

Übungsstunde 8

Einführung in die Programmierung

Plan für heute

Nachbesprechung Übung 7

Eigenständiges Lösen einer Aufgabe

Vorbesprechung Übung 8

Kahoot

Probleme bei Übung 6

Keine :)

Sehr gut gelöst

War aber auch nicht die längste/schwierigste Serie

Umfragewerte

Lösen der Bonusaufgabe: ~ 1h bis 4h

Besprechung:

- EBNF (Theorie)
- Linked-List

static mit Bedacht nutzen, hier falsch:

 **LinkedList.java**  2.54 KiB

```
1  import java.util.List;
2
3  public class LinkedList {
4      IntNode first;
5      IntNode last;
6      static int size=0;
7
8
9      public void addLast(int data) {
```

Nachbesprechung Übung 7

Aufgabe 1: EBNF Wiederholung

1. Erstellen Sie eine EBNF Beschreibung $\langle xyz \rangle$, die als legale Symbole genau jene Wörter zulässt, in denen für jedes X entweder ein Y oder drei Z als Gruppe auftreten.

Beispiele legaler Symbole: "XY", "XXYZZZ", "", "XYYXZZZX"

$$\langle yz \rangle \leq Y \mid ZZZ$$
$$\langle xyz \rangle \leq \{ X \langle xyz \rangle \langle yz \rangle \mid \langle yz \rangle \langle xyz \rangle X \}$$

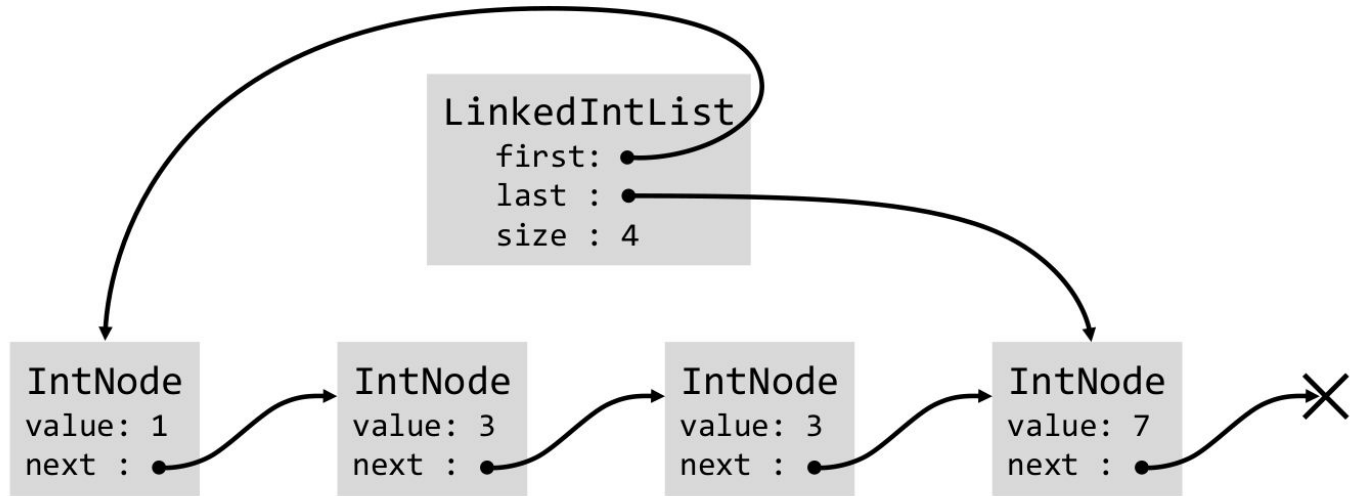
2. Erstellen Sie eine EBNF Beschreibung $\langle abc \rangle$, die als legale Symbole genau jene Wörter zulässt, die folgende Bedingungen erfüllen:

- Ein Wort besteht ausschliesslich aus den Symbolen A, B, C, X, Y, Z.
- Ein Wort beginnt mit A und endet mit B.
- Nach jedem X kommt entweder ein Y oder ein Z. Y und Z können nur direkt nach einem X auftreten.
- C kann nur in Paaren auftreten (CC) und kann nicht direkt nach einem Z folgen.

