Zurückgegeben wird ein `Int`, welcher folgende Bedeutung hat:

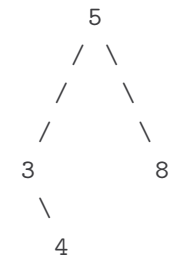
- -1: der linke Wert ist kleiner als der rechte Wert
- 1: der linke Wert ist größer als der rechte Wert
- 0: beide Werte sind gleich groß

Diese `compare` Funktion folgt den gleichen Regeln, wie die `compareTo` Methode in dem `Comparable` Interface aus Java.
Gegeben sei beispielsweise folgender Baum:



[2]Sie benötigen die `compare` Funktion, weil Vergleichsoperatoren wie <= oder > nicht auf Werten vom Typ `A` funktionieren.

Wenn man diesem binären Baum einen Knoten mit dem Wert 4 hinzufügen möchte, wird zunächst überprüft, ob die 4 kleiner als die 5 ist. Wenn ja, wird im linken Teilbaum weitergeschaut. Jetzt wird überprüft, ob die 4 kleiner ist als die 3. Wenn nein, wird im rechten Teilbaum weitergeschaut. Da hier ein leerer Baum ist, wird ein Knoten mit dem Wert 4 eingefügt. Der daraus resultierende binäre Baum sieht wie folgt aus:



3 Testen mit JUnit

In dieser Aufgabe sollen Sie alle Methoden, die Sie in Aufgabe 2 implementiert haben, mit dem JUnit Framework automatisiert testen. Hierfür müssen Sie zumnächst JUnit einrichten. Sie können sich entweder den Screencast 29 Testen und Optimieren dazu anschauen, oder der Anleitung im Anhang folgen. Die folgenden Codebeispiele verwenden JUnit 5. Daher sollten Sie das ebenfalls verwenden, um Verwirrungen zu vermeiden.

3.1 Der erste Test

Erzeugen Sie in dem Ordner *test/kotlin* eine neue Klasse namens `TreeTest` (diesen Schritt können Sie sich sparen, wenn Sie der Anleitung im Anhang gefolgt sind). Ersetzen Sie die Klassendefinition mit folgendem Code:

```
import org.junit.jupiter.api.Assertions.assertEquals
import org.junit.jupiter.api.Test

class TreeTest {

  @Test
  fun `an empty tree should be empty`() {
    val tree = emptyTree<Int>()
    assertEquals(true, tree.isEmpty())
  }
}
```

Jede mit `@Test` markierte Methode wird als Unit Test angesehen. Machen Sie sich mit der `assertEquals` Methode vertraut. Schauen Sie sich diesen Test an und versuchen Sie zu verstehen, was dort getestet wird.
Sie können nun den Test ausführen, indem Sie den grünen Play Button links neben der Klasse drücken. IntelliJ lässt alle Tests laufen. Wenn ein Test funktioniert hat, wird dieser grün darstellt. Wenn nicht, dann ist der Test rot und Sie sehen eine entsprechende Fehlermeldung. Das Ziel ist es, dass jeder Test grün durchläuft.
Ändern Sie in dem Aufruf `assertEquals` das `true` zu einem `false`. Führen Sie den Test aus. Dieser sollte nun scheitern.

3.2 Alle Methoden testen

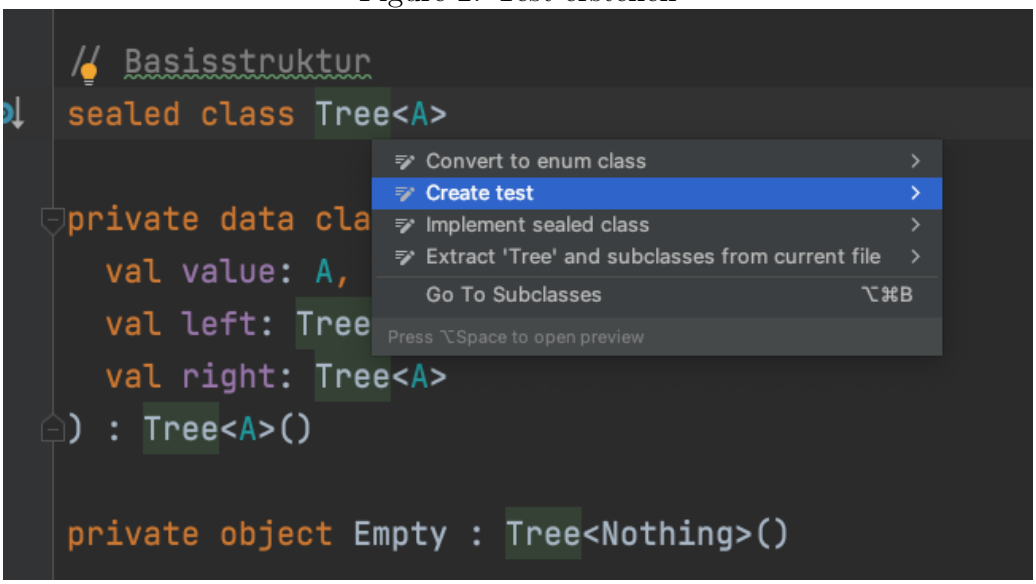
Für eine Methode können Sie mehrere Tests schreiben. In Aufgabe 3.1 haben wir nur den positiven Fall getestet (ein leerer Baum soll leer sein). Man kann noch einen weiteren Test für die `isEmpty` Methode schreiben, die testet, ob ein Baum mit einem Knoten nicht leer ist.

