252-0027-00: Einführung in die Programmierung

Übungsblatt 8

Abgabe: 19. November 2024, 23:59

Die Bonusaufgabe für diese Übung wird erst am Dienstag Abend der Folgewoche (also am 19. 11.) um 17:00 Uhr publiziert und Sie haben dann 2 Stunden Zeit, diese Aufgabe zu lösen. Der Abgabetermin für die anderen Aufgaben ist wie gewohnt am Dienstag Abend um 23:59. Bitte planen Sie Ihre Zeit entsprechend.
Checken Sie mit Eclipse wie bisher die neue Übungs-Vorlage aus. Importieren Sie das Eclipse-Projekt wie bisher.

Aufgabe 1: Loop-Invariante

Gegeben ist eine Postcondition für das folgende Programm

```
public int compute(String s, char c) {
  // Precondition s != null
  int x;
  int n;

  x = 0;
  n = 0;

  // Loop-Invariante:
  while (x < s.length()) {
    if (s.charAt(x) == c) {
      n = n + 1;
    }
    x = x + 1;
  }

  // Postcondition: count(s, c) == n
  return n;
}
```

Die Methode `count(String s, char c)` gibt zurück, wie oft der Character `c` im String `s` vorkommt. Schreiben Sie die Loop-Invariante in die Datei "LoopInvariante.txt". **Tipp:** Benutzen Sie

die `substring`-Methode.

Aufgabe 2: Linked List

Bisher haben Sie Arrays verwendet, wenn Sie mit einer grösseren Anzahl von Werten gearbeitet haben. Ein Nachteil von Arrays ist, dass die Grösse beim Erstellen des Arrays festgelegt werden muss und danach nicht mehr verändert werden kann. In dieser Aufgabe implementieren Sie selbst eine Datenstruktur, bei welcher die Grösse im Vornherein nicht bestimmt ist und welche jederzeit wachsen und schrumpfen kann: eine *linked list* oder *verkettete Liste*.

Eine verkettete Liste besteht aus mehreren Objekten, welche Referenzen zueinander haben. Für diese Aufgabe besteht jede Liste aus einem "Listen-Objekt" der Klasse `LinkedIntList`, welches die gesamte Liste repräsentiert, und aus mehreren "Knoten-Objekten" der Klasse `IntNode`, eines für jeden Wert in der Liste. Die Liste heisst "verkettet", weil jedes Knoten-Objekt ein Feld mit einer Referenz zum nächsten Knoten in der Liste enthält. Das `LinkedIntList`-Objekt schliesslich enthält eine Referenz zum ersten und zum letzten Knoten und hat ausserdem ein Feld für die Länge der Liste.

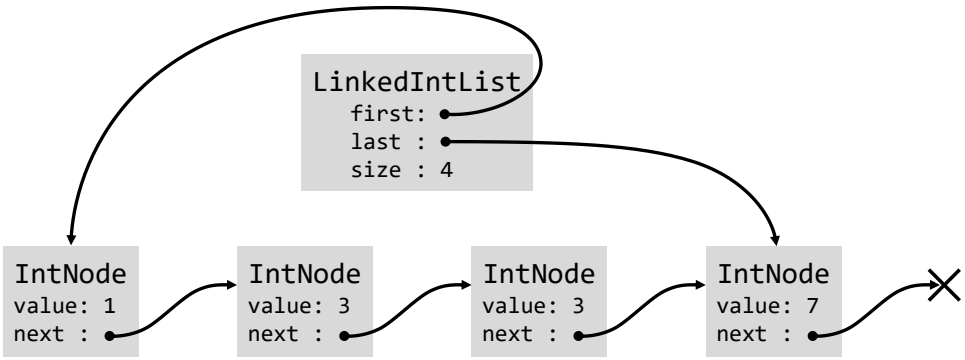


Abbildung 1: Verkettete Liste mit Werten 1, 3, 3, 7.

Abbildung 1 zeigt eine Liste, welche die Werte 1, 3, 3, 7 enthält. Beachten Sie, dass das `next`-Feld des letzten Knotens in der Liste auf kein Objekt zeigt, d.h. den Wert `null` enthält. Ausserdem wird eine leere Liste so repräsentiert, dass beide Felder `first` und `last` den Wert `null` enthalten (und `size` gleich `0` ist).