Beispiele legaler Symbole: "AB", "ACCB", "AXZB", "ABAB", "AXYXZXZXYCCB"

$$\langle abc \rangle \leq A \{CC\} \{ XY \{CC\} \mid XZ \mid A \{CC\} \mid B \{CC\} \} B$$

Aufgabe 2: Linked List



Aufgabe 2: Linked List

Immer folgende Cases beachten:

- Leere Liste
- Liste enthält genau ein Element
- Sonst

Immer alle Variablen updaten (falls nötig): size, first, (last)

Nutzt **last**, wenn möglich und nicht extra Loop dafür machen

Aufgabe 2: Linked List

Java besitzt den Garbage Collector

=> muss nicht alles dereferenzieren, **first = last = null** reicht

Name	Parameter	Rückg.-Typ	Beschreibung
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

addFirst(int value)

```
void addFirst(int value) {  
    IntNode newNode = new IntNode(value);  
    newNode.next = first;  
    first = newNode;  
    if(last == null)  
        last = newNode;  
    size++;  
}
```

addLast(int value)

```
void addLast(int value) {  
    IntNode newNode = new IntNode(value);  
    if(isEmpty())  
        first = newNode;  
    else  
        last.next = newNode;  
  
    last = newNode;  
    size++;  
}
```

int removeFirst()

```
int removeFirst() {  
    if(isEmpty()) {  
        Errors.error("removeFirst() on empty list!");  
    }  
  
    int value = first.value;  
    if(first == last) {  
        // List has only one element, so just clear it  
        clear();  
    }  
    else {  
        first = first.next;  
        size--;  
    }  
  
    return value;  
}
```

```
int removeLast() {  
    if(isEmpty()) {  
        Errors.error("removeLast() on empty list!");  
    }  
  
    int value = last.value;  
    if(first == last) {  
        // List has only one element, so just clear it  
        clear();  
    }  
    else {  
        // List has more than one element  
        IntNode currentNode = first;  
        while(currentNode.next != last)  
            currentNode = currentNode.next;  
  
        currentNode.next = null;  
        last = currentNode;  
        size--;  
    }  
    return value;  
}
```



```
void clear() {  
    first = last = null;  
    size = 0;  
}
```

```
boolean isEmpty() {  
    return size == 0;  
}
```

Für get(...) und set(...)

```
/**
 * For internal use only.
 */

IntNode getNode(int index) {
    if(index < 0 || index >= size) {
        Errors.error("getNode() with invalid index: " + index);
    }

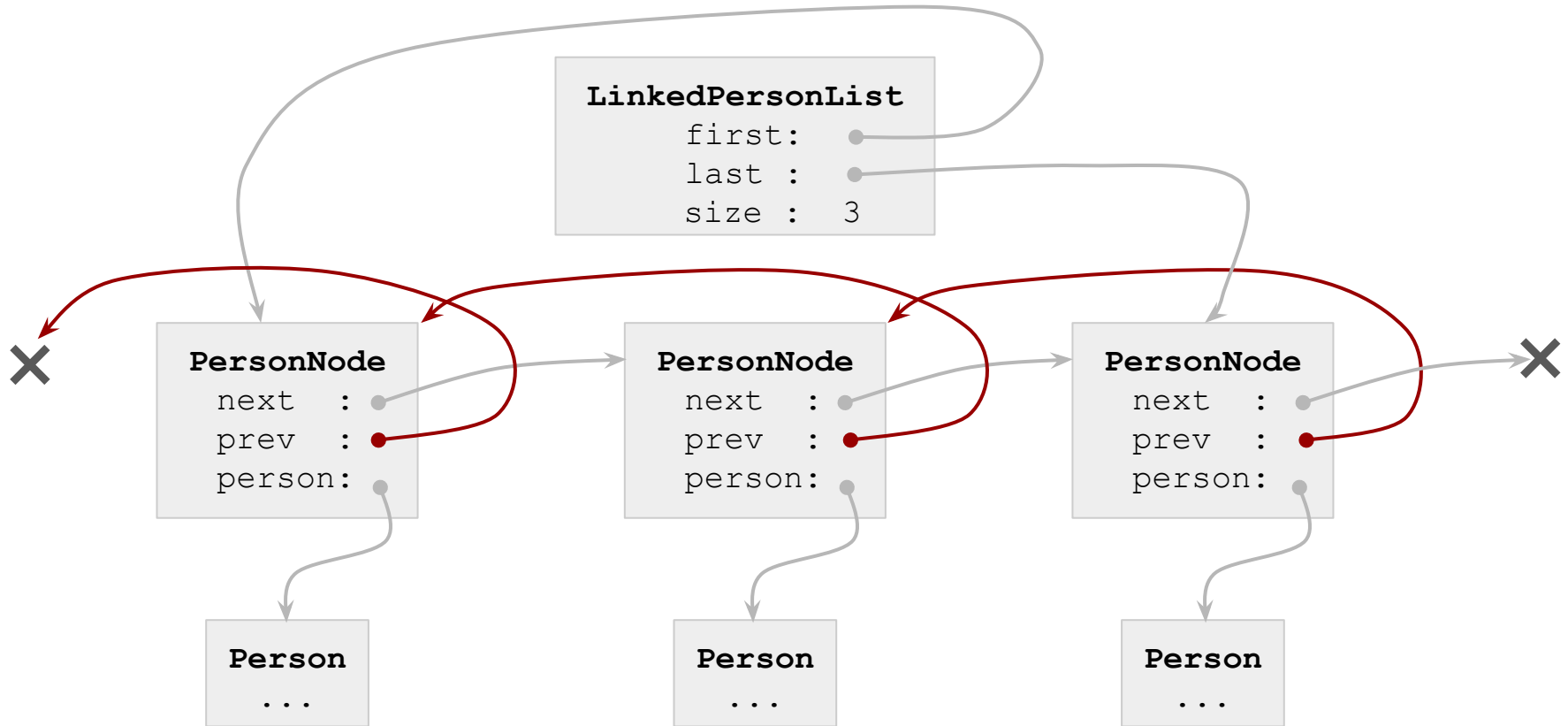
    IntNode current = first;
    for(int i = 0; i < index; i++)
        current = current.next;
    return current;
}
```

