Informatik

2

Hinweise zu den Übungsaufgaben

Prof. Dr.-Ing. Holger Vogelsang

Sommersemester 2017

Don’t panic

12

Bonusaufgabe

Die folgenden Aufgaben dienen nur der weiteren Einarbeitung in Java. Ihre Bearbeitung ist freiwillig möglich, wobei Sie teilweise Musterlösungen im Ilias finden.

12.1 Sieb des Eratosthenes

Notwendige Vorkenntnisse zur Lösung der Aufgabe: Arrays, imperative Sprachelemente, Datentypen.
Berechnen Sie die Primzahlen, die kleiner als eine vorgebene konstante Zahl `MAX` sind. Verwenden Sie dazu den Algorithmus, der unter dem Namen „Sieb des Eratosthenes“ bekannt ist:

1. Schreiben Sie alle Zahlen von 2 bis MAX auf.
2. Streichen Sie dann alle Vielfachen von 2, dann von 3, dann von 5 (wieso nicht 4?), . . . usw.
3. Bis zu welcher Zahl müssen Sie den Algorithmus laufen lassen?
4. Alle jetzt nicht gestrichenen Zahlen sind Primzahlen. Geben Sie diese auf dem Bildschirm aus.

Hinweis zur Implementierung: Verwenden Sie ein Array von Wahrheitswerten (`boolean`). Der Index des Feldes entspricht der Zahl. Verwenden Sie die folgenden Methoden zur Zeitmessung.

- Startzeitpunkt: `long start = System.currentTimeMillis();`
- Endzeitpunkt: `long end = System.currentTimeMillis();`
- Zeitdifferenz in msec: `end - start;`

Messen Sie bitte nur die Zeit für die Berechnung (ohne Ausgabe). Um zu testen, ob Ihr Algorithmus korrekt ist, fügen Sie die folgende Methode in Ihren Code ein und rufen diese mit jeder Zahl sowie Ihrem Ergebnis (`true` = ist Primzahl, `false` = ist keine Primzahl) auf. Die Methode `testPrim` gibt nur im Fehlerfall eine Meldung aus („no news is good news“). Sie müssen die Funktionsweise der Methode nicht verstehen.

```
/**
  * Test auf Primzahl (Überprüfung der Implementierung).
  * @param numberToTest Die auf Primzahl zu testende Zahl.
  * @param testPrediction Vermutung des Studenten.
  *                       <ul>
  *                         <li> true: Ist eine Primzahl.
  *                         <li> false: Ist keine Primzahl.
  *                       </ul>
  */
  private static void testPrim(int numberToTest,
                               boolean testPrediction) {
    BigInteger bi = new BigInteger("" + numberToTest);
    if (bi.isProbablePrime(100) != testPrediction) {
      System.err.println("Falsch: Die Zahl " + numberToTest
                         + " ist "
                         + (testPrediction ? " keine "
                                           : " eine ")
                         + "Primzahl");
    }
  }
```

Fügen Sie die folgende Zeile als erste in Ihre Datei ein:

```
import java.math.BigInteger;
```

12.2 Ballspiel

Notwendige Vorkenntnisse zur Lösung der Aufgabe: Arrays, imperative Sprachelemente, Datentypen.
In dieser Aufgabe stellt ein Array ein kleines Spielfeld das, über das ein Ball bewegt wird. Das Spielfeld ist von einer Mauer umgeben.

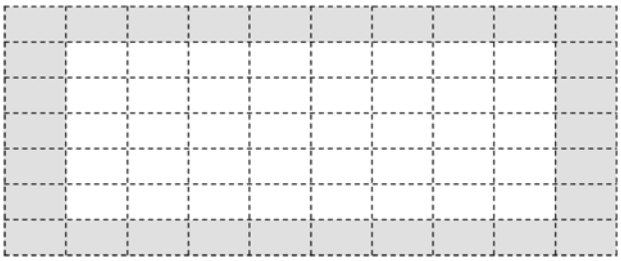

Abbildung 12.1: Aufbau des Spielfeldes

Dabei gelten die folgenden Regeln:

- Ein Element des Arrays ist entweder frei oder durch den Rand belegt.
- Die Größe des Spielfeldes soll beliebig veränderbar sein.

