252-0027-00: Einführung in die Programmierung

Übungsblatt 7

Abgabe: 14. November 2023, 23:59

Checken Sie mit Eclipse wie bisher die neue Übungsvorlage aus. Importieren Sie beide Eclipse-Projekte (das Projekt für den Bonus und das Projekt für die restlichen Aufgaben). *Vergessen Sie nicht, Ihren Programmcode zu kommentieren!*

Aufgabe 1: EBNF Wiederholung

In dieser Aufgabe erstellen Sie wieder EBNF-Beschreibungen. Alle Beschreibungen werden in der Text-Datei "EBNF.txt" abgelegt, welche sich in Ihrem Git-Repository befindet.

Verwenden Sie "<=" als Zeichen für "ist definiert als". Der Name der letzten Regel ist durch die Aufgabenbeschreibung vorgegeben, andere Namen können Sie frei wählen. Da reine Textdateien keine Kursivschrift unterstützen, stellen wir die Namen von Regeln zwischen < und >, also z.B. `<name>`.

1. Erstellen Sie eine EBNF Beschreibung `<xyz>`, 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"`

2. Erstellen Sie eine EBNF Beschreibung `<abc>`, 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"`

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.

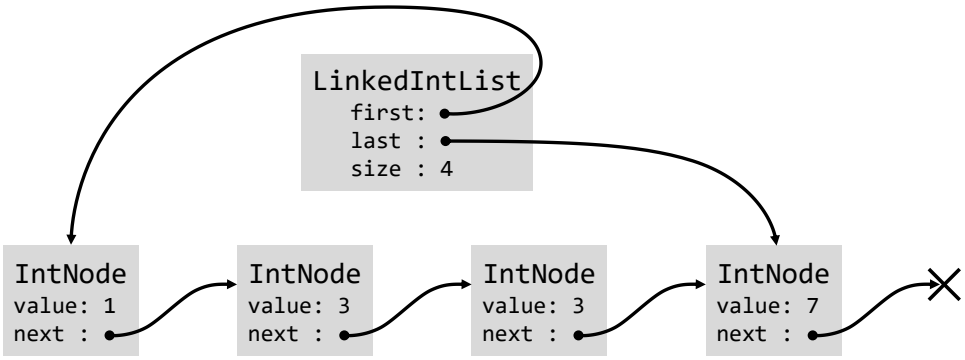


Abbildung 1: Verkettete Liste mit Werten 1, 3, 3, 7.

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).

a) Implementieren Sie die Klassen `LinkedIntList` und `IntNode` (in "LinkedIntList.java" bzw. "IntNode.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".

b) 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 solange Werte einlesen, bis der Benutzer eine ungültige Eingabe macht. Verwenden Sie dazu `Scanner.hasNextInt()`.

Um alle Werte auszugeben, soll Ihr Programm von Knoten zu Knoten "springen", angefangen beim ersten Knoten und solange, bis das Ende der Liste erreicht, d.h. der nächste Knoten gleich `null` ist. Sie können folgendes Code-Stück dafür verwenden:

```
for(IntNode n = list.first; n != null; n = n.next) { ... }
```

Aufgabe 3: Doubly-linked List

Die Liste, die Sie in der letzten Übung implementierten, ist "einfach verkettet", d.h., jeder Knoten hat nur eine Referenz auf den nächsten in der Liste. Doppelt verkettete Listen sind auch von hinten nach vorne verkettet, d.h., jeder Knoten hat auch eine Referenz zum vorherigen. Dies bringt einige Vorteile mit sich, z.B. lassen sich so Werte am Ende der Liste effizient entfernen und man kann die Liste einfach von hinten nach vorne durchgehen (über die Liste "iterieren"). Ausserdem kann man einfacher Werte aus dem Innern der Liste entfernen.

a) Sie finden in Ihrem Projekt eine `LinkedPersonList`, welche gleich funktioniert wie die Musterlösung zur `LinkedIntList`, einfach für `Person`-Objekte anstatt für `int`s. Erweitern Sie diese

einfach verkettete Liste zu einer doppelt verketteten Liste. Fügen Sie dafür ein `prev`-Feld zu `PersonNode` hinzu und ändern Sie die Methoden in `LinkedPersonList` wo nötig (oder vorteilhaft).

b) Fügen Sie eine `removeNode()`-Methode hinzu, welche ein gegebenes `PersonNode`-Objekt aus der Liste entfernt. Diese Methode kann verwendet werden, um einen Knoten innerhalb einer

```
for(PersonNode n = list.first; n != null; n = n.next) { ... }
```

Schleife zu entfernen.

c) Erweitern Sie die Tests in `LinkedPersonListTest`. Die vorgegebenen Tests sind für die einfach verkettete Liste ausgelegt. Fügen Sie wo nötig Test-Code hinzu und schreiben Sie eine Test-Methode für `removeNode()`. Testen Sie vor allem die neu hinzugekommene Konsistenz-Bedingung:

Für alle Knoten `n` gilt: `n.next == null || n.next.prev == n` (und analog für `n.prev`).

xkcd: Forgetting by Randall Munroe (CC BY-NC 2.5, modified)

Aufgabe 4: Rechnungen (Bonus!)