get(int index) & set(int index, int value)

```
int get(int index) {  
    return getNode(index).value;  
}
```

```
void set(int index, int value) {  
    getNode(index).value = value;  
}
```

Aufgabe 3: Doubly-linked List



prev

```
void addFirst(Person value) {  
    PersonNode newNode = new PersonNode(value);  
    if(isEmpty())  
        last = newNode;  
    else {  
        first.prev = newNode;  
        newNode.next = first;  
    }  
  
    first = newNode;  
    size++;  
}
```

removeNode

```
void removeNode(PersonNode node) {  
    if(node == first)  
        removeFirst();  
    else if(node == last)  
        removeLast();  
    else {  
        node.prev.next = node.next;  
        node.next.prev = node.prev;  
        size--;  
    }  
}
```

Aufgabe 3: Doubly-linked List

Auch hier immer die 3 Cases prüfen: leer, 1 Element, sonst

Verwendet eure Methoden wieder

Nutzt **prev**, wenn möglich und nicht extra Loop dafür machen

Immer an alle Attribute der veränderten Nodes denken

Aufgabe 4: Bonusaufgabe (Bonus!)

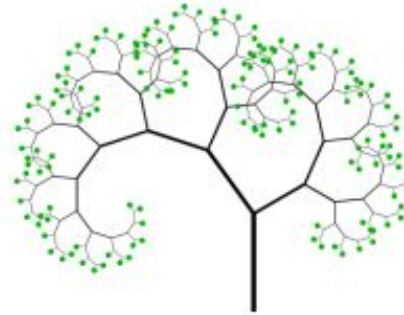
Feedback nach der Korrektur direkt per Git

Aufgabe 5: Rekursion

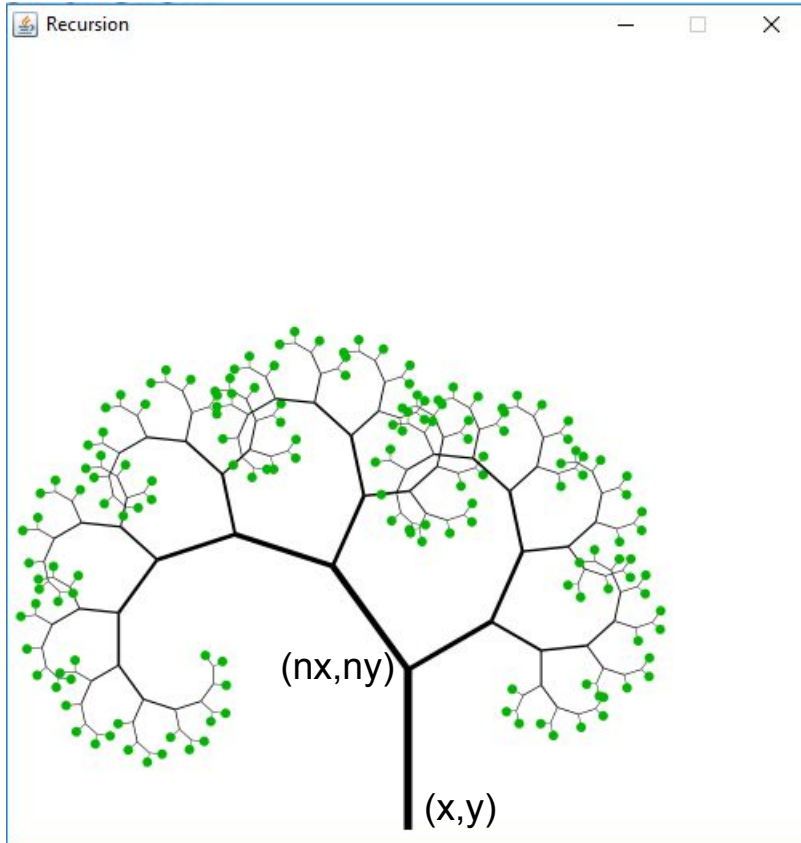
In dieser Aufgabe zeichnen Sie einen Baum mittels Rekursion. Wie auf dem Bild erkennbar, besteht der Baum aus mehreren zusammenhängenden Segmenten. Für jedes Segment gibt es zwei "Sub-Bäume", einen gedreht im Uhrzeigersinn und einen gedreht im Gegenuhrzeigersinn.

