Technische Universität Braunschweig
Institut für Programmierung und Reaktive Systeme

Dr. Werner Struckmann
9. Mai 2017

Programmieren II

11. Übungsblatt

Hinweis: Auf diesem und den folgenden Übungsblättern finden Sie jeweils eine Pflichtaufgabe. Ihre Lösung der Pflichtaufgabe dieses Übungsblatts müssen Sie bis spätestens zum

21. Mai 2017 um 15 Uhr

auf der Web-Seite

`https://prog-abgabe.ips.cs.tu-bs.de`

abgeben. Beachten Sie, dass Sie Ihre Lösung auch Ihrem Tutor erläutern müssen.

Halten Sie sich bei der Programmierung an die in der Vorlesung vorgestellten Richtlinien zur Formatierung von Java-Programmen. Auf der Internetseite zu dieser Veranstaltung finden Sie eine Zusammenstellung dieser Richtlinien. Kommentieren Sie Ihre Lösung der Pflichtaufgabe. Der Kommentar muss Ihren Namen, Ihre Matrikelnummer und Ihre Übungsgruppe, sowie eine Beschreibung Ihrer Lösung enthalten. Auf der Abgabeseite finden Sie eine Möglichkeit, die Formatierung Ihrer Lösung zu checken.

Aufgabe 59: Der abstrakte Datentyp „Keller mit Elementen vom Typ T“ wird durch die generische Schnittstelle

```
interface Stack<T> {
  boolean isEmpty();
  T top() throws MyStackException;
  void push(T x);
  void pop() throws MyStackException;
}
```

beschrieben.

a) Geben Sie eine Klasse `LinkedListStack` an, die diese Schnittstelle durch eine verkettete Liste implementiert. Lösen Sie jeweils eine eigene Ausnahme aus, falls eine Operation nicht definiert ist. Realisieren Sie Ihre Implementierung ohne den Import fremder Klassen.

b) Testen Sie Ihre Implementierung, indem Sie eine Klasse `LinkedListStackTest` anlegen, mit der alle Methoden des Interfaces für die Datentypen `Integer` und `Character` mit JUnit getestet werden. Ihr Test soll dabei auch den Fehlerfall `top/pop` für einen leeren Keller enthalten.

c) Testen Sie anschließend Ihr Programm mit der folgenden Anwendung:

> Editoren besitzen häufig ein sog. *Löschsymbol*. Wenn beispielsweise das Zeichen `"#"` das Löschsymbol ist, dann wird die Zeichenkette `"abc#d##e"` zur Zeichenkette `"ae"`, denn das erste Löschsymbol löscht `"c"`, das zweite `"d"` und das dritte `"b"`.

Testen Sie Ihr Programm, indem Sie eine Zeichenkette aus einer Datei einlesen und das Ergebnis ausgeben. Falls die Datei die Zeichenkette `"abc#d##e"` enthält, soll also `"ae"` ausgegeben werden. Falls die Datei die Zeichenkette `"abc #d #e"` enthält, soll dementsprechend `"abcde"` ausgegeben werden. Verwenden Sie dafür den folgenden Algorithmus:

- Jedes Zeichen der Datei wird eingelesen.
- Zeichen, die nicht das Löschsymbol sind, werden in einem Keller des Datentyps `Character` gespeichert.
- Falls das Zeichen das Löschsymbol ist, wird das oberste Kellerzeichen gelöscht.
- Falls alle Zeichen eingelesen wurden, befindet sich die gesuchte Zeichenkette in umgekehrter Reihenfolge auf dem Keller. Geben Sie den Kellerinhalt in umgekehrter Reihenfolge aus. Verwenden Sie dazu einen zweiten Keller.

Wie Zeichen aus einer Datei eingelesen werden können, zeigt das folgende Beispielprogramm:

```
import java.io.*;
public class Test {
  public static void main(String[] args) {
    FileReader f;
    int c;
    try {
      f = new FileReader("input.txt");
      while ((c = f.read()) != -1) {
         System.out.print((char)c);
      }
      f.close();
    } catch (IOException e) {
      System.out.println("Dateifehler!");
    }
  }
}
```

d) Realisieren Sie die Klasse `ArrayStack`, die die Schnittstelle durch ein Feld implementiert.

