252-0027-00: Einführung in die Programmierung

Übungsblatt 7

Abgabe: 12. November 2019, 10:00

Checken Sie mit Eclipse wie bisher die neue Übungs-Vorlage aus. *Vergessen Sie nicht, Ihren Programmcode zu kommentieren!*

Aufgabe 1: 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 2: Split (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.

Auf dem letzten Übungsblatt haben Sie eine Linked List für Integer implementiert. In dieser Aufgabe fügen Sie einer ähnlichen Linked List eine weitere Methode `split` hinzu, welche einen Integer als Parameter nimmt und eine Linked List zurückgibt. Die Methode entfernt aus der Liste auf der sie aufgerufen wird alle Elemente, welche grösser als das Argument sind, und fügt diese entfernten Elemente in der gleichen Reihenfolge der zurückgegebenen Liste hinzu.

Wir verwenden dafür `SpecialIntLinkedList` und `SpecialIntNode` anstatt `IntLinkedList` und `IntNode`. Der Unterschied ist, dass `SpecialIntNode` neben einem `next` Feld auch ein Feld `oldNext` hat. Dieses Feld soll nach einem Aufruf von `split` den `SpecialIntNode` enthalten, welcher dem Nachfolger aus der Ursprünglichen Liste entspricht.

Abbildung 1 zeigt ein Beispiel: Wir beginnen mit einer Liste `This`, welche die Werte 3, 7, 2, 4, und 5 enthält. Die grünen Pfeile repräsentieren das `next` Feld, wobei kein Pfeil gezeichnet ist, wenn das Feld `null` enthält. Danach wird `split(4)` auf `This` aufgerufen, wobei die Liste `Result` zurückgegeben wird. `This` enthält danach die Werte kleiner oder gleich 4, nämlich 3, 2, und 4, während `Result` die Werte grösser als 4 enthält, nämlich 7 und 5. Zusätzlich zeigt das Feld `oldNext`, welches mit roten Pfeilen repräsentiert wird, auf den entsprechenden Nachfolger aus der ursprünglichen Liste, so dass man erneut 3, 7, 2, 4, und 5 enthält, wenn man den roten Pfeilen folgt.

Die `SpecialIntNode` Instanzen müssen nicht den gleichen Instanzen aus der Ursprungsliste entsprechen, das heisst, dass neue `SpecialIntNode` Instanzen erzeugt werden dürfen.

Vervollständigen Sie die Methode `split()` in der Klasse `SpecialLinkedIntList`. Die Methode soll, wie oben definiert, die Liste splitten. In der Datei "Split.java" befindet sich ein einfacher Klient um die Methode auszuführen und in der Datei "SplitTest.java" finden Sie ein paar einfache Tests.

Bitte ändern sie die Klassen `SpecialIntLinkedList` und `SpecialIntNode` nur wo vorgesehen ab.

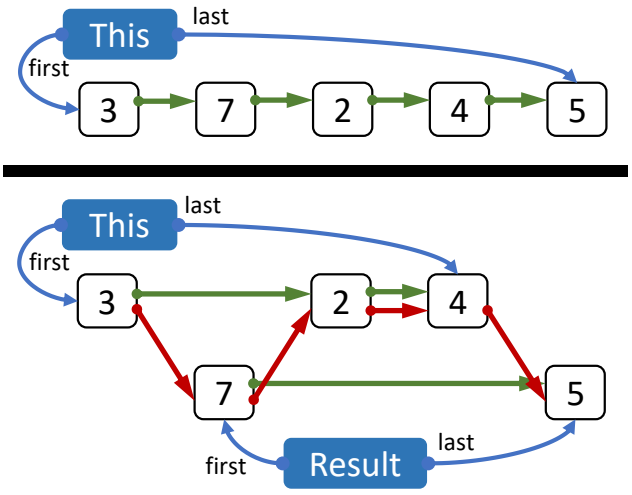


Abbildung 1: Liste vor und nach dem Aufruf von `This.split(4)`. Die Liste aus welcher `split` aufgerufen wird ist `This` und die zurückgegebene Liste ist `Result`. Grüne Pfeile repräsentieren das `next` Feld und rote Pfeile repräsentieren das `oldNext` Feld. Nicht gezeigte Pfeile entsprechen `null`.

Aufgabe 3: 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.

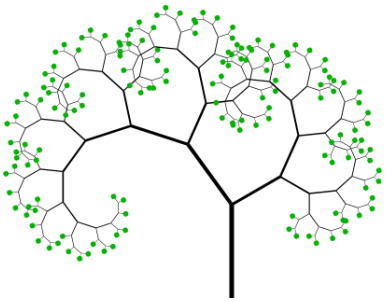
   Beispiele legaler Symbole: `"XY", "XXYZZZ", "", "XYYXZZZX"`

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 4: 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.

   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.

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.