Der Baum wird in mehreren Schritten gezeichnet. Jeder Schritt ist durch vier Parameter (x, y, α, l) bestimmt:

- Startpunkt (x, y) des aktuellen Segments
- Richtung α des Segments
- Länge l des Segments



Aufgabe 5: Rekursion



```
// 1. Zeichnen des aktuellen Segments
```

```
double nx = x + Math.cos(angle) * length;
```

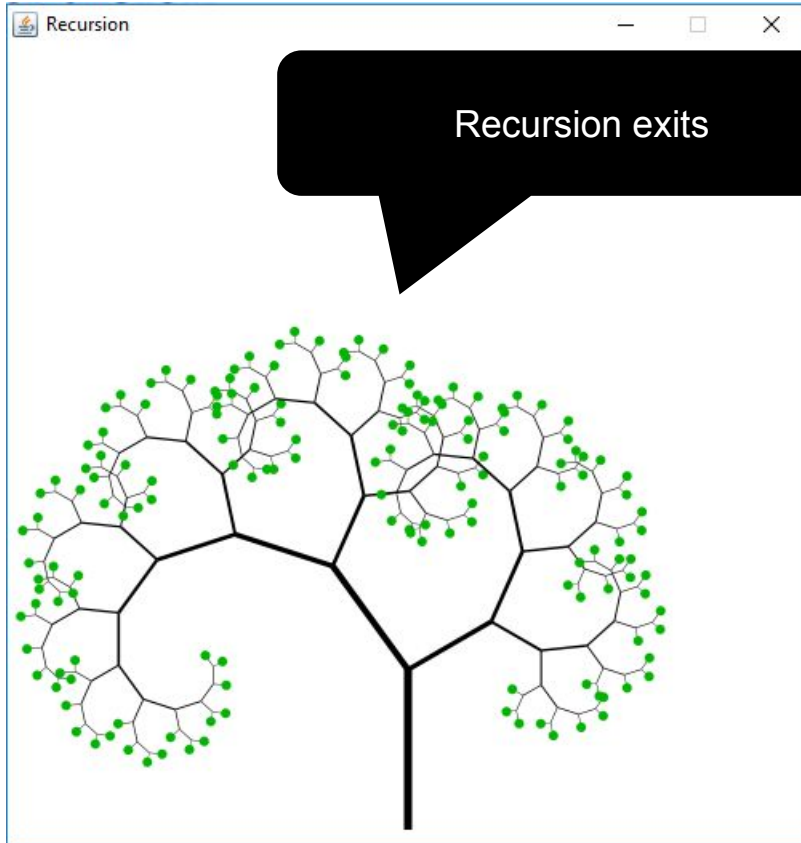
```
double ny = y - Math.sin(angle) * length;
```

```
win.setStrokeWidth(length / 20);
```

```
win.setColor(0, 0, 0);
```

```
win.drawLine(x, y, nx, ny);
```

Aufgabe 5: Rekursion



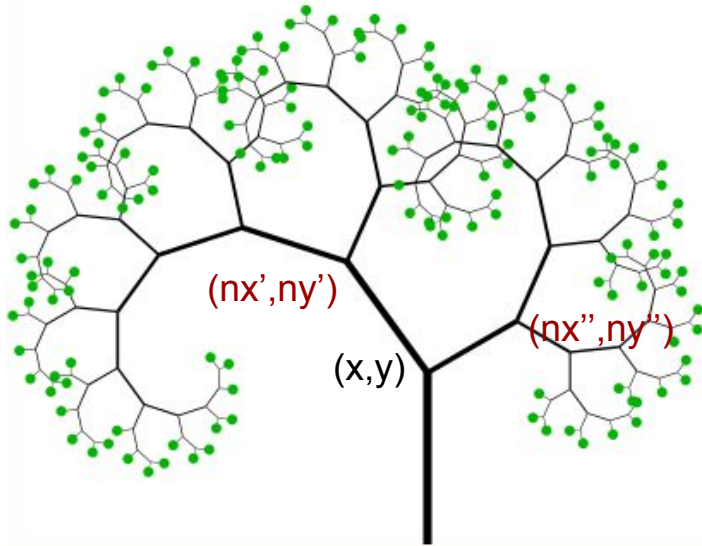
```
// Falls l < 10 endet die Rekursion und  
// am Ende des Segments wird ein Blatt  
// gemalt.
```

```
if (length < 10.0) {  
    win.setColor(0, 180, 0);  
    win.fillCircle(nx, ny, 3);  
}
```

```
// Jede Rekursion muss einen Ausgang  
// haben!
```