Gehen Sie bei der Entwicklung wie folgt vor:

1. Schreiben Sie eine Methode `init`, die zwei Parameter übergeben bekommt:
   - `int width`: Breite des Feldes
   - `int height`: Höhe des Feldes

2. Die Methode `init` erzeugt aus diesen Daten ein Spielfeld und trägt es als Attribut der Klasse ein. Die Koordinate (0,0) liegt in der linken oberen Ecke.
3. Jetzt wird der Ball an die linke, obere Ecke gesetzt und diagonal in Richtung rechte, untere Ecke bewegt.
4. Stößt der Ball auf den Rand, so wird er gemäß Einfallswinkel = Ausfallswinkel bewegt.
5. Schreiben Sie eine Methode `move`, die den Ball bewegt:
   - Fügen Sie zunächst die folgenden Attribute zur Klasse hinzu:
     - `ballX`: X-Position des Balles
     - `ballY`: Y-Position des Balles
     - `direction`: Richtung des Balles. Überlegen Sie sich, wie Sie die Richtung ausdrücken wollen und welche Möglichkeiten es hier gibt.
   - Die Methode `move` verwendet die Attribute, um die neue Ballposition zu bestimmen.
   - Anschließend trägt sie die neuen Werte in die drei Attribute ein.
6. Gehen Sie dabei von der Vereinfachung aus, dass in der Ecke des Spielfeldes der Ball immer direkt zurückbewegt wird.
7. Schreiben Sie eine weitere Methode `print`, die das Spielfeld in einer einfachen, grafischen Darstellung auf dem Bildschirm ausgibt:
   - Rand: `#`
   - Ball: `*`
   - Freies Feld: (Leerzeichen)
8. Schreiben Sie eine `main`-Methode, die ein Spielfeld erzeugt und den Ball 20 mal bewegt. Nach jeder Bewegung geben Sie das Spielfeld aus.
9. Kommentieren Sie Ihre Lösung!

12.3 Palindrom-Test

Notwendige Vorkenntnisse zur Lösung der Aufgabe: Arrays, Strings, imperative Sprachelemente, Datentypen.
In dieser Aufgaben sollen Sie prüfen, ob ein Wort ein Palindrom ist. Ein Palindrom ist ein Wort, das vorwärts und rückwärts gelesen dasselbe ist, wobei nicht zwischen Klein- und Großbuchstaben unterschieden wird. Beispiele: „Anna“, „Lagerregal“
Um die Aufgabe interessanter zu gestalten, sollen auch Palindromsätze erkannt werden. Damit sind Sätze gemeint, die nach Entfernung von Leerzeichen vorwärts und rückwärts gelesen identisch sind. Satzzeichen wie „.“ sollen nicht vokommen. Beispiel: „Erika feuert nur untreue Fakire“
Die folgenden Wörter und Sätze müssen Sie prüfen. Der Kommentar gibt an, ob es sich im ein Palindrom handelt oder nicht.

```
private static String[] palindromTests = {
   "Anna",                              // true
   "nie",                               // false
   "Reittier",                          // true
   "",                                  // false
   null,                                // false
   "Nasenloch",                         // false
   "Ada",                               // true
   "Rentner",                           // true
   "Anna hetzte Hanna",                 // true
   "Informatik 2",                      // false
   "Renate bittet Tibetaner",           // true
   "Erika feuert nur untreue Fakire" // true
};
```

Die Klasse `Character` erleichert Ihnen den Umgang mit Zeichen. Sie können mit der Methode `char toLowerCase(char)` ein Zeichen in einen Kleinbuchstaben umwandeln und mit `boolean isWhitespace(char)` prüfen, ob ein Zeichen ein unsichtbares Zeichen wie ein Leerzeichen, ein Tabulator usw. ist.