1. Implementieren Sie die Klasse `LinkedIntList` (in "LinkedIntList.java"), welche zusammen eine verkettete Liste von `int`s ergeben. Fügen Sie zu den jeweiligen Klassen die benötigten Felder hinzu und implementieren Sie danach folgende Methoden in `LinkedIntList`, um die Klasse benutzerfreundlicher zu machen:

Name	Parameter	Rückg.-Typ	Beschreibung
`addLast`	`int value`	`void`	fügt einen Wert am Ende der Liste ein
`addFirst`	`int value`	`void`	fügt einen Wert am Anfang der Liste ein
`removeFirst`		`int`	entfernt den ersten Wert und gibt ihn zurück
`removeLast`		`int`	entfernt den letzten Wert und gibt ihn zurück
`clear`		`void`	entfernt alle Wert in der Liste
`isEmpty`		`boolean`	gibt zurück, ob die Liste leer ist
`get`	`int index`	`int`	gibt den Wert an der Stelle `index` zurück
`set`	`int index, int value`	`void`	ersetzt den Wert an der Stelle `index` mit `value`
`getSize`		`int`	gibt zurück, wie viele Werte die Liste enthält

Einige dieser Methoden dürfen unter gewissen Bedingungen nicht aufgerufen werden. Zum Beispiel darf `removeFirst()` nicht aufgerufen werden, wenn die Liste leer ist, oder `get()` darf nicht aufgerufen werden, wenn der gegebene Index grösser oder gleich der aktuellen Länge der Liste ist. In solchen Situationen soll sich Ihr Programm mit einer Fehlermeldung beenden. Verwenden Sie folgendes Code-Stück dafür:

```
if(condition) {
    Errors.error(message);
}
```

Ersetzen Sie *`condition`* mit der Bedingung, unter welcher das Programm beendet werden soll, und *`message`* mit einer hilfreichen Fehlermeldung. Die `Errors`-Klasse befindet sich bereits in Ihrem Projekt, aber Sie brauchen sie im Moment nicht zu verstehen.

Sie finden einige Tests für die verkettete Liste in "LinkedIntListTest.java".

2. Erstellen Sie ein Programm `Echo.java`, welches vom Benutzer `int`-Werte entgegen nimmt, diese in einer `LinkedIntList` speichert und zum Schluss alle Werte in der Liste wieder ausgibt. Das Programm soll so lange Werte einlesen, bis der Benutzer eine ungültige Eingabe macht. Verwenden Sie dazu `Scanner.hasNextInt()`.

Aufgabe 3: Executable Graph

In dieser Aufgabe verwenden wir gerichtete azyklische Graphen, um Programme zu repräsentieren. Der Programmzustand ist dabei immer durch ein Tupel $(sum, counter)$ gegeben, wobei *sum* und *counter* ganze Zahlen sind. Programmzustände werden durch `ProgramState`-Objekte modelliert, wobei `ProgramState.getSum()` (bzw. `ProgramState.getCounter()`) dem ersten Element (bzw. dem zweiten Element) des Tupels entspricht.

Eine Ausführung des Programms manipuliert den Programmzustand. Das Resultat eines Programms ist gegeben durch den erreichten Programmzustand, nachdem alle Operationen im Programm ausgeführt wurden. Programme können nichtdeterministisch sein: Das heisst, für ein einzelnes Programm kann es für den gleichen Startzustand mehrere Programmausführungen geben, welche zu unterschiedlichen Resultaten führen.

Knoten in Graphen werden durch `Node`-Objekte modelliert. `Node.getSubnodes()` gibt die Kinderknoten als ein Array zurück (m ist genau dann ein Kinderknoten von n, wenn es eine ausgehende gerichtete Kante von n zu m gibt). Wir unterscheiden drei Arten von Knoten, wobei

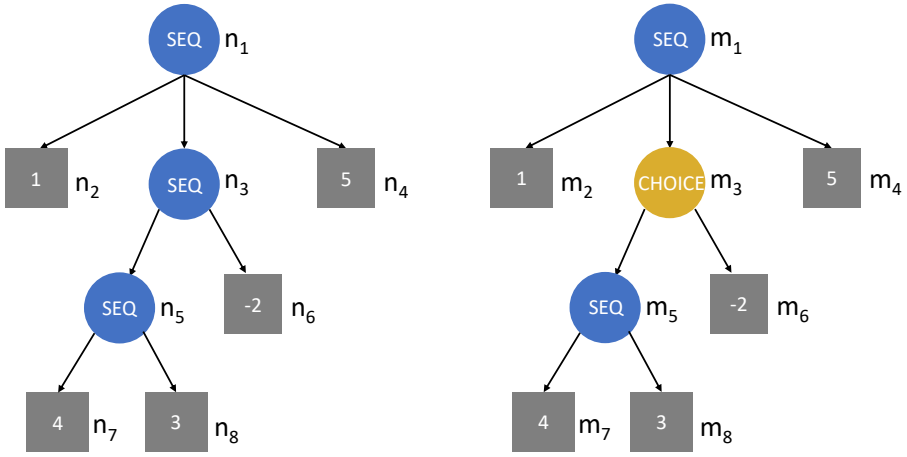
die Methode `Node.getType()` die Knotenart als `String` zurückgibt. Um ein Programm, welches durch den Knoten `n` repräsentiert wird, auszuführen, muss man den "Knoten `n` ausführen". Wir beschreiben nun die drei Knotenarten und jeweils die Ausführung der Knoten:

1. Additionsknoten (`Node.getType()` ist "ADD"): Solche Knoten besitzen einen Additionswert a gegeben durch `Node.getValue()` (eine ganze Zahl) und bei der Ausführung dieses Knotens wird der Programmzustand von $(sum, counter)$ zu $(sum + a, counter + 1)$ aktualisiert. Die Kinderknoten von solchen Knoten werden bei der Ausführung ignoriert.