Aufgabe 5: Rekursion

Anderenfalls werden rekursiv zwei weitere Zeichenschritte aufgerufen: Für den ersten Zeichenschritt werden die Argumente $l' = 0.8 \cdot l$ und $\alpha' = \alpha + \frac{\pi}{5}$, und für den zweiten die Argumente $l'' = 0.6 \cdot l$ und $\alpha'' = \alpha - \frac{\pi}{3}$ verwendet. Der Startpunkt für beide Zeichenschritte entspricht dem Endpunkt des aktuellen Segments.



```
drawTree(win, nx, ny,  
         length * 0.8,  
         angle + Math.PI / 5);
```

```
drawTree(win, nx, ny,  
         length * 0.6,  
         angle - Math.PI / 3);
```

Aufgabe 5: Rekursion





```
//recursive method
```

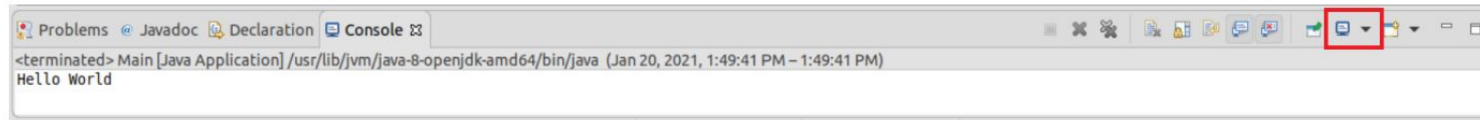
```
public static void drawTree(Window win, double x, double y, double length, double angle) {  
    double nx = x + Math.cos(angle) * length;  
    double ny = y - Math.sin(angle) * length;  
  
    win.setStrokeWidth(length / 20);  
    win.setColor(0, 0, 0);  
    win.drawLine(x, y, nx, ny);  
  
    if (length < 10.0) {  
        win.setColor(0, 180, 0);  
        win.fillCircle(nx, ny, 3);  
    } else {  
        drawTree(win, nx, ny, length * 0.8, angle + Math.PI / 5);  
        drawTree(win, nx, ny, length * 0.6, angle - Math.PI / 3);  
    }  
}
```


Alte Bonusaufgabe

Verhindern von Abstürzen






Bevor Sie ein Programm oder einen Test ausführen, achten Sie darauf, dass alle anderen Programme und Tests korrekt terminiert wurden. Wenn zu viele Programme gleichzeitig laufen, dann wird Ihr Computer langsamer, manchmal einfrieren, und im schlimmsten Fall abstürzen. Auch verhält sich dann manchmal Eclipse oder der Debugger unnatürlich.

Das geht von Ihrer Zeit ab. Ein Problem ist, dass, auch wenn die aktive Konsole mit  terminiert wurde und somit ausgegraut ist () , es noch weitere Konsolen geben kann, welche immer noch laufen. Wenn das Icon  vorhanden und nicht ausgegraut ist () , dann gibt es mehr als eine Konsole, was auch heisst, dass mehrere Programme oder Tests noch nicht terminiert sein können:



Durch einen Klick auf den Pfeil von , wird eine Liste aller vorhandenen Konsolen angezeigt:



Durch klicken von  werden alle terminierte Konsolen geschlossen. Klicken Sie wiederholt  und  (oder ) bis  ausgegraut oder verschwunden ist, um alle Programme und Tests zu terminieren und alle Konsolen zu schliessen.