Aufgabe 60: Der folgende Algorithmus überführt einen vollständig geklammerten, arithmetischen Ausdruck, der in Infix-Notation vorliegt und nur binäre Operatoren enthält, in Postfix-Notation:

> Der Algorithmus legt zunächst einen Keller für `Character` an. Dann wird der arithmetische Ausdruck von links nach rechts abgearbeitet. Dabei wird
> - eine öffnende Klammer ignoriert,
> - eine Zahl in die Ausgabe geschrieben und
> - ein Operator auf dem Keller abgelegt.
>
> Bei einer schließenden Klammer wird das oberste Symbol vom Keller entfernt und in die Ausgabe geschrieben.

Beispielsweise wird der Ausdruck

```
(5*(((9+8)*(4-6))+7))
```

durch diesen Algorithmus in die Postfix-Darstellung

```
5 9 8 + 4 6 - * 7 + *
```

überführt.

Schreiben Sie ein Java-Programm, das einen Ausdruck in Infix-Notation zunächst von der Kommandozeile einliest, den Ausdruck dann mithilfe des obigen Algorithmus in Postfix-Notation überführt und schließlich den so gewonnenen Ausdruck mit einem Keller, auf dem Zahlen vom Typ `Integer` gespeichert werden, auswertet. Sie können davon ausgehen, dass nur ganzzahlige positive Werte und die zweistelligen Operatoren `+`, `-`, `*`, `/` und `%` auf der Kommandozeile eingegeben werden und dass der Ausdruck in syntaktisch korrekter Form vorliegt. Gehen Sie folgendermaßen vor:

a) Verwenden Sie die Schnittstelle und die Implementierung aus der vorigen Aufgabe.

b) Schreiben Sie eine Java-Methode `infixToPostfix`, die einen arithmetischen Ausdruck, der in Infix-Notation vorliegt, in Postfix-Notation überführt.

c) Schreiben Sie eine Java-Methode `evaluatePostfix`, die einen Ausdruck in Postfix-Notation auswertet, indem sie die Zahlen zunächst auf einem Keller für Zahlen vom Typ `Integer` speichert und bei der Verarbeitung eines Operators zurückholt.

Nennen Sie Ihre Klasse, die die `main`-Methode enthält und die den Infix-Ausdruck einliest, `Evaluate`. Beispielsweise soll damit der Aufruf

```
java Evaluate (5*(((9+8)*(4-6))+7))
```

zur Ausgabe `-135` führen.

Aufgabe 61: In dieser Aufgabe betrachten wir das Problem, die Reihenfolge der Elemente eines Arrays eines generischen Typs umzukehren.

a) Entwickeln Sie eine rekursive Lösung für dieses Problem.

b) Geben Sie eine iterative Lösung mithilfe eines Kellers an. Verwenden Sie für den Keller eine geeignete Klasse aus dem Java Collections Framework.

c) Testen Sie Ihre Lösungen aus a) und b) an Kellern mehrerer Klassen.

Aufgabe 62: Der abstrakte Datentyp „Baum mit Elementen vom Typ T“ sei durch die generische Schnittstelle

```
interface Tree<T extends Comparable<T>> {
  boolean isEmpty();        // überprüft, ob der Baum leer ist
  boolean isInTree(T x);    // überprüft, ob x enthalten ist
  int size();               // liefert die Anzahl der Elemente
  int height();             // liefert die Höhe des Baums
  T minimum();              // liefert das kleinste Element
  T maximum();              // liefert das größte Element
  void insert(T x);         // fügt x in den Baum ein
  void delete(T x);         // löscht ein Vorkommen von x
  void deleteAll(T x);      // löscht alle Vorkommen von x
  String toString();        // liefert eine In-Order-Darstellung
}
```

definiert.

Implementieren Sie diesen abstrakten Datentyp durch binäre Suchbäume. Stellen Sie durch Verwendung einer Klasseninvarianten sicher, dass jeder Baum nach dem Einfügen oder dem Löschen eines Knotens die Bedingung eines binären Suchbaums erfüllt.

Testen Sie Ihr Programm, indem Sie ganze Zahlen, die in einer Datei gespeichert sind, nacheinander einlesen und in den Baum einfügen. Geben Sie anschließend den Baum mithilfe der Methode `toString()` aus. Welche Komplexität besitzt dieses Sortierverfahren im günstigsten bzw. ungünstigsten Fall?

