

# Miniprojekt 3 – Listen, Stapel und Bäume








## Hinweis

Die Klasse Main enthält eine main-Methode mit der Sie die Implementierung der Aufgaben 1-3 testen können. Für dieses und zukünftige Miniprojekte wäre es aber grundsätzlich sinnvoller, wenn Sie Ihren eigenen Code zum Testen der Funktionen ihrer Datenstrukturen schreiben. Um eine sinnvolle Menge an Tests zu definieren, müssen Sie sich die verschiedenen Fälle aller Operationen vor Augen führen. Die Fähigkeit diese Fälle zu erkennen und im Kopf (d.h. ohne Play-Button in Eclipse) durchzuspielen ist ein wichtiges Merkmal guter Entwickler. Da die Prüfung ohne Rechnerunterstützung abläuft, ist diese Fähigkeit auch wichtig um hohe Punktzahlen (sprich gute Noten) zu erreichen.

Falls Ihnen das zu kompliziert oder zu aufwändig erscheint, sollten Sie zumindest folgendes Vorgehen beim Lösen von Programmieraufgaben umsetzen: Nach der Fertigstellung einer Funktion und nach jeder Änderung, die sie gerne Testen würden, **lesen Sie sich zuerst ihren Code nochmal durch**. **Spieren Sie dabei im Kopf den Methodenaufwurf bzw. die resultierende Sequenz von Anweisungen ab**, die durch unterschiedliche Parameterwerte entsteht, und vergleichen Sie den Ablauf mit Ihrer Erwartung. **Nutzen Sie niemals den Test zur Ersten oder gar zu alleinigen Validierung** ihres Programms. **Die mächtigste Fehlersuchmaschine** (und die einzige, die bei der Prüfung zulässig ist) **sind Sie selbst!** Falls Sie es schaffen, Ihren inneren Schweinehund, der immer nur ganz schnell mal auf Play drücken will, konsequent zu besiegen, wird sich die Zahl der „Bugs“ die Sie „schnell noch fixen“ müssen spürbar und nachhaltig reduzieren.

## Aufgabe 1 – Listen

In dieser Aufgabe geht es um die Implementierung einer Liste zur Speicherung einer beliebigen Menge von Zeichenketten. Hierfür sollen Sie zunächst zwei unterschiedliche Implementierungsmöglichkeiten umsetzen und anschließend miteinander vergleichen. Beide Implementierungsmöglichkeiten sollen die selbe Funktionalität anbieten, die in der Schnittstelle **StringList** wie folgt definiert ist:

-  **void appendString(String text)** – Fügt den übergebenen String am Ende der Liste, d.h. als letztes Element, ein.
-  **String getStringAt(int i)** – Liefert den String an der Stelle i zurück oder null falls i ungültig ist.
-  **void insertStringAt(int i, String text)** – Fügt den String als das i-te Element in die Liste ein. Die bestehenden Elemente an den Stellen  $\geq i$  werden um eins nach hinten verschoben.
-  **void insertStringListAt(int i, StringList list)** – Fügt eine Liste von Strings an der i-ten Stelle ein. Die bestehenden Elemente an den Stellen  $\geq i$  werden um die Länge der Liste verschoben.
-  **String replaceStringAt(int i, String text)** – Ersetzt das i-te Element der Liste durch den String und liefert den vorherigen Wert der Liste zurück.
-  **String removeStringAt(int i)** – Entfernt das i-te Element der Liste und liefert den entfernten String zurück. Etwaige Elemente an Positionen  $> i$  werden um eins nach vorne geschoben.
-  **String getFirstString()** – Liefert das erste Element der Liste zurück oder null, falls die Liste keine Elemente enthält.

- **String getLastString()** – Liefert das letzte Element der Liste zurück oder null, falls die Liste keine Elemente enthält.
- **StringList reverseStringList()** – Erzeugt eine neue Liste mit gleichem Inhalt aber invertierten Indizes, d.h. aus der Liste (A, B, C, D) wird die Liste (D, C, B, A) erzeugt.
- **int getIndexOfString(String text, int startIndex)** – Sucht den übergebenen String in der Liste beginnend an der Stelle startIndex und liefert den ersten Index zurück an dem der String gefunden wird oder -1, falls die Teilliste, die über startIndex definiert ist, den String nicht enthält.
- **int countElements()** – Liefert die Anzahl der Elemente in der Liste zurück.
- **String[] toStringArray()** – Liefert den Inhalt der Liste als String Array zurück. Die Reihenfolge im Array soll dabei der Reihenfolge der Elemente in der Liste entsprechen. Das Array soll aber nicht mit der Liste verknüpft sein, d.h. es soll eine eigenständige Kopie sein.