Vorbesprechung Übung 8

Aufgabe 1: Loop Invariante

Gegeben ist eine Postcondition für das folgende Programm

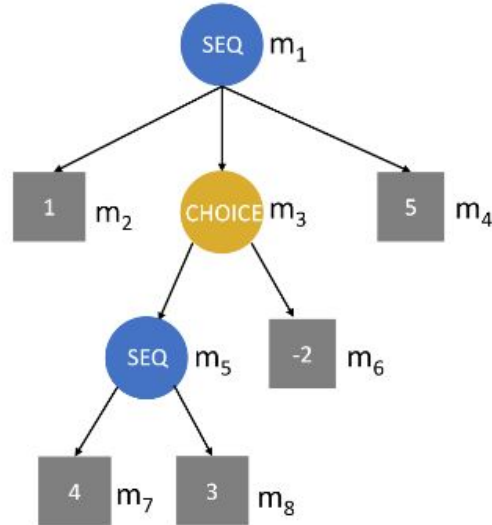
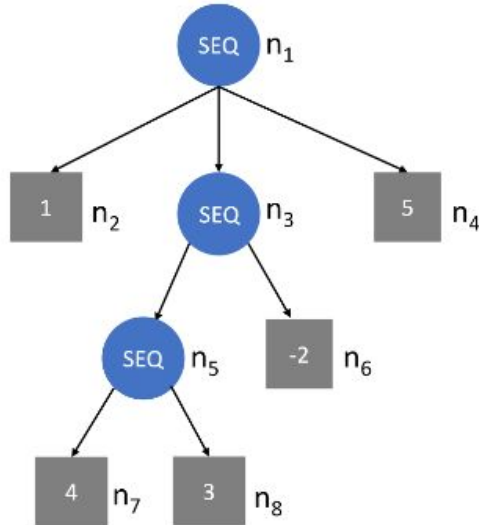
```
public int compute(String s, char c) {  
    int x;  
    int n;  
  
    x = 0;  
    n = 0;  
  
    // Loop Invariante:  
    while (x < s.length()) {  
        if (s.charAt(x) == c) {  
            n = n + 1;  
        }  
        x = x + 1;  
    }  
  
    // Postcondition: count(s, c) == n  
    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: Executable Graph (Bonus!)

Siehe Aufgabenblatt

Aufgabe 2: Executable Graph (Bonus!)



Aufgabe 3: Warteschlangen-Simulation

Doppelt verkettete Listen eignen sich gut als Warteschlangen (engl. "Queue"). "Software-Warteschlangen" werden in vielen Anwendungen verwendet, um Objekte oder Daten zwischenspeichern, bevor sie verarbeitet werden. In dieser Aufgabe sollen Sie sie hingegen verwenden, um die realen Warteschlangen vor den Kassen in Ihrer nächsten Migros-Filiale zu simulieren.

Die Simulation läuft in diskreten Schritten ab. Das heisst, die Zeit wird in gleichgross Intervalle unterteilt und in jedem Intervall passieren gewisse Dinge. Zum Beispiel kann in einem Intervall ein Kassierer einen Gegenstand scannen oder eine Person die Warteschlange wechseln. Die Simulation hat folgende Parameter:

Anzahl Kassen und Kassier-Effizienz: Eine Liste $[p_0, p_1, \dots, p_n]$ der Länge n mit $0 \leq p_i < 1$. Es gibt n (besetzte) Kassen und der i -te Kassierer scannt in jedem Zeitschritt mit Wahrscheinlichkeit p_i einen Gegenstand aus dem Warenkorb der vordersten Person in seiner Warteschlange. Wenn er den letzten Gegenstand einer Person gescannt hat, verlässt diese Person die Kasse.

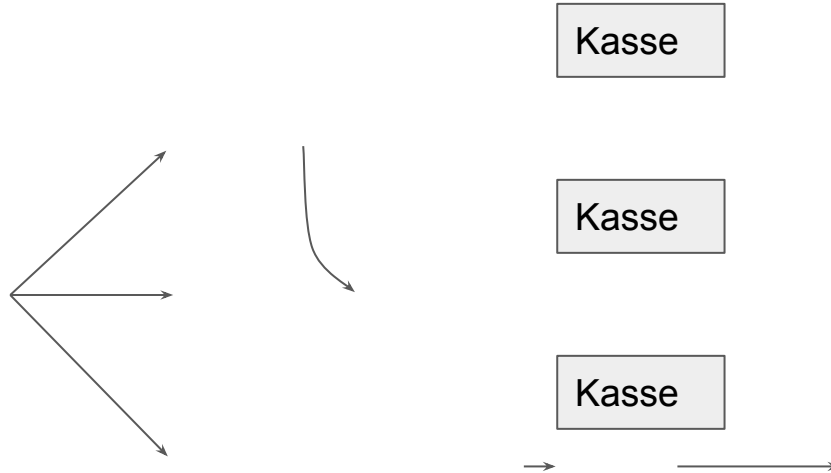
