Theorie

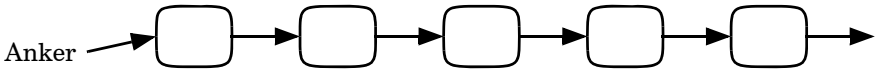
1. Lineare Listen

Eine verkettete Liste (*linked list*) ist eine dynamische Datenstruktur zur Speicherung von Objekten (in Java zur Speicherung von *Referenzen* auf Objekte). Insofern ist eine verkettete Liste einem Array von Referenzen ähnlich. Der grosse Unterschied besteht aber darin, dass das Array eine fixe maximale Länge besitzt, während eine verkettete Liste wachsen und schrumpfen kann, sich eben dynamisch verhält. Auf ein Element des Arrays kann über den Index direkt zugegriffen werden, da die Abspeicherung der Elemente in aneinander grenzenden Speichereinheiten erfolgt und somit die Speicheradresse direkt berechnet werden kann. Im Gegensatz dazu speichert die verkettete Liste die Elemente an beliebigen Orten auf dem Heap. Daher kann auf Listenelemente nicht über einen Index zugegriffen werden, stattdessen müssen sie gesucht werden. Ausgehend vom Ausgangspunkt (Anker) kann jeweils von einem Element auf das nächste Element (bei doppelt verketteten Listen auch noch auf das vorangehende Element) zugegriffen werden.

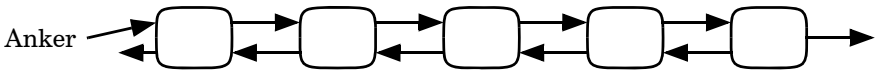
Verkettete Listen eigenen sich dafür, eine unbekannte Anzahl von Objekten abzuspeichern, wenn kein direkter Zugriff auf die einzelnen Objekte verlangt wird und die Anzahl der Objekte stark schwankt. Sie sind geeignet zur Speicherung von Teilen eines Ganzen, beispielsweise von Bildobjekten in einer Zeichnung. Sie liegen oft Datenstrukturen wie Stack (*LIFO = Last In, First Out*) oder Queue (*FIFO = First In, First Out*) zu Grunde, wenn die Zahl ihrer Elemente grossen Schwankungen unterworfen ist.

Es gibt einfach verkettete Listen und doppelt verkettete Listen. Im ersten Fall kann nur in einer Richtung durch die Liste traversiert werden, im anderen Fall in beiden Richtungen. Verkettete Listen können als lineare Listen oder Ringlisten (das Ende ist wieder mit dem Anfang verkettet) ausgeführt werden. Sie können ein oder mehrere Einstiegspunkte haben. Oft werden am Anfang und/oder am Schluss einer Liste so genannte «dummy»-Elemente angefügt, welche keine Nutzinformation enthalten. Dies hat den Vorteil, dass in den Methoden zur Bearbeitung der Liste keine Fallunterscheidungen für Randelemente gemacht werden müssen.

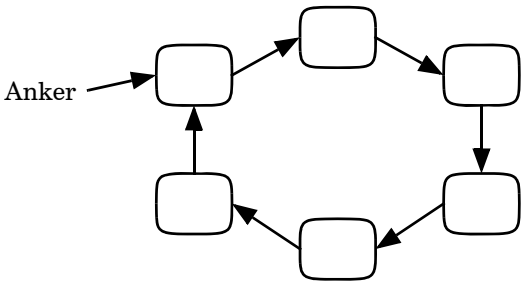
einfach verkettete Liste



doppelt verkettete Liste



Ringliste



Jedes Listenelement muss neben der Nutzinformation noch die notwendigen Referenzen zur Verkettung enthalten. Oft setzt sich deshalb ein Listenelement aus einem Objekt, welches die Verkettungsinformation enthält, und einer Referenz auf ein Nutzobjekt zusammen.

Beim Verarbeiten von Listen muss darauf geachtet werden, dass jedes Listenelement an einer direkt oder indirekt zugänglichen Referenz befestigt bleibt. Bei Umgebungen, welche keinen Garbage-Collector haben, muss darauf geachtet werden, dass die Listenbearbeitung keine Memory-Leaks (Speicherlecks) hinterlässt.