12.4 Komplexe Zahlen

Erstellen Sie eine Klasse `Complex` zur Verwaltung komplexer Zahlen. Die Klasse hat die privaten Attribute `real` und `imag` sowie die öffentlichen, arithmetischen Operationen `add` zur Addition einer weiteren komplexen Zahl und `absoluteValue` zur Berechnen des Betrages der komplexen Zahl.
Hinweise zu komplexen Zahlen:

- Mathematisches Modell mit zwei vorgegebenen komplexen Zahlen z_1 und z_2:
  $z_1 = real_1 + i * imag_1$
  $z_2 = real_2 + i * imag_2$
- Addition:
  $z_1 + z_2 := real_1 + real_2 + i * (imag_1 + imag_2)$
- Betrag:
  $|z1| = \sqrt{(real_1^2 + imag_1^2)}$

Definieren Sie die Addition als Methode mit nur einem Parameter! Definieren Sie Methoden zum Zugriff auf die privaten Daten und globale Funktionen, die diese Methoden aufrufen. Deklarieren Sie Konstruktoren mit und ohne Parameter.

12.5 Taschenrechner

Notwendige Vorkenntnisse zur Lösung der Aufgabe: Klassen, imperative Sprachelemente, Datentypen.
Schreiben Sie eine Klasse `Calculator`, die die Funktionalität eines Taschenrechners umsetzt. Der Rechner soll die folgenden Methoden besitzen:

- `public void add(double value)`: Addiert `value` zur Zwischensumme.
- `public void sub(double value)`: Subtrahiert `value` von der Zwischensumme.
- `public void neg()`: Negiert das Zwischenergebnis. Das Resultat wird das neue Zwischenergebnis.

- `public void clearResult()`: Löscht das intern gespeicherte Zwischenergebnis.
- `public double getResult()`: Liest das Zwischenergebnis aus.
- `public long getResultAsLong()`: Liest das Zwischenergebnis aus und liefert dieses als mathematisch korrekt gerundete `long`-Zahl zurück.
- `public void ln()`: Berechnet den natürlichen Logarithmus des Zwischenergebnisses und legt das Resultat als neues Zwischenergebnis ab.
- `public void log()`: Berechnet den Logarithmus zur Basis 10 des Zwischenergebnisses und legt das Resultat als neues Zwischenergebnis ab.
- `public void pow(double exp)`: Berechnet $result^{exp}$ und legt das Resultat als neues Zwischenergebnis ab.
- `public void sin()`: Berechnet den Sinus des Zwischenergebnisses und legt das Resultat als neues Zwischenergebnis ab.
- `public void round()`: Rundet das Zwischenergebnis in eine ganze Zahl.
- `public void or(double arg)`: Führt eine bitweise Oder-Verknüpfung von Argument und Zwischenergebnis durch. Das Resultat wird das neue Zwischenergebnis. Runden Sie beide Zahlen vorher!
- `public void and(double arg)`: Führt eine bitweise Und-Verknüpfung von Argument und Zwischenergebnis durch. Das Resultat wird das neue Zwischenergebnis. Runden Sie beide Zahlen vorher!
- `public void not()`: Invertiert alle Bits des Zwischenergebnisses. Das Resultat wird das neue Zwischenergebnis. Runden Sie dazu die Zahl vorher!

Suchen Sie für die Berechnung der mathematischen Ausdrücke passende Methoden aus der JDK-Dokumentation und schreiben Sie eine kleine `main`-Methode für die Klasse `Calculator`, um dessen einwandfreie Funktionsweise zu testen. Wenn Sie wollen, dann können Sie die Korrektheit noch besser mit JUnit-Tests sicherstellen. Denken Sie daran, die Methoden mit Javadoc-Kommentaren zu versehen.
Erweitern Sie Ihren Taschenrechner um einen Ergebnisspeicher:

- `public void memoryClear()`: Löscht den Speicher.
- `public void memoryAdd()`: Addiert die Zwischensumme zum Speicher.
- `public void memorySave()`: Legt die Zwischensumme im Speicher ab.
- `public double memoryRecall()`: Liest den Wert des Speichers aus.

Testen Sie auch diese neue Funktionalität, indem Sie die `main`-Methode erweitern oder neue JUnit-Tests erstellen.