Achtung: Diese Aufgabe gibt Bonuspunkte (siehe "Leistungskontrolle" im `www.vvz.ethz.ch`). Die Aufgabe muss eigenhändig und alleine gelöst werden. Die Abgabe erfolgt wie gewohnt per Push in Ihr Git-Repository auf dem ETH-Server. Verbindlich ist der letzte Push vor dem Abgabetermin. Auch wenn Sie vor der Deadline committen, aber nach der Deadline pushen, gilt dies als eine zu späte Abgabe. Bitte lesen Sie zusätzlich die allgemeinen Regeln.

In dieser Aufgabe sollen Sie einen Teil des Systems implementieren, das für den lokalen Stromversorger die Rechnungen erstellt.

Vervollständigen Sie die `process`-Methode in der Klasse `Bills`. Die Methode hat zwei Argumente: einen `Scanner`, von dem Sie den Inhalt der Eingabedatei lesen sollen, und einen `PrintStream`, in welchen Sie die unten beschriebenen Informationen schreiben.

Ihr Programm muss nur korrekt formatierte Eingabedateien unterstützen. Ein Beispiel einer solchen Datei finden Sie im Projekt unter dem Namen "Data.txt". Exceptions im Zusammenhang mit Ein- und Ausgabe können Sie ignorieren.

Eine valide Eingabedatei enthält Zeilen, die entweder den Tarif, der angewendet werden soll, oder die Daten für den Stromverbrauch eines Kunden beschreiben. Der Verbrauch eines Kunden ist niemals grösser als 100000 Kilowattstunden.

Eine Tarifbeschreibung hat folgendes Format:

$$\texttt{Tarif}\,␣\,n\,␣\,l_1\,␣\,p_1\,\ldots\,l_n\,␣\,p_n$$

Folgendes gilt für die Parameter:

- `Tarif` (so geschrieben) ist ein Keyword, das angibt, dass die Zeile einen Tarif beschreibt.
- n is eine positive ganze Zahl, welche die Anzahl der Intervalle angibt, für welche ein Strompreis festgelegt ist.
- Auf n folgt eine Folge von n Paaren von ganzen Zahlen ($l_1\,␣\,p_1\,\ldots\,l_n\,␣\,p_n$). Die erste Zahl eines Paares gibt die Obergrenze des Intervalls an und die zweite den Preis für diesen Verbrauch; für ein i, so dass $1 \leq i \leq n$, ist l_i also der Verbrauch (in Kilowattstunden), bis zu welchem der Strompreis p_i (in Rappen pro Kilowattstunde) zur Anwendung kommt ($l_i > 0$ und $p_i \geq 0$). Die Paare sind jeweils mit einem Whitespace voneinander getrennt (und l_i und p_i jeweils voneinander auch).

Hier sind einige Beispiele für Tarifbeschreibungen:

1. `Tarif 1 100000 30`
   Es gibt ein Intervall und für jede Kilowattstunde müssen 30 Rappen bezahlt werden.
2. `Tarif 2 1000 10 100000 30`
   Es gibt zwei Intervalle. Die ersten 1000 Kilowattstunden kosten 10 Rappen pro Kilowattstunde. Der Rest kostet 30 Rappen pro Kilowattstunde.
3. `Tarif 3 100 40 1000 10 100000 30`
   Es gibt drei Intervalle. Die ersten 100 Kilowattstunden kosten 40 Rappen pro Kilowattstunde. Die nächsten 1000 Kilowattstunden kosten 10 Rappen pro Kilowattstunde. Der Rest kostet 30 Rappen pro Kilowattstunde.

Wenn ein Kunde im Jahr 2000 Kilowattstunden verbraucht, so beträgt die Rechung für das erste Beispiel 600 Franken, im zweiten Beispiel 400 Franken und 410 Franken im dritten.

