Discord: timostucki

# Meine Aufgaben Ratings:

- Best Case: Ihr macht alle Aufgaben (, wenn die Zeit reicht)
- Falls nicht: Ich mache Ratings zur Wichtigkeit der einzelnen Aufgaben

- (Keine offizielle Empfehlung, meine subjektive Meinung auf Basis meiner eigenen Erfahrung als Student in diesem Kurs)
- ! Sagen nichts über die Schwierigkeit der Aufgaben aus !

Wichtig für die Prüfung:
(Prüfungs- oder
prüfungsähnliche Aufgaben)

Wichtig für euer Verständnis:
(Aber nicht in Prüfungsform)

Wichtig für tiefes Verständnis/
Zusatzaufgaben:
(Bspw. viele vom gleichen Typ)

# 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.

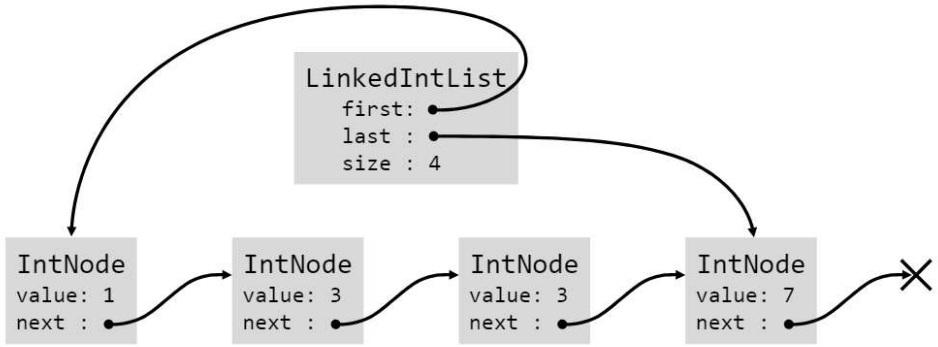



Abbildung 1: Verkettete Liste mit Werten 1, 3, 3, 7.

# Aufgabe 2: Linked List

| 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.

# Aufgabe 3: Executable Graph

Alte Prüfungsaufgabe aus FS22:
https://exams.vis.ethz.ch/exams/m36aa1mk.pdf

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 und 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:

# Aufgabe 3: Executable Graph

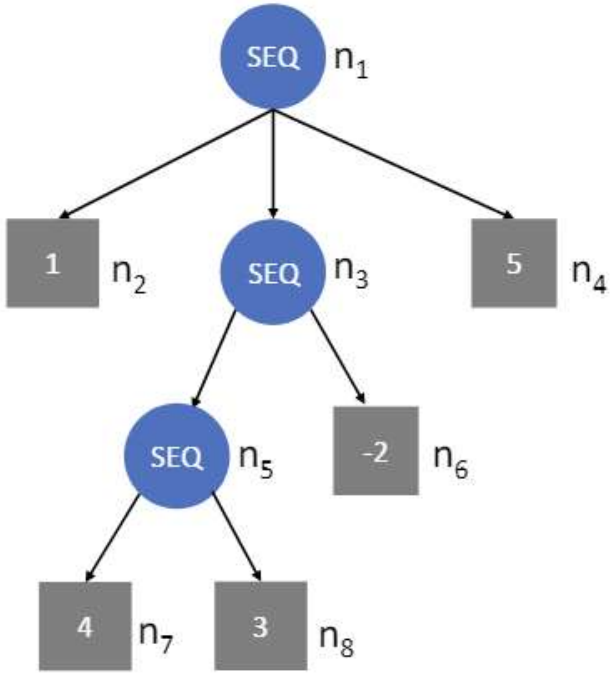
1. Additionsknoten (`Node.getType()` ist "ADD"): Solche Knoten besitzen einen Additionswert $a$ gegeben durch `Node.getValue()` (eine ganze Zahl) und bei der Ausführung dieses Knotens wird der Programmzustand von $(sum, counter)$ zu $(sum + a, counter + 1)$ aktualisiert. Die Kinderknoten von solchen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

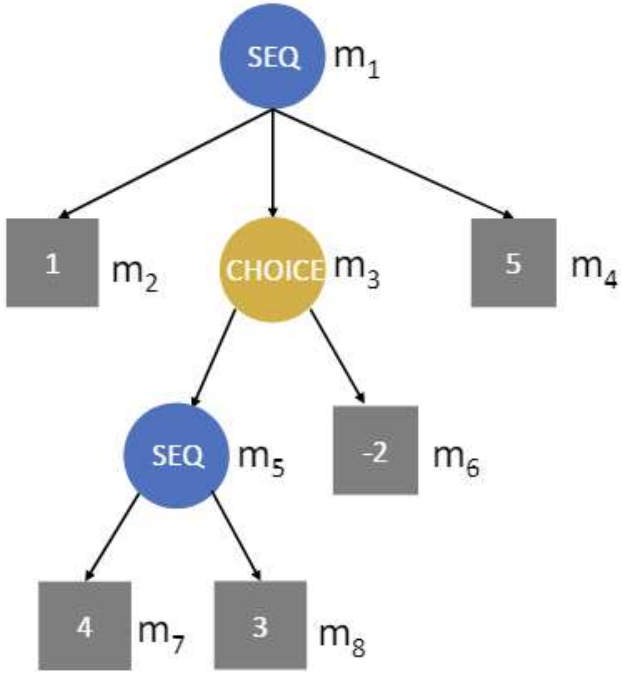
2. Sequenzknoten (`Node.getType()` ist "SEQ"): Bei der Ausführung eines Sequenzknoten `n` werden die Kinderknoten von `n` nacheinander ausgeführt. Die Reihenfolge in welcher die Kinderknoten ausgeführt werden spielt keine Rolle, da der erreichte Programmzustand für jede Reihenfolge gleich ist. `Node.getValue()`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

**Aufgabe 3: Executable Graph**



# 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 `Character`. Diese sind bereits so implementiert, dass alles funktioniert. Die Klasse `Player` hat jedoch ein Feld `character` von 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 `Character` mit diesen drei Optionen existiert bereits. Ändern Sie den Typ des Feldes zu `Character` und passen Sie den Code in allen drei Klassen so an, dass die Charakter-Logik überall den Typ `Character` statt `String` verwendet.

# NICHT VERGESSEN

## Aufgabe 5: Timed Bonus

Die Bonusaufgabe für diese Übung wird erst am Dienstag Abend der Folgewoche (also am 18. 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 IntelliJ, wie bisher, die neue Übungs-Vorlage aus. Importieren Sie das IntelliJ-Projekt wie bisher.