Pflichtaufgabe 63: In der letzten Pflichtaufgabe haben Sie die Datenstruktur einer doppelt-verketteten, zirkulären Liste für Objekte von einem speziellen Typ erstellt. In dieser Aufgabe soll Ihre Liste nun für weitere Objekte durch Generizität angepasst werden. Hierfür wird Ihnen wieder eine JAR-Datei zur Verfügung gestellt, die eine generische sowie zwei explizite DataSources zur Erzeugung von Integer- und String-Objekten enthält. Des Weiteren ist ein Interface *NextFunction* gegeben, mit dem die generische DataSource Objekte von jedem gewünschten Typ erzeugen kann. Ebenfalls sind drei Datenstruktur-Interfaces für eine generische Liste, eine generische Queue und einen generischen Stack gegeben.

Sie dürfen für diese Aufgabe **keine** Klassen oder Methoden des JCF (Java Collections Framework) verwenden. Des weiteren dürfen Sie die vorgegebenen Dateien **nicht** verändern.

Datenquelle und Elemente: Die DataSource wurde im Vergleich zur letzten Aufgabe angepasst und kann über ihren Konstruktor ein Objekt vom Typ NextFunction entgegen nehmen, das aus einem dreistelligen Integer-Array neue Objekte erzeugt. Im Gegensatz zur letzten Aufgabe lassen sich die Elemente der DataSource nur noch indirekt über einen Iterator[1] aufzählen. Hierdurch ist es nun möglich, die DataSource mehrfach zu durchlaufen.

Erstellen Sie die Klasse EntryDataSoure, die die generische DataSource erweitert und Objekte vom Typ Ihrer Element-Objekte aus der letzten Aufgabe beim Aufruf von next()[2] an den Iterator zurück gibt.

Ihr Element-Typ soll dieses mal das generische Comparable<T>-Interface[3] implementieren. Hierbei dürfen Sie die Ordnungsrelation für das Vergleichen von Objekten Ihres Element-Typs selbst definieren. Des Weiteren müssen für Ihre Element-Klasse geeignete Implementierungen für die equals(o)[4]- und toString()-Methode[5] bereitgestellt werden.

MyList: Passen Sie Ihre Klasse MyList aus der letzten Aufgabe an das geänderte List-Interface an, sodass Ihre Klasse generisch ist. Bitte beachten Sie, dass das List-Interface keine Methode sortedInsert(e) mehr beinhaltet. Ihre MyList soll nun immer eine unsortierte Liste repräsentieren.

MySortedList: Erstellen Sie eine von MyList abgeleitete generische Klasse für sortierte Listen von Objekten, die das Comparable<T>-Interface implementieren. Sie können hierfür unnötige bzw. für die Struktur gefährliche Methoden überschreiben und eine UnsupportedOperationException[6] werfen.
Hinweis: Überlegen Sie sich hierbei, ob ein append() in einer sortierten Liste sinnvoll ist.

---

[1] https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html
[2] https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html#next--
[3] https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html
[4] https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#equals-java.lang.Object-
[5] https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#toString--
[6] https://docs.oracle.com/javase/8/docs/api/java/lang/UnsupportedOperationException.html

MyQueue: Ihnen ist ein Interface für eine Queue-Implementierung gegeben. Erstellen Sie eine Klasse, die Ihre MyList-Klasse verwendet und nur die folgenden Methoden implementiert:

`add(e)` Fügt das Element e ans Ende der Warteschlange ein.
`remove()` Löscht das vorderste Element und gibt dies zurück. Wenn die Warteschlange leer ist, wird null zurückgegeben.
`element()` Gibt das vorderste Element zurück. Wenn die Warteschlange leer ist, wird null zurückgegeben.

MyStack: Im Gegensatz zu einer Queue handelt es sich bei einem Stack um eine Datenstruktur, die nach dem *last in first out* Prinzip arbeitet. Erstellen Sie einen Stack, der Ihre MyList-Klasse verwendet und nur die folgenden Methoden implementiert:

`push(e)` Legt das Element e auf den Kopf des Stacks.
`pop()` Löscht das Element vom Kopf und gibt es zurück. Wenn der Stack leer ist, wird null zurückgegeben.
`peek()` Gibt das Element auf dem Kopf zurück. Wenn der Stack leer ist, wird null zurückgegeben.

Testen: Schreiben Sie für *jede* der *vier* Datenstrukturen einen einfachen Test, der jeder der *drei* DataSources im Modus *A* benutzt und mindestens das Einfügen und Löschen aller Elemente der DataSouce verwendet.