Bei Ihrer Implementierung können Sie davon ausgehen, dass die Strings in der Liste nicht null sind. Weiterhin können Sie die Annahmen über die Indizes (i und startIndex) treffen, die in den Kommentaren der Schnittstelle definiert sind.

### Implementierung der Liste auf Basis eines Arrays

Eine einfache Möglichkeit zur Implementierung einer Liste ist die Nutzung eines Arrays. Da Arrays in Java jedoch eine unveränderliche Größe haben, muss bei allen Vorgängen, die die Länge der Liste verändern (z.B. appendString, removeStringAt, etc.), der Inhalt der Liste in ein neues Array mit entsprechender Größe an die richtige Stelle(n) kopiert werden. Implementieren Sie die Methoden der Klasse **ArrayStringList** so dass diese den aktuellen Zustand der Liste in dem bereits vorhandenen Feld mit dem Namen **items** speichert.

**Tip:** Da Sie für diese Aufgabe häufig die Inhalte von Position x bis Position y aus einem Ausgangsarray in ein Zielarray an der Stelle z kopieren müssen, können Sie sich Tipparbeit sparen indem sie hierfür zunächst eine entsprechende Methode implementieren.

### Implementierung der Liste als (einfach)verkettete Liste

Um die aufwändigen Kopiervorgänge einer Array-basierten Liste zu vermeiden, kann man anstelle eines Arrays auch einzelne Listenelemente verwenden, die mit Hilfe einer Variable auf ihren Nachfolger verweisen. Die Liste besteht dann konzeptionell aus einer Verkettung von Elementen. Um auch leere Listen mit Hilfe eines Objekts repräsentieren zu können, wird diese Verkettung in der Regel hinter einer Fassade versteckt, die lediglich auf das erste Element der Liste verweist. Ist die Liste leer, ist dieser Verweis null. Durch diesen Trick (also die Fassade) müssen die Nutzer der Liste nicht immer den Fall „Liste ist leer“ gesondert behandeln.

Die Klasse **LinkedList** setzt diese Struktur mit Hilfe der inneren Klasse **LinkedList.Item** um. Ein **Item** besteht aus einem Verweis auf das nächste Element (s. **next**) und dem zugehörigen String (s. **string**). Die Klasse **LinkedList** enthält einen Verweis auf das erste Element der Liste (s. **head**). Ist die Liste leer, ist head gleich null. Die Methode **getFirstItem** ist bereits implementiert. Implementieren Sie die verbleibenden Methoden der Schnittstelle **StringList**.

**Tip:** Für einige Operationen wie z.B. das Entfernen eines Elements, ist es erforderlich, dass Sie sich bei der Iteration über die Elemente nicht nur das derzeitige Element, sondern auch den Vorgänger merken. Sind sie dann beim zu entfernenden Element angekommen, müssen sie den next-Verweis im Vorgänger auf den Nachfolger setzen. Der Grund hierfür ist, dass eine einfach verkettete Liste nur den next-Verweis mit sich führt und man deshalb nur vorwärts durch die Liste laufen kann.

Um dieses Problem zu vermeiden könnte man auch einen Verweis auf den Vorgänger in Item speichern. Diesen müsste man dann natürlich bei Änderungen entsprechend nachführen. Diese Idee

also in der  
Aufgabe „from“  
& for(i=from;...)

bezeichnet man häufig als „zweifach verkettete Liste“ oder auch „mehrfach verkettete Liste“ (engl. Double Linked List) im Gegensatz zur „einfach verketteten Liste“ (engl. Single Linked List), die man in der Regel auch nur „verkettete Liste“ (engl. Linked List) nennt.