2. Sequenzknoten (`Node.getType()` ist "SEQ"): Bei der Ausführung eines Sequenzknoten `n` werden die Kinderknoten von `n` nacheinander ausgeführt. Die Reihenfolge in welcher die Kinderknoten ausgeführt werden spielt keine Rolle, da der erreichte Programmzustand für jede Reihenfolge gleich ist. `Node.getValue()` ist irrelevant.

3. Auswahlknoten (`Node.getType()` ist "CHOICE"): Bei der Ausführung eines Auswahlknoten `n` wird ein beliebiger Kinderknoten von `n` ausgewählt und ausgeführt. `Node.getValue()` ist irrelevant. Diese Knoten führen zu Nichtdeterminismus.

Sie dürfen davon ausgehen, dass Sequenz- und Auswahlknoten immer mindestens einen Kinderknoten haben, und dass es zwischen zwei Knoten immer höchstens einen Pfad gibt. Die folgende Abbildung zeigt zwei Beispielgraphen, wobei Knoten mit der Beschriftung "SEQ" (bzw. "CHOICE") Sequenzknoten (bzw. Auswahlknoten) entsprechen und die Zahlen in Additionsknoten den Additionswerten entsprechen.



Beim linken Graphen in der Abbildung gibt es immer nur eine mögliche Ausführung von n_1 pro Startzustand. Für den Startzustand $(1, 2)$ ist das Resultat gegeben durch $(1 + 1 + 4 + 3 - 2 + 5, 2 + 5) = (12, 7)$. Beim rechten Graphen gibt es zwei mögliche Ausführungen von m_1. Beim Auswahlknoten m_3 wird entweder m_5 oder m_6 ausgeführt (da m_5 und m_6 die Kinderknoten von m_3 sind). Die beiden möglichen Resultate für den Startzustand $(0, 0)$ sind $(0 + 1 + 4 + 3 + 5, 0 + 4) = (13, 4)$ (wenn m_5 gewählt wird) und $(0 + 1 - 2 + 5, 0 + 3) = (4, 3)$ (wenn m_6 gewählt) wird.

Implementieren Sie `GraphExecution.allResults(Node n, ProgramState initState)`, welche für den Startzustand `initState` alle möglichen Resultate für das Programm repräsentiert durch `n` zurückgibt. Die Resultate sollten als eine Liste von `ProgramState`-Objekten zurückgegeben werden (repräsentiert durch die Klasse `LinkedProgramStateList`). Die Reihenfolge der

zurückgegeben Liste spielt keine Rolle. Wenn das gleiche Resultat durch genau k verschiedene Ausführungen generiert werden kann, dann muss das Resultat k Mal in der zurückgegeben Liste vorkommen. Zwei Ausführungen sind unterschiedlich, wenn es mindestens einen Knoten gibt, der in einer aber nicht in der anderen Ausführung ausgeführt wird.

Wir stellen zwei Testdateien zur Verfügung. "GraphExecutionTest.java" enthält Tests, welche wir an einer Prüfung geben würden. "GradingGraphExecutionTest.java" enthält Tests, welche wir zum Korrigieren einer Prüfung verwenden würden. Testen Sie ihre Lösung zuerst ausgiebig mit "GraphExecutionTest.java" (am besten fügen Sie selber neue Tests hinzu) und dann können Sie "GradingGraphExecutionTest.java" verwenden, um zu sehen, wie ihre Lösung an einer Prüfung abgeschnitten hätte.

Aufgabe 4: Energiespiel

In dieser Aufgabe üben Sie den Umgang mit Enums. Dafür haben Sie einen Ordner `EnergieSpiel` mit drei Klassen `GameApp`, `Game` und `Player`, sowie ein Enum `Role`. Diese sind bereits so implementiert, dass alles funktioniert. Die Klasse `Player` hat jedoch ein Feld `role` vom Typ `String`. Java lässt also zu, dass in diesem Feld ein beliebiger String abgespeichert werden kann. Das Spiel hat aber eigentlich nur genau drei Möglichkeiten: `HONEST`, `TRICKSTER` oder `SORCEROR`. Das Enum `Role` mit diesen drei Optionen existiert bereits. Ändern Sie den Typ des Feldes zu `Role` und passen Sie den Code in allen drei Klassen so an, dass die Rollen-Logik überall den Typ `Role` statt `String` verwendet.