12.6 Restaurantsimulation

Notwendige Vorkenntnisse zur Lösung der Aufgabe: Klassen, Pakete, imperative Sprachelemente, Datentypen, `ArrayList`-Klasse.
Die Lösung der Aufgabe soll die Sitzplatzbelegung in einem Restaurant simulieren. Dieses Restaurant wird von Besuchergruppen betreten. Das Restaurant weist jeder Gruppe einen Tisch zu. Nachdem alle Mitglieder der Gruppe gegessen haben, verlässt die Gruppe das Restaurant, und die Sitzplätze sind wieder frei. Modellierung:

- Ein Restaurant besteht aus mehreren Tischen. Alles andere ist für diese Simulation nicht relevant.
- Jeder Tisch hat eine bestimmte Anzahl von Stühlen.
- Ein Stuhl ist frei oder belegt.
- Eine Besuchergruppe besteht aus einer bestimmten Anzahl Personen.

Eine Besuchergruppe hat immer das folgende Verhalten:

- Restaurant betreten
- Platzsuche gemäß unten aufgeführter Strategie
- Aufenthalt im Restaurant
- Restaurant verlassen

Formal läuft die Platzsuche einer Gruppe mit n Personen so ab, wobei sich Gruppen niemals auftrennen:

1. Versuche, einen noch völlig leeren Tisch mit genau n Stühlen zu finden. Der Tisch muss also die exakt passende Stuhlzahl aufweisen.
2. Versuche, einen völlig leeren Tisch mit mindestens n Stühlen zu finden. Der Tisch darf somit mehr Stühle als Besucher in der Gruppe haben.
3. Suche einen Tisch, an dem noch mindestens n freien Stühle zu finden sind. An diesem Tisch dürfen bereits auch andere Gruppen sitzen. Wenn es mehrere solcher Tische gibt, so muss der genommen werden, an dem nach der Platzierung der Gruppe so wenig Stühle wie möglich frei bleiben.

Vorgehensweise zur Implementierung der Lösung:

1. Finden der Klassen, Implementierung des Verhaltens.
2. Test zur Verifikation der Funktionsweise anhand einer Testklasse. Aufgabe der Testklasse:
   - Erzeugen eines Restaurants mit zufälliger Tischanzahl und zufälliger Sitzplatzanzahl je Tisch.
   - Erzeugen einer Anzahl von Gruppen mit zufälligen Größen.
   - Platzieren der Gruppen. Beachten Sie Sonderfälle (z.B. kein Tisch mehr frei).
   - Gruppen verlassen das Restaurant.

   Nur die Testklasse darf Bildschirmausgaben vornehmen.

Erstellen Sie die Dokumentationskommentare für Ihre Klassen!

12.7 Iterator für die doppelt verkettete Liste

Implementieren Sie einen Iterator für Ihre doppelt verkettete Liste (siehe Pflichtaufgaben). Dieser Iterator kann im einfachsten Fall die existierende Schnittstelle `Iterator` implementieren. Da sich aber doppelt verkettete Listen sehr effizient vorwärts und rückwärts durchlaufen lassen, sollte Ihre Iterator am besten die vorhandene Schnittstelle `ListIterator` implementieren. Sehen Sie sich dazu diese API der Schnittstelle im Paket `java.util` an. Das Erzeugen des Iterators erfolgt durch Aufruf der Methode `listIterator()` Ihrer Listenklasse. Werfen Sie dazu einen Blick auf die vorhandene Klasse `java.util.LinkedList` des JDK. Dort ist der `ListIterator` ebenfalls vorhanden.

12.8 Programmexport als lauffähige JAR-Datei

Notwendige Vorkenntnisse zur Lösung der Aufgabe: Eclipse-Bedienung, einfache Java-Kenntnisse aus „Informatik 1“.
JAR-Dateien sind Archive („**J**va **AR**chive“) im ZIP-Format, die neben Klassen als Bytecode auch andere Ressourcen wie Bilder usw. enthalten können. Interessant ist insbesondere die Möglichkeit, innerhalb einer sogenannten Manifest-Datei die Klasse im Archiv angeben zu können, die die `main`-Methode des Projektes enthält. Solch eine JAR-Datei kann im Betriebssystem direkt z.B. durch einen Doppelklick ausgewählt werden, um die `main`-Methode aufzurufen. Die Manifest-Datei muss `MANIFEST.MF` heißen und im Verzeichnis `META-INF` des Archivs liegen. Führen Sie die folgenden Schritte in Eclipse aus, um eine ausführbare JAR-Datei zu erstellen:

