

## 2. Listen

### 2.1. Motivation

An Arrays als Implementierungswerkzeug für Collections stört, dass sich der Speicheraufwand für das Array nicht automatisch an den für die Collection tatsächlich aktuell benötigten Speicherbedarf angleicht. Man muss also eventuell mit einer *statischen* Datenstruktur *dynamischen* Bedarf abdecken.

Eine dynamische Datenstruktur, deren Speicheraufwand unmittelbar am Bedarf orientiert ist, sind *verlinkte Listen*. Diese werden mittels Hilfsobjekten und Referenzen dazwischen aufgebaut.

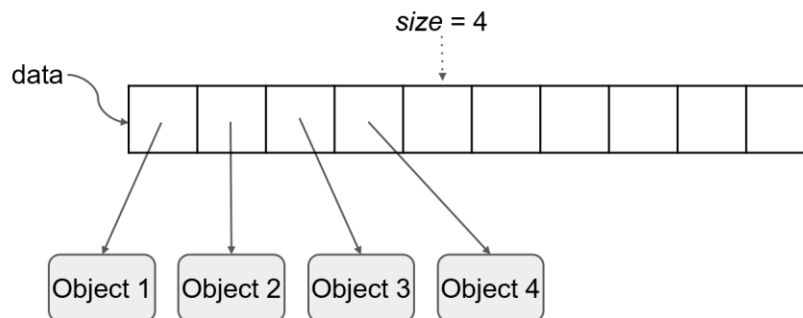
Ein weiterer Vorteil solcher verlinkter Listen ist, dass beim Hinzufügen oder Entfernen von Elementen keine Elemente weiter hinten in der Liste verschoben werden müssen.

### 2.2. Lernziele

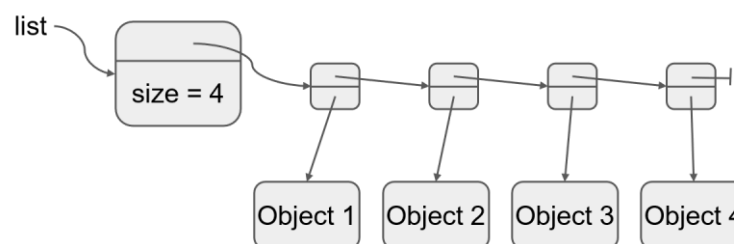
- Sie können verlinkte Listen in Java implementieren.
- Sie können mehrere Implementierungsvarianten mit ihren jeweiligen Vor- und Nachteilen angeben und anhand gegebener Anforderungen abwägen.
- Sie können verlinkte Listen einsetzen, um Datenstrukturen mit speziellen Zugriffsregeln wie *Stack* und *Queue* zu implementieren.

### 2.3. Collections mit Linked Lists

Collections mit Arrays programmiert sehen im Speicher etwa so aus:



Die Speicherplätze für die Referenzen auf die Elemente sowie die Grösse der Collection sind hier statisch. Variiert die Grösse der Collection über die Zeit hinweg stark, bietet sich eine andere Speicherform an: Zu einer Liste verlinkte Knoten (auch genannt Nodes).



Die für eine solche Datenstruktur nötigen Klassen in Java sehen z.B. so aus (noch ohne Operationen):

```
public class LinkedList<E> implements List<E> {
    private int size = 0;
    private Node<E> first;

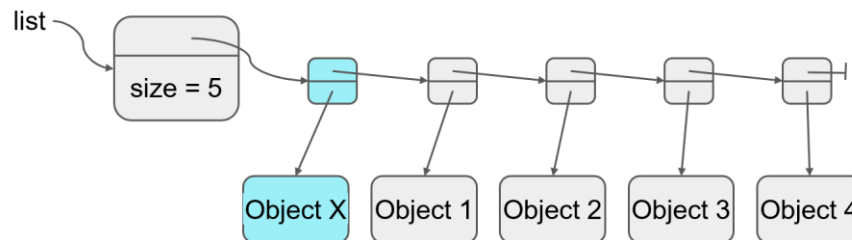
    private static class Node<E> {
        private final E elem;
        private Node<E> next;

        private Node(E elem) { this.elem = elem; }
        private Node(E elem, Node<E> next) { this.elem = elem; this.next = next; }
    }
}
```

Um mit dieser Datenstruktur das Interface *Collection* zu implementieren, muss man überlegen, wie die Operationen *boolean add(E e)*, *boolean contains(Object o)* und *boolean remove(Object o)* realisiert werden können. Wir legen im Folgenden die Bag-Semantik und unsortierte Speicherung zu Grunde.