Erschein-Wahrscheinlichkeit: Eine Zahl e , wobei $0 \leq e < 1$. In jedem Zeitschritt erscheint mit Wahrscheinlichkeit e eine neue Person und steht an einer zufälligen Schlange an.

Warenkorb-Grösse: Eine natürliche Zahl w . Jede Person hat zu Beginn zwischen 0 und w Gegenstände in seinem Warenkorb. Die genaue Anzahl jeder Person ist zufällig.

Faulheit der Leute: Eine natürliche Zahl f . Da gewisse Schlangen länger werden können als andere, möchten die Leute vielleicht mal wechseln, abhängig von ihrer Faulheit. Eine Person wechselt (in jedem Zeitschritt) zur momentan kürzesten Schlange, falls diese um mindestens f kürzer ist als die aktuelle Position¹ dieser Person in seiner Schlange.

Aufgabe 3: Warteschlangen-Simulation

Doppelt verkettete Listen eignen sich gut als Warteschlangen (engl. "Queue"). "Software-Warteschlangen" werden in vielen Anwendungen verwendet, um Objekte oder Daten zwischenspeichern, bevor sie verarbeitet werden. In dieser Aufgabe sollen Sie sie hingegen verwenden, um die realen Warteschlangen vor den Kassen in Ihrer nächsten Migros-Filiale zu simulieren.



Aufgabe 4: Dominator

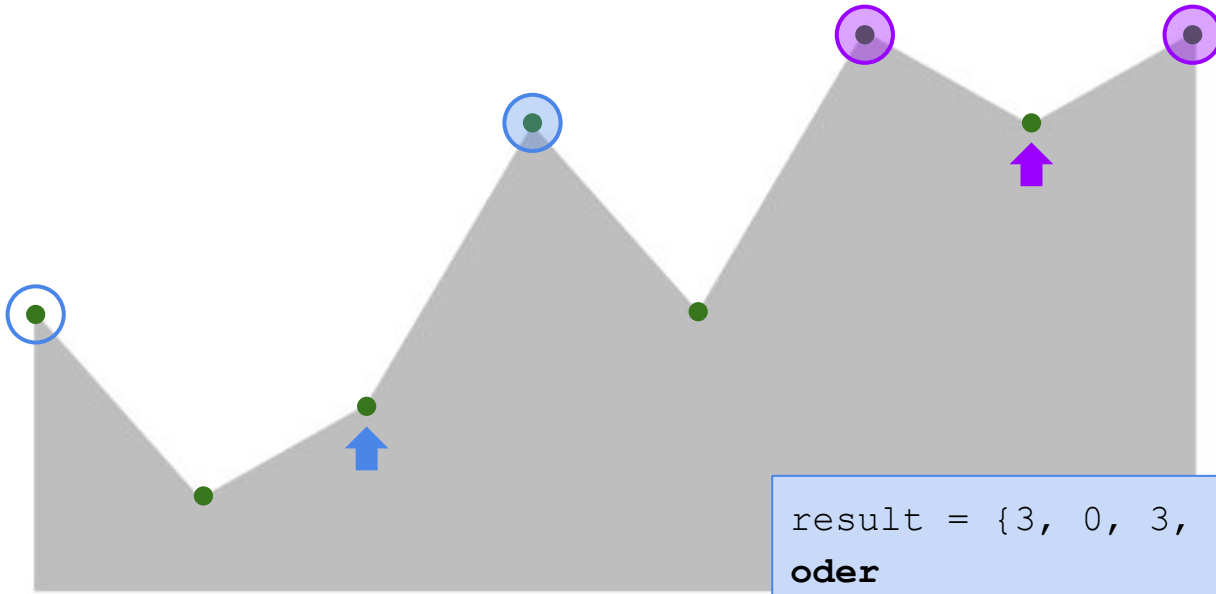
Schreiben Sie Junit-Tests für die Methode `int[] dominators(int[] elevations)`

Die Methode bekommt ein Array, mit Höhenangaben für jeden Punkt P, als Input und gibt ein Array mit dem Index des jeweiligen Dominators zurück.

Ein Dominator D eines Punktes P hat von allen Punkten, welche höher als P liegen, die kleinste (horizontale) Distanz zu P. Falls zwei solche Punkte existieren, ist der höhere der beiden der einzige Dominator. Falls beide dieser Punkte gleich hoch sind, gibt es zwei mögliche Dominatoren. Falls kein Dominator für P existiert, enthält `result[P]` die Zahl -1.

Aufgabe 4: Dominator

Höhenserie {3, 1, 2, 5, 3, 6, 5, 6}



