# Übungsblatt 3

Abgabe: 28.05.2023

---

Auf diesem Übungsblatt macht die Person die Implementierung, die in einem Telefonbuch zuerst gelistet würde. Die andere Person macht die Tests.

## Aufgabe 1 Ordnung ist das halbe Leben ($4 \times 25\,\%$)

### Aufgabe 1.1 Implementierung

Im Archiv dieses Übungsblatts gibt es die abstrakte Klasse *Set<E>*, die die Basis für vier verschiedene Implementierungen ist, um Mengen als einfach verkettete Listen zu repräsentieren. Hierzu gibt es bereits die Klasse *Node<E>*, die einen Knoten in der Listenstruktur darstellt. Da das Testen, ob ein Element enthalten ist, die zentrale Operation von Mengen ist, müsst ihr hierfür verschiedene Varianten so genannter *selbstanordnender Listen* umsetzen. Die Idee dabei ist, dass nach dem erfolgreichen Finden eines Elements die Liste so umgeordnet wird, dass dieses Element das nächste Mal schneller gefunden wird. Dem liegt die Annahme zugrunde, dass nach manchen Elementen häufiger gesucht wird als nach anderen.

Folgende Strategien für die Selbstanordnung der Liste müssen als eigene Klassen implementiert werden, die alle direkt oder indirekt von *Set<E>* erben:

**Naiv.** Die Reihenfolge der Elemente der Liste wird nicht verändert. Neue Elemente werden an den Anfang eingefügt. Hierfür wurde eine Implementierung bereits begonnen, die ihr in der Klasse *SetNaive<E>* findet.

**MF-Regel (*Move-to-front*).** Wie *Naiv*, aber jeder gefundene Eintrag wird entnommen und an den Anfang der Liste gestellt. Es bietet sich an, diese Implementierung von der naiven erben zu lassen, um die Einfügemethode nicht noch einmal implementieren zu müssen.

**T-Regel (*Transpose*).** Wie *Naiv*, aber jeder gefundene Eintrag wird mit seinem Vorgänger vertauscht, also eine Stelle näher zum Listenanfang verschoben.[1] Auch hierfür bietet sich ein Erben von der naiven Methode an.

**FC-Regel (*Frequency Count*).** Das Einfügen bzw. Wiedereinfügen passiert entsprechend der Häufigkeit der Suchanfragen, d.h. häufig gesuchte Elemente stehen dichter am Anfang. Ein dafür geeigneter Zähler ist in der Klasse *Node<E>* bereits vorhanden.

In der Klasse *Set<E>* sind lediglich zwei Methoden abstrakt:

**contains:** Die öffentliche Methode muss testen, ob ein Element in der Menge enthalten ist. Wenn es gefunden wird, wird es möglicherweise in der Liste verschoben. Bitte beachtet, dass die Gleichheit von Elementen mit *equals* bestimmt werden muss.

**addToList:** Fügt ein neues Element in die Liste ein. Die öffentliche Methode *add* ruft *addToList* immer erst nach einem erfolglosen Aufruf von *contains* auf, d.h. wenn geprüft wurde, dass sich das neue Element nicht bereits in der Liste befindet.

[1] Hierbei müssen die Listenelemente vertauscht werden, nicht der Inhalt der Elemente!

Die Klasse *Set<E>* speichert eine Referenz auf den Kopf der einfach verketteten Liste, die mit *getHead* gelesen und mit *setHead* verändert werden kann. Zudem wird die Schnittstelle *Iterable<E>* implementiert, d.h. es gibt eine Methode *iterator*. Letztere darf aber nicht für die Implementierung in abgeleiteten Klassen genutzt werden.

### Aufgabe 1.2 Tests

Für das Testen soll eine zur Implementierung analoge Ableitungshierarchie erzeugt werden. Diese wurde im bereit gestellten Projekt bereits begonnen. Die abstrakte Klasse *SetTest* testet bereits alle bereits in *Set<E>* vorhandenen Methoden, soweit sie unabhängig von der Listenanordnung sind. *SetNaiveTest* erbt von *SetTest* und soll Tests ergänzen, die spezifisch für die naive Variante sind. Analog ist für die anderen Implementierungen zu verfahren. *SetTest* enthält eine