### 2.3.1. boolean add(E e)

Wir beginnen mit dem Einfügen eines Objekts in eine bestehende Liste. Die einfachste Möglichkeit wäre es, das neue Objekt am Anfang der Liste einzufügen. Dazu wird eine neue Node erstellt und es werden drei Zeiger neu gesetzt (*first* der Liste; *elem* und *next* der neuen Node). Zusätzlich wird die Variable *size* der Liste inkrementiert. Die Laufzeitkomplexität für das Einfügen am Anfang einer Liste beträgt  $O(1)$ .



In Java:

```
public boolean add(E e) {
    Node<E> node = new Node<E>(e, null);
    node.next = first;
    first = node;
    size++;
    return true;
}
```

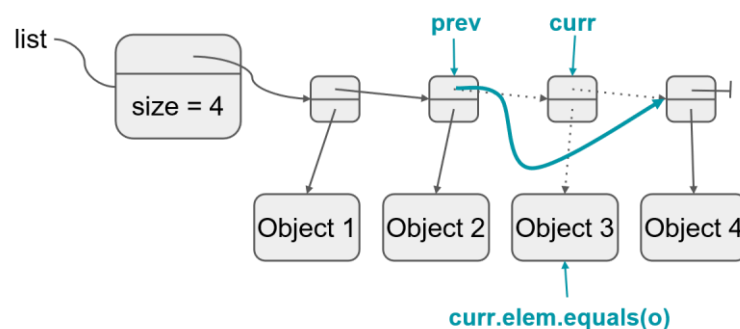
### 2.3.2. boolean contains(Object o)

Für die Suche nach einem bestimmten Objekt in der Liste iterieren wir von vorne nach hinten über alle Nodes. Wir machen also eine *lineare Suche*. Sobald wir das Objekt finden, brechen wir ab. Laufen wir über die Liste hinaus, ist das Objekt nicht in der Liste. Eine lineare Suche hat Komplexität  $O(n)$ .

### 2.3.3. boolean remove(Object o)

Das Löschen eines Objekts aus einer Liste ist etwas komplizierter. Damit wir alle Zeiger wieder richtig setzen können, müssen wir während der Suche nach dem zu löschenden Objekt nicht nur das aktuelle Objekt in der Liste (*current*) kennen, sondern auch dessen Vorgänger (*previous*). Sobald wir diese gefunden haben, können wir den *next*-Zeiger von *previous* umhängen auf den Nachfolger von *current*. Anschließend müssen wir noch die Variable *size* der Liste dekrementieren.

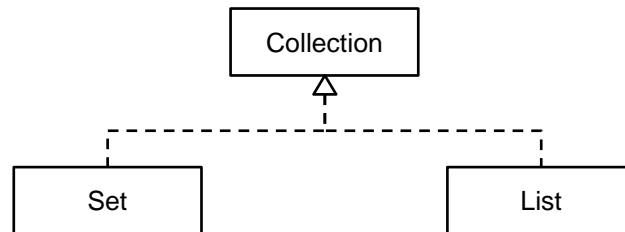
Etwas Vorsicht ist geboten, wenn das erste Objekt gelöscht wird. Dann muss nämlich auch der *first*-Zeiger angepasst werden. Die Laufzeitkomplexität beträgt auch hier aufgrund einer linearen Suche  $O(n)$ .



**Passende Aufgaben:** Programmieren Aufgabe 1.A-1.C.

## 2.4. Listen im Java Collection Framework

Auch mit verlinkten Listen werden die Daten in einer Sequenz gespeichert. Man kann also, wie bei Arrays, die Elemente von 0 bis  $n-1$  durchnummerieren und Zugriffe anhand von Indizes ermöglichen. Im Java Collection Framework implementieren solche Strukturen das Interface *List*. Dieses ist eine Spezialisierung des Interface *Collection*, jedoch nicht von *Set*:

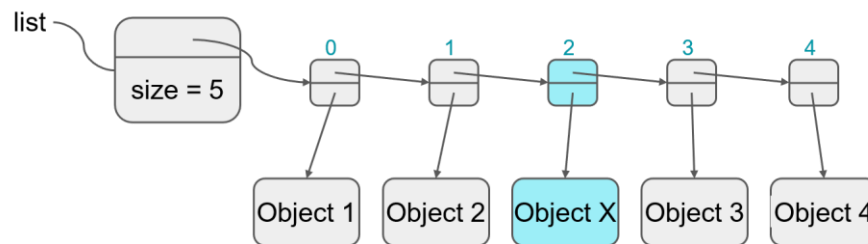


Dies bedeutet, dass Listen alle Methoden des Collection Interface mit Bag-Semantik implementieren. Zu den Methoden, die *List<E>* zusätzlich zu *Collection<E>* definiert, gehören u.a.:

- `void add(int index, E element)`,
- `E get(int index)` und
- `E remove(int index)`.