- Rechtsklick auf das Projekt, „Export“ auswählen
- im Dialog „Java“ und „Runnable JAR File“ auswählen
- Wenn das Projekt in Eclipse mindestens einmal gestartet wurde, lässt sich hier die Start-Konfiguration des Projektes auswählen. In dieser Konfiguration stehen sowohl Informationen wie der Name der Klasse, die die `main`-Methode besitzt, als auch eventuell abhängige Bibliotheken. Die Information zum Start wird verwendet, um die Manifest-Datei anzulegen.
- Unter „Export Destination“ wird angegeben, wie die JAR-Datei heißen und in welches Verzeichnis sie geschrieben werden soll.
- In der Auswahl zum „Library Handling“ wählen Sie für die Beispiele aus der Vorlesung am besten „Package required libraries into generated JAR“. Dabei werden JAR-Dateien, die das Projekt benötigt, in die erzeugte JAR-Datei kopiert und stehen beim Start direkt zur Verfügung.

Testen Sie, ob sich die erzeugte JAR-Datei ausführen lässt. Sehen Sie sich den Inhalt des erzeugten Archivs an (z.B. mit „7Zip“). Interessant ist hier insbesondere die Manifest-Datei.

12.9 Datenspeicherung

Erweitern Sie Ihre Lösung aus der Zeitmessungs-Aufgabe und speichern Sie Ihre Messergebnisse in einer reinen Text-Datei (CSV-Format). Die Datei soll folgenden zeilenweisen Aufbau besitzen:

```
Art der Messung,Datenstrukturname,Zeit in Millisekunden
```

Beispiel:

```
Einfügen,java.util.HashSet,20
```

Jede Messung erhält eine eigene Zeile.

12.10 Dynamisches Hashverfahren

In der Vorlesung „Informatik 2“ wurden zwei verschiedene Hashverfahren vorgestellt. Weitere sind aus Zeitgründen nicht besprochen worden. Diese Aufgabe stellt einen sehr einfachen Ansatz für dynamische Hashtabellen vor, der auch in ähnlicher Form in der

Java-Klasse `HashMap` Einsatz findet. In der Praxis existieren weitere Verfahren, die aus Gründen des Aufwandes hier nicht betrachtet werden sollen.

12.10.1 Einfügen

Überschreitet der Füllgrad der Hashtabelle einen vorgegebenen Wert, dann wird das Array der Hashtabelle doppelt so groß. Dadurch sinkt der Füllgrad wieder. In Abbildung 12.2 ist eine Situation einer dynamischen Hashtabelle zu sehen.