```
result = {3, 0, 3, 5, 5, -1, 5, -1}
```

oder

```
result = {3, 0, 3, 5, 5, -1, 7, -1}
```

Aufgabe 4: Dominator

Weiteres Beispiel:

elevations = {5, 1, 5, 6}

```
result = {3, 0, 3, -1}
```

```
result = {3, 2, 3, -1}
```


Aufgabe 4: Dominator

Um die Stärke Ihrer Tests zu beurteilen, werden verschiedene, teilweise fehlerhafte Implementierungen mithilfe Ihrer Tests überprüft. Je mehr Fehler Ihre Tests aufdecken, desto besser. Tests sollten fehlschlagen, falls die Implementierung fehlerhaft ist, und erfolgreich durchlaufen, falls keine Fehler vorhanden sind.

Als Hilfe können Sie sich die Lösung der Aufgabe "Black-Box Testing" aus der letzten Serie anschauen.

Aufgabe 5: Self-avoiding Random Walks

Die Stadt besteht aus $2*N$ Strassen, die in einem regelmässigen Gitter angeordnet sind (N ist ungerade und > 1). Die Stadt kann also vollständig durch die $N*N$ Kreuzungen der Strassen beschrieben werden.

Eine einmal besuchte Kreuzung wird nicht nochmal besucht. Der Wolf wählt zufällig unter den Richtungen, die ihn zu einer neuen, noch nicht besuchten Kreuzung führen. Wenn der Wolf den Stadtrand erreicht hat, ist die Flucht erfolgreich verlaufen. Wenn der Wolf an eine Kreuzung kommt, von der aus er keine unbesuchte Kreuzung erreichen kann, dann ist die Flucht fehlgeschlagen.

Aufgabe 5: Self-avoiding Random Walks

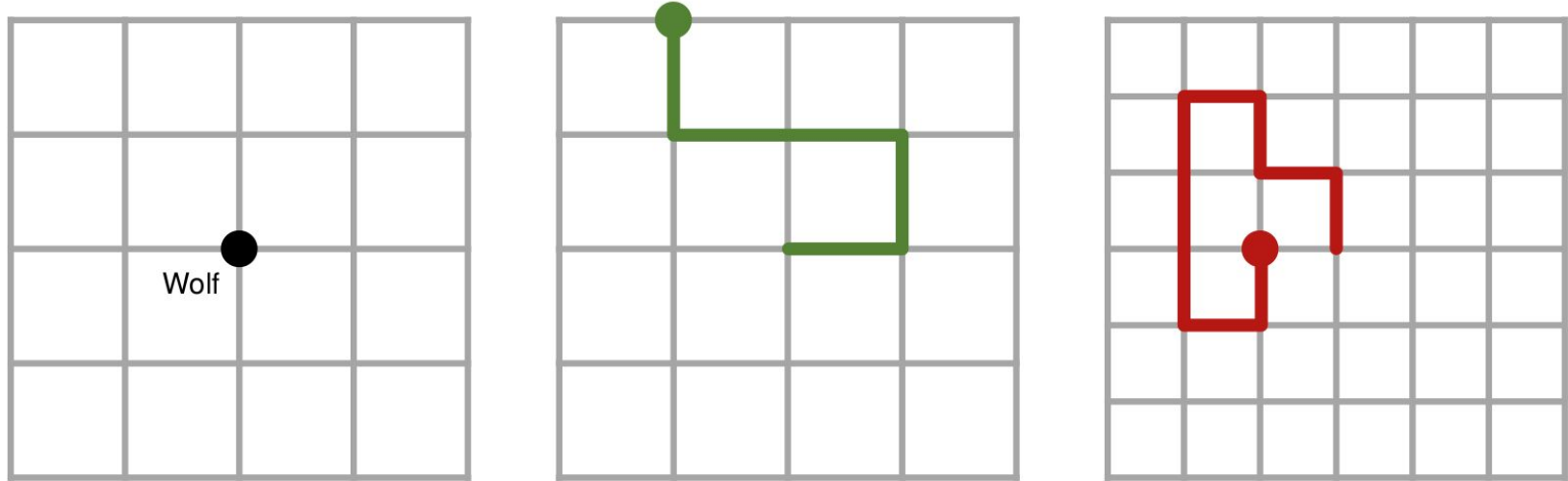


Abbildung 3: Wolf in der Stadt. Ausgangslage, erfolgreiche Flucht und fehlgeschlagene Flucht

Kahoot

Zusatzaufgabe

Gegeben sei die Klasse `LinkedList` aus der Vorlesung. (Auch als `.java` verfügbar). Implementieren Sie die Methode `LinkedList.trim(LinkedList other)`, die als Parameter eine andere `LinkedList other` entgegen nimmt. Sollte das Exemplar (von `LinkedList`) auf das die Methode `trim` angewendet wird, Knoten gemeinsam haben mit `other`, so wird der erste gemeinsame Knoten als Ergebnis zurückgegeben. Gibt es keine gemeinsamen Knoten (oder ist `other` null), so soll null zurückgegeben werden.