### Vergleich der Implementierungen (ohne Jack)

Vergleichen Sie die folgenden Methoden aus `ArrayStringList` und `LinkedListStringList` in Bezug auf den Speicherbedarf und die Laufzeitkomplexität:

- **`String getStringAt(int i)`**
- **`void insertStringAt(int i, String text)`**
- **`String removeStringAt(int i)`** –

Gegeben sind zwei Programme A und B. Das Programm A erstellt eine Liste mit ca. 50 Elementen und greift dann ca. 1 Million Mal auf zufällig ausgewählte Elemente der Liste zu. Das Programm B erstellt eine Liste mit 1 Million Elementen und arbeitet diese dann ab. In jedem Verarbeitungsschritt wird dafür das erste Element zunächst aus der Liste entfernt und dann verarbeitet. Welche Ihrer Listenimplementierungen würden Sie für das Programm A und das Programm B wählen?

Sollten Sie sich damit unsicher fühlen, diskutieren Sie Ihre Lösung mit Ihren Kommilitonen oder besuchen Sie die Beratungsstunde.

## Aufgabe 2 - Stapel

Ähnlich wie eine Liste ist auch ein Stapel (engl. Stack) eine Datenstruktur zur Verwaltung einer veränderlichen Anzahl an Elementen. Im Gegensatz zur Liste bietet er aber in der Regel primär eine push- und eine pop-Operation an. Die Push-Operation legt ein neues Element auf den Stapel. Die Pop-Operation entfernt das zuletzt auf den Stapel gelegte Element und liefert es zurück. (In manchen Bibliotheken findet man noch peek mit dem man das oberste Element auf dem Stapel lesen kann, ohne es vom Stapel zu nehmen).

Die Implementierungsmöglichkeiten eines Stapels sind grundsätzlich die selben wie bei einer Liste und man kann auch eine Liste zur Implementierung eines Stapels verwenden. Hierfür könnte man z.B. `push(x)` durch `insertStringAt(0, x)` und `pop()` durch `removeStringAt(0)` implementieren. Für diese Aufgabe konzentrieren wir uns jedoch lediglich auf die Array-basierte Variante, deren Kopierproblem wir ein wenig optimieren wollen. Basis für die Optimierung ist folgende Überlegung:

Die Größenänderung des Arrays ist mit einem vergleichsweise hohen Aufwand verbunden. In der Liste in Aufgabe 1 findet eine Größenänderung bei allen Operationen statt, die die Zahl der Elemente in der Liste ändern. Um dies zu vermeiden können wir für unseren Stapel die Anzahl der Elemente (im Nachfolgenden **size**) von der aktuellen Größe des Arrays (im Nachfolgenden **capacity**) trennen. Natürlich muss dabei die Größe des Arrays mindestens der Anzahl der Elemente entsprechen. Sie könnte aber auch durchaus größer sein. Ist dies der Fall, können weitere push-Operationen ohne Kopiervorgang ausgeführt werden. Erst wenn die Zahl der Elemente die Größe des Arrays übersteigt, muss ein neues Array erzeugt werden und der Inhalt kopiert werden. In dem Fall können wir dann vorbeugend die Größe nicht nur um 1 sondern gleich um  $x$  erhöhen. Damit können wir dann zusätzlich  $x-1$  weitere push-Operationen ohne Kopiervorgang ausführen.

Implementieren Sie auf Basis dieser Idee die push-Methode in der Klasse **`StringStack`**. Hierfür müssen Sie in der Variable **size**, die aktuelle Anzahl der Elemente auf dem Stack mitführen (s. **`getSize()`**). Die **capacity** ergibt sich über die Länge des Arrays (s. **`getCapacity()`**). Wird eine push-Operation ausgeführt, die die **capacity** übersteigt, erweitern Sie zunächst die **capacity** um den Wert der Variable **increment**, bevor sie das Element auf den Stapel legen.

Implementieren Sie danach die vorgegebene `pop`-Methode, so dass diese das oberste Element auf dem Stapel zurück liefert und die **capacity** reduziert, falls die Zahl der ungenutzten Elemente dem doppelten Wert von **increment** entspricht. Durch diese Kapazitätsreduktion stellen wir sicher, dass die Platzverschwendung (die durch die Abwägung zwischen Kopieraufwand und Speicherbedarf entsteht) auf diesen Wert begrenzt ist.