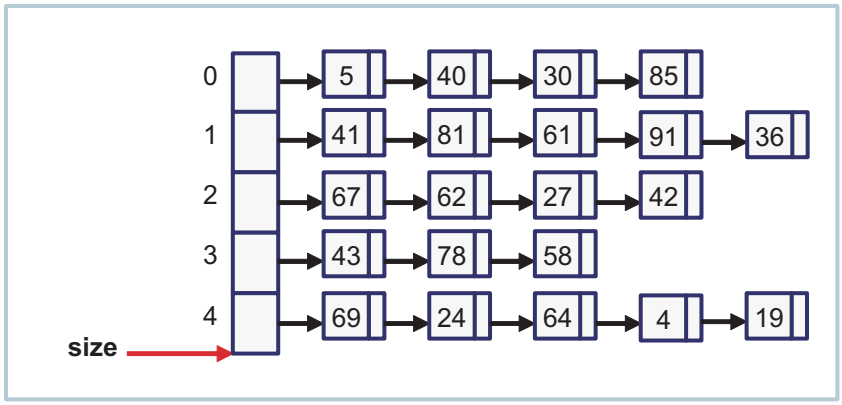

Abbildung 12.2: Aktuelle Situation einer dynamischen Hashtabelle

Es werden hier im Bild nur `Integer`-Werte abelegt, um die Grafik übersichtlich zu halten. `size` beinhaltet die Größe der Hashtabelle. Soweit unterscheidet sich dieser Ansatz nicht von dem, der in der Vorlesung vorgestellt wurde: Neue Einträge werden einfach an die entsprechenden Listen angehängt. Bei einem hohen Füllgrad würden sich also sehr lange Listen bilden. Um dieses Problem zu umgehen, wird ein maximaler Füllgrad `max` der Tabelle festgelegt. In diesem Beispiel hier beträgt er 400%. Im Schnitt sollen also nicht mehr als vier Werte je Tabellenfeld abgelegt werden. Realistischer ist ein Füllgrad von 50–75%.
Wird ein weiterer Wert eingefügt, dann befinden sich 21 Einträge in der Tabelle, sie ist überfüllt (Füllgrad 420%). Somit ist es Zeit für eine Reorganisation mit dem folgenden prinzipiellen Ablauf:

1. Der Speicherplatz der Hashtabelle wird verdoppelt.
2. Die Einträge aller Listen der alten Hashtabelle werden neu einsortiert. Dazu wird ihr Hashwert wie bisher verwendet. Allerdings erfolgt jetzt die Modulo-Berechnung so, dass die doppelt so große Tabelle berücksichtigt wird. Anhand dieses Wertes erfolgt die Einsortierung in die neue Tabelle.
3. Die alte Tabelle wird nicht mehr benötigt und durch die neue ersetzt.
4. `size` wird verdoppelt.

Die Abbildung 12.3 zeigt die Tabelle nach der Reorganisation, wobei die verschobenen Einträge durchgestrichen dargestellt sind.

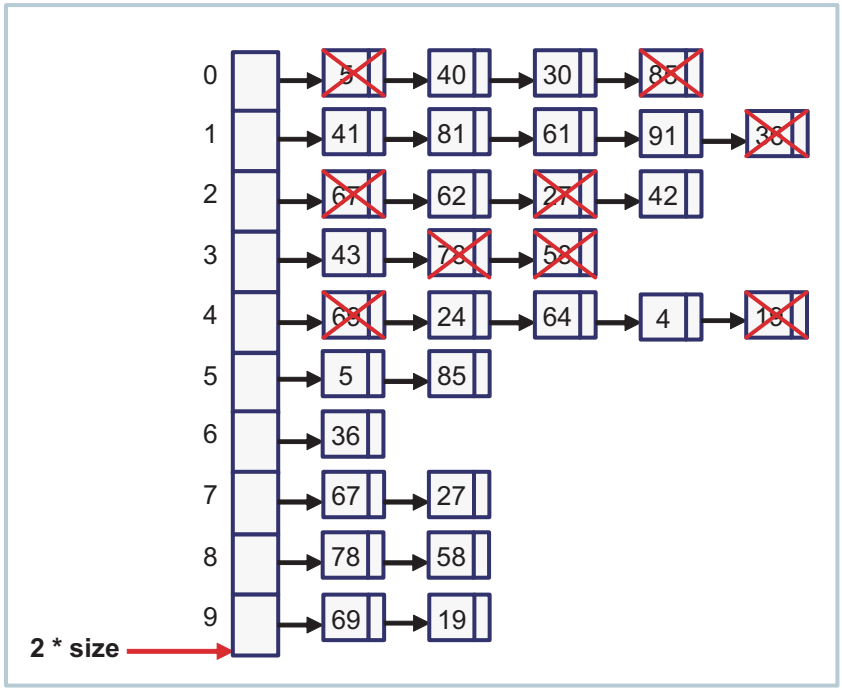

Abbildung 12.3: Dynamische Hashtabelle nach der Reorganisation

12.10.2 Löschen

Beim Löschen dagegen soll nicht auf Einhalten einer Untergrenze untersucht werden. Hier wird nicht mehr benötigter Platz nicht wieder freigegeben.

12.10.3 Aufgabenstellung

Sie können für die Implementierung Hashtabelle gerne das Fragment aus der Vorlesung „Informatik 2“ verwenden und erweitern.

Struktur

Jedes Feld der Hashtabelle besitzt eine verkettete Liste zur Aufnahme der Daten.