- Überlegen Sie sich, wie viele und welche Tests es für jede Methode gibt.
- Implementieren Sie für jede Methode **mindestens einen** Test.

4 Anhang: Anleitung für JUnit 5 in IntelliJ

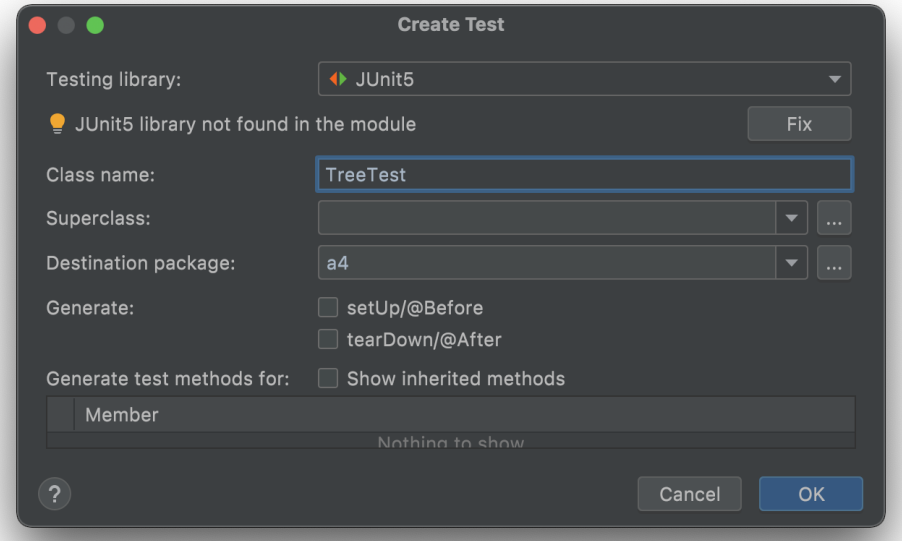
Gehen Sie mit dem Cursor auf die Klasse Tree und drücken Sie *ALT+ENTER* (Windows) oder *OPTION+ENTER* (Mac). Nun erscheint das in Abbildung 2 dargestellte Menü. Wählen Sie *Create Test* bzw. *Test erstellen* aus.

Figure 2: Test erstellen



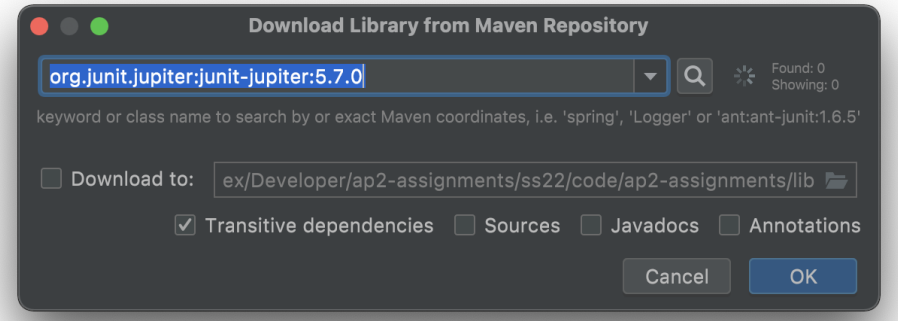
Danach öffnet sich das in Abbildung 3 gezeigte Fenster. Dort wählen Sie oben *JUnit 5* aus und drücken auf *Fix* bzw *Beheben*.

Figure 3: JUnit 5 auswählen



Im nächsten Fenster soll JUnit 5 heruntergeladen werden. Bestätigen Sie mit *OK* (Abbildung 4). Das kann ggf. etwas dauern.

Figure 4: JUnit 5 herunterladen



Zum Schluss erscheint das Fenster in Abbildung 5. Die Warnung aus Abbildung 3 sollte weg sein. Drücken Sie auf *OK*. Eine Klasse namens `TreeTest` wurde für Sie im richtigen Ordner (*test/kotlin*) erstellt. Sie können jetzt die Aufgabe weiter bearbeiten.

Figure 5: Testklasse erstellen