Die Beschreibung des Stromverbrauchs eines Kunden hat folgendes Format:

$$ID\,␣\,v_{q_1}\,␣\,v_{q_2}\,␣\,v_{q_3}\,␣\,v_{q_4}$$

Hierbei gilt für die Parameter:

- ID ist eine positive ganze Zahl.
- v_{q_1} ist eine ganze Zahl, die den Verbrauch im ersten Quartal in Kilowattstunden angibt ($v_{q_1} \geq 0$).
- v_{q_2} ist eine ganze Zahl, die den Verbrauch im zweiten Quartal in Kilowattstunden angibt ($v_{q_2} \geq 0$).

- v_{q_3} ist eine ganze Zahl, die den Verbrauch im dritten Quartal in Kilowattstunden angibt ($v_{q_3} \geq 0$).
- v_{q_4} ist eine ganze Zahl, die den Verbrauch im vierten Quartal in Kilowattstunden angibt ($v_{q_4} \geq 0$).

Hier ist ein Beispiel für eine Verbrauchbeschreibung:

```
115 0 0 0 2000
```

Der Kunde mit ID 115 hat nur im vierten Quartal Strom verbraucht. Da waren es 2000 Kilowattstunden.

Ein einmal gelesener Tarif wird für alle Kunden angewendet, die nach dieser Tarifinformation in der Eingabedatei erscheinen. Wenn ein neuer Tarif erscheint, dann gilt der danach für die weiteren Kunden bis auf Weiteres. Sie können davon ausgehen, dass eine Kunden-ID nur einmal in der Eingabedatei vorkommen kann und dass die erste Zeile der Eingabedatei eine Tarifbeschreibung ist.

Die Methode `process` soll die Eingabedatei verarbeiten und für jeden Kunden eine Zeile

$$ID \text{␣} b$$

in den der Methode in `output` übergebenen `PrintStream` schreiben. ID ist die ID des Kunden (`int`) und b ist eine **ganze** Zahl, die die jeweilige Rechnung für den Jahresverbrauch **in Franken** angibt. (Zuerst muss der Jahresverbrauch berechnet werden, dann kann der entsprechende Tarif angewendet werden.) Berechnen Sie den Rechnungsbetrag und runden Sie das Resultat anschliessend (vor der Ausgabe, aber nach den Berechnungen) auf die nächste ganze Zahl. Sie können hierfür die Methode `Math.round(double a)` verwenden. Die Ausgabe darf keine weiteren Zeichen enthalten. Sie können den Betrag so ausgeben, wie er von der `println`-Anweisung herausgegeben wird, d.h. Sie brauchen das Ergebnis nicht zu formatieren.

In der Datei "BillsTest.java" finden Sie einen einfachen Test, um das Format Ihres Outputs zu testen.

Tipp: Sie können die Aufgabe ohne weitere Vorgaben implementieren. Wir empfehlen, dass Sie sich überlegen, was sinnvolle Klassen sein könnten und was für Teilaufgaben (die dann als Methode implementiert werden können) zweckmässig sind.

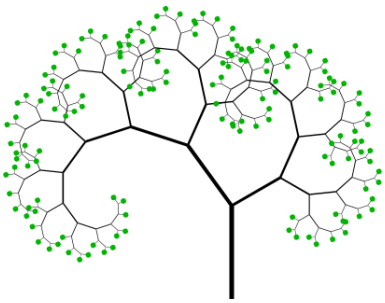
Aufgabe 5: Rekursion

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

In jedem Schritt geschehen zwei Dinge:



1. Zeichnen des aktuellen Segments
2. Falls $l < 10$, endet die Rekursion und am Ende des Segments wird ein Blatt gemalt.

   Andernfalls 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.

Erstellen Sie ein Programm "Recursion.java", welches einen Baum nach den zuvor beschriebenen Regeln in ein Fenster zeichnet.

1. Schreiben Sie eine rekursive Methode `drawTree`, welche einen Zeichenschritt durchführt. Zusätzlich zu den oben beschriebenen Parametern sollte diese Methode einen Parameter für das `Window`-Objekt haben.
2. Starten Sie die Rekursion in der `main`-Methode mit den Argumenten $x_0 =$ `SIZE/2`, $y_0 =$ `SIZE`, $l_0 = 100$, $\alpha_0 =$ `Math.PI/2`, wobei `SIZE` die Grösse des Fensters ist.
3. Experimentieren Sie mit den verschiedenen Zahlen, z.B. mit der Mindestlänge oder mit den Winkeländerungen.

Tipp: Sie können den Zeichenvorgang animieren, indem Sie zwischen Teil 1 und 2 jedes Zeichenschrittes `refresh(100)` auf das `Window`-Objekt aufrufen.