Konstruktoren

Die Klasse erhält zwei Konstruktoren:

1. Ein Konstruktor erhält lediglich die Initialgröße der Tabelle als Parameter und verwendet als Schwellwert die Konstante 0.75.
2. Ein weiterer Konstruktor besitzt zwei Parameter. Es handelt sich dabei um die Initialgröße sowie den Schwellwert. Prüfen Sie diese Parameter mit Assertions auf sinnvolle Werte.

Methoden

Die Klasse besitzt die folgenden Methoden:

- `put`: Fügt einen Wert mit dem angegebenen Schlüssel ein. Duplikate sind nicht erlaubt.

- `get`: Liest den Wert zum übergebenen Schlüssel wieder aus. Diese Methode existiert bereits.
- `remove`: Löscht das Paar aus Schlüssel und Wert aus der Tabelle. Als Übergabeparameter erhält die Methode den Schlüssel der zu löschenden Daten. Die Methode gibt als Ergebnis den zum Schlüssel gehörigen Wert zurück. Existiert der Schlüssel nicht in der Tabelle, dann ist der Rückgabewert `null`.

Aus Effizienzgründen darf der Hashwert bei einer Reorganisation nicht neu berechnet werden. Statt dessen wird er zusammen mit Schlüssel und Wert in der Liste abgelegt. Nur die Modulo-Berechnung erfolgt stets erneut. Bauen Sie das Lösungsgerüst aus der Vorlesung so um, dass Sie statt eines Paars, bestehend aus Schlüssel und Wert, ein Tripel aus Schlüssel, Wert und Hashwert speichern. Die Listenelemente nehmen solche Tripel auf.

Testprogramm

Das Testprogramm überprüft alle Methoden eingehend. Schreiben Sie weiterhin eine Methode, die den Inhalt der Hashtabelle ausgibt. Diese Methode wird in einem Test vor und nach dem Einfügen bzw. Löschen aufgerufen.

12.11 Erweiterung des binären Suchbaums

In der Vorlesung wurde Ihnen bereits eine sehr einfache Teilimplementierung eines binären Suchbaumes gezeigt, dessen Implementierung Sie in den Beispielen zur Vorlesung unter dem Namen `SimpleMapMap` finden. Die Implementierung besitzt Methoden zum Einfügen und Suchen.

12.11.1 Löschen

Erweitern Sie die Klasse so, dass Sie auch Elemente aus dem Baum löschen können, indem Sie den zu entfernenden Schlüssel angeben.

12.11.2 Ausgabe als XML-Dokument

Schreiben Sie eine Methode, die den Baum in Form eines XML-Dokumentes ausgegeben kann. Dabei sollen auch die optischen Einrückungen stimmen. Der linke Nachfolger eines Knotens wird immer vor dessen rechtem Nachfolger ausgegeben. Verwenden Sie den Preorder-Durchlauf. Beispielausgabe:

```
<treemap>
  <node key="A" value="42">
    <node key="D" value="567"/>
    <node key="F" value="234">
      <node key="E" value="456"/>
      <node key="H" value="345"/>
    </node>
  </node>
</treemap>
```

Um die Einrückungen sehr einfach zu implementieren, übergeben Sie beim rekursiven Aufruf für die Ausgabe der Kindknoten die aktuelle Einrücktiefe. Diese nehmen die Kindknoten und erhöhen sie für ihre eigenen Ausgaben.

12.11.3 Iterator

Achtung: Diese Aufgabe ist wirklich sehr schwierig und nicht klausurrelevant. Erweitern Sie Ihre Baum-Klasse aus Aufgabe 12.11 so, dass Sie Iteratoren unterstützt. Hinweise:

- Sie müssen mit Hilfe der Iteratoren den Baum komplett sortiert (inorder) durchlaufen können.
- Eine rekursive Lösung ist nicht möglich, da der Iterator nach jedem gefundenen Knoten diesen zurückgibt.
- Verwenden Sie daher die in der Vorlesung aufgezeigte iterative Variante, die ein Stack-Objekt zum Zwischenspeichern der aktuellen Position verwendet.
- Jeder Iterator muss ein eigenes Stack-Objekt haben, da ja eventuell mehrere Iteratoren auf einem Baum operieren. Daher verwenden Sie den Stack als Attribut der Iterator-Klasse.