Die genauen Anforderungen an diese Methoden finden sich in der Java-Dokumentation<sup>1</sup>. Darüber hinaus wird die Spezifikation von `boolean add(E e)` konkreter gefasst: Das neue Element ist *am Ende der Liste* anzufügen.

Beispiel: `add(2, e)`



**Passende Aufgaben:** Programmieren Aufgaben 1.D-1.E.

Das Java Collection Framework enthält ausser den bisher erwähnten Interfaces auch verschiedene Implementierungen. Für *List* sind dies vor allem die beiden bisher diskutierten Lösungsansätze:

- *ArrayList* ist eine Array-basierte Implementierung mit unsortierter Speicherung und Bag-Semantik. Die Länge ist nicht begrenzt. Wird mehr Platz benötigt, so wird das Array vergrößert und alle Elemente werden in das neue Array kopiert. Zugriff auf einzelne Elemente mittels Index ist mit Laufzeitaufwand  $O(1)$  schnell. Der Aufwand von Operationen, bei denen viele Elemente verschoben werden müssen, wie bei *remove* oder *indexiertem add*, ist  $O(n)$ .  
Hinweis: Wird der Default-Konstruktor verwendet, wird die *ArrayList* mit einer initialen Kapazität von 10 erzeugt. Es empfiehlt sich daher für Anwendungen, die deutlich mehr Elemente speichern wollen, gleich den Konstruktor mit dem Kapazitätsparameter zu verwenden und einen grösseren Wert zu übergeben.
- *LinkedList* ist die Implementierung einer dynamischen verlinkten Liste. Der Nachteil ist, dass der Zugriff auf ein beliebiges Element via Index in  $O(n)$  liegt.

<sup>1</sup> Java List API, Verfügbar unter <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/List.html>

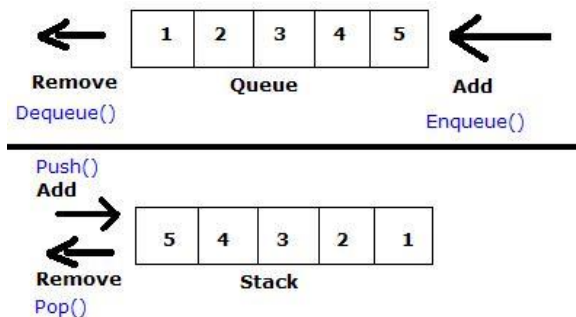
## 2.5. Datenstrukturen Stack und Queue

Stacks und Queues sind spezielle Datenstrukturen, deren Implementierungen auf Listen basieren. Der Stack arbeitet nach dem LIFO-Prinzip (**L**ast-**I**n-**F**irst-**O**ut) und die Queue nach dem FIFO-Prinzip (**F**irst-**I**n-**F**irst-**O**ut). Eine Queue liefert die Elemente also in genau der Reihenfolge aus, in der sie eingefügt wurden. Ein Stack kehrt die gerade Reihenfolge um: Das zuletzt eingefügte Element kommt als erstes zurück.

Auf einem Stack können Elemente durch Aufruf der Methode *push()* abgelegt werden. Zugriff hat man jeweils nur auf das „oberste Element“. Die Methode *top()* gibt eine Referenz auf dieses Element zurück, ohne es vom Stack zu entfernen. Die Methode *pop()* liefert ebenfalls das oberste Element des Stacks zurück und entfernt es aus der Datenstruktur.

In Java gibt es die Klasse *Stack*, die unter anderem das *List*-Interface implementiert, womit natürlich dann auch Zugriffe auf andere Positionen im Stack möglich sind.

Für die Queue existiert in Java ein Interface. Dieses Interface definiert für das Hinzufügen, Entfernen sowie den Zugriff jeweils zwei verschiedene Methoden. Die eine Methodengruppe wirft eine Exception, wenn die jeweilige Operation nicht ausgeführt werden kann, die andere arbeitet mit Rückgabewerten.



*Inhalt der beiden Datenstrukturen nach dem Einfügen der Zahlen 1-5<sup>2</sup>*

**Passende Aufgaben:** Arbeitsblatt Aufgabe 1, Programmieren Aufgabe 2.

<sup>2</sup> Grafik kopiert von: <http://www.c-sharpcorner.com/>, 12.2.2014