Beschreiben Sie kurz, warum sich diese Idee nicht so einfach auf die Implementierung der Schnittstelle **StringList** übertragen lässt.

**Tipp:** Gibt es Operationen, bei denen diese Optimierung nicht so ohne weiteres greift? Falls ja welche und finden Sie eine Gemeinsamkeit?

## Aufgabe 3 – Binärbäume

In dieser Aufgabe geht es um die Implementierung eines geordneten Binärbaums, der einfache Zahlenwerte speichern soll. Jede Zahl soll dabei höchstens einmal vorkommen. Die Klasse **BinaryIntTree** definiert hierfür eine Reihe von Methoden, die die Datenstruktur unterstützen soll. Darüber hinaus definiert sie eine innere Klasse **Node**, die einen einzelnen Knoten im Baum repräsentieren soll. Entsprechend enthält ein Node den zu speichernden Zahlenwert (**value**), sowie einen Verweis auf ein linkes (**leftChild**) und ein rechtes Kind (**rightChild**).

Genau wie bei **LinkedStringList**, handelt sich bei **BinaryIntTree** also wieder um eine Fassade, die die eigentliche Datenstruktur (Node) kapselt und somit den Nutzer des Baums vor der Behandlung des Sonderfalls „Baum hat keine Wurzel“ schützt. Den Einstiegspunkt in den Baum liefert entsprechend die **root** Variable in der Klasse **BinaryIntTree**, die null ist, falls der Baum noch keine Knoten enthält.

Implementieren Sie jetzt die folgenden Methoden in **BinaryIntTree**:

- **boolean insertValue(int value)** – Diese Methode fügt einen Wert in den Baum ein, sofern dieser nicht bereits vorhanden ist. Die Operation soll dabei sicherstellen, dass der Baum geordnet bleibt, d.h. für jeden Knoten gilt  $\text{leftChild.value} < \text{value} < \text{rightChild.value}$ . Wurde der Baum durch den Aufruf verändert, soll die Methode `true` ansonsten `false` zurückgeben.
- **boolean containsValue(int value)** – Diese Methode soll prüfen, ob ein Wert im Baum bereits vorhanden ist. Die Implementierung soll sich dabei die Ordnung im Baum zu Nutze machen um die Suche auf relevante Teile zu beschränken.
- **int getNodeCount()** – Diese Methode soll die Zahl der Knoten im Baum zurückliefern.
- **int[] toIntArray()** – Diese Methode soll die Knoten im Baum in einem sortierten Array ausgeben. Die Sortierung soll dabei direkt während der Befüllung erfolgen. **Tipp:** Hierfür brauchen Sie eine in-order Traversierung.

**Tipp:** Die Traversierung von Bäumen, die sie für diese Methoden brauchen sind ein Lehrbuchbeispiel für die Eleganz von Rekursion zur Lösung bestimmter Aufgaben. Die Zahl der Knoten ergibt sich z.B. aus  $1 + \text{Anzahl der Knoten im linken Teilbaum} + \text{Anzahl der Knoten im rechten Teilbaum}$ . D.h. wenn Sie **getNodeCount** in der Klasse **Node** rekursiv implementieren und dann in der Klasse **BinaryIntTree** nur die `root`-Prüfung auf null machen und den Aufruf ansonsten an die Wurzel weiterleiten, bekommen Sie eine sehr kompakte und leicht verständliche Lösung. Dieses Vorgehen sehen Sie auch in der vorgegebenen Implementierung von **toIntArray**, die eine noch zu implementierende Methode für die Traversierung aufruft.

### Zum Nachdenken

Implementieren Sie jetzt die folgenden Methoden in **BinaryIntTree**:

- **boolean isFull()** – Diese Methode soll true zurückliefern, falls jeder Knoten entweder 0 oder 2 Kinder besitzt (oder der Baum keine Knoten hat). In anderen Worten: alle Knoten, die keine Blattknoten sind, haben 2 Kinder.
- **boolean isPerfect()** – Diese Methode soll true zurückliefern, falls alle Knoten die keine Blätter sind 2 Kinder haben und alle Blattknoten auf der selben Ebene sind (d.h. die zwischen Knoten und Wurzel ist für alle Kinder gleich). **Tipp:** Was gilt dann für die minimale und die maximale Ebene der Kinder?