2. Information Hiding

Alle zuvor eingeführten Datenstrukturen dienen der dynamischen Verwaltung von Objektmengen. Die Benutzerin einer dynamischen Datenstruktur möchte sie verwenden können, ohne etwas über deren inneren Aufbau wissen zu müssen. Selbstverständlich braucht sie eine Dokumentation, welche ihr Auskunft über die vorhandenen Methoden und deren Verhaltensweisen gibt (*API Doc*). Die internen Implementierungsdetails werden jedoch von der Benutzerin versteckt. Man spricht in diesem Zusammenhang von *Information Hiding*.

Konkret für eine lineare Liste bedeutet dies: Jedes Element der Liste besteht einerseits aus den eigentlichen *Nutzdaten*, anderseits aus der *Verkettungs-Information*, welche die Elemente miteinander verbindet. Die Benutzerin interessiert sich nur für die korrekte Speicherung der *Nutzdaten*, braucht aber nichts über die Gestalt eines Elements, insbesondere über die *Verkettungs-Information* zu wissen. Aus diesem Grund werden die Elemente selbst als *innere Klassen* in der Listenklasse versteckt. Es stellt sich nun noch die Frage, ob eine *statische innere Klasse* oder eine *Elementklasse* verwendet werden soll: Aus *logischer Sicht* bietet sich eine Elementklasse an, da jedes Element an ein bestimmtes Listen-Objekt gebunden ist. Aus der *Sicht der Effizienz* hingegen ist eine statische Klasse besser, da deren Objekte keine implizite Referenz aus das äussere Objekt mitführen, was ja auch gar nicht nötig ist.

Solange die Benutzerin nur einzelne Objekte in der dynamischen Datenstruktur abspeichern und diese später wieder lesen möchte, reicht es vollkommen aus die notwendigen Speicher- und Lesebefehle zu kennen. Der innere Aufbau der Datenstruktur hat zwar einen Einfluss auf die Geschwindigkeit, ist aber zweitrangig aus funktionaler Sicht. Anders sieht es jedoch aus, wenn die Benutzerin gezielt *alle* Objekte der Liste besuchen und eventuell lesen möchte. Entweder weiss sie dann über den internen Aufbau der Datenstruktur Bescheid (wie zum Beispiel beim Array) oder sie erhält ein Instrumentarium, um alle Objekte der Reihe nach besuchen zu können. Da ein Offenlegen der internen Implementierung zu diesem Zweck das Information-Hiding-Prinzip torpedieren würde, ist es wohl ratsam, ein spezielles Instrumentarium für das Durchlaufen aller Objekte anzubieten. Ein solches Instrumentarium könnten zum Beispiel Iteratoren sein.

3. Iteratoren

Ein Iterator erlaubt es Ihnen, alle Elemente einer dynamischen Datenstruktur nacheinander zu durchlaufen (iterieren). Dies ist analog zum Durchlaufen aller Elemente eines Arrays, wo Sie eine Laufvariable so initialisieren, dass Sie zuerst aufs erste Element zugreifen können und dann in einer **`for`**-Schleife jeweils die Laufvariable erhöhen, bis Sie schliesslich am Ende des Arrays angelangt sind.

Das Collection-Framework von Java verwendet ein solches Iteratorkonzept: Mit der Methode **`listIterator()`** aus der abstrakten Basisklasse **`AbstractList`** erhalten Sie eine Referenz auf ein neu erzeugtes Objekt der Klasse **`ListIterator<T>`**. Dieses neue Iterator-Objekt zeigt standardmässig auf den Anfang der Liste Sie können es anschliessend verwenden, um mit der Methode **`hasNext()`** zu überprüfen, ob schon alle Elemente durchlaufen worden sind und falls nicht, mit der Methode **`next()`** das nächste Element zu durchlaufen und zu bekommen.

```
List<String> l;
// Liste l erzeugen und mit Daten (Strings) füllen

ListIterator<String> it = l.iterator();
while(it.hasNext()) {
  System.out.println(it.next());
}
```

4. Stack

Der Stack (Stapel) ist eine dynamische Datenstruktur, welche dadurch charakterisiert ist, dass man nur auf das oberste Element des Stapels zugreifen (**`top`**), ein neues Element auf den Stapel legen (**`push`**) oder das oberste Element des Stapels entfernen (**`pop`**) kann.

Ein Stack kann basierend auf einer einfach verketteten Liste aufgebaut werden. Üblicherweise zeigt der Anker auf das oberste Element des Stapels, d.h. die Operation **`push()`** entspricht einem Einfügen am Anfang der Liste und die Operation **`pop()`** einem Entfernen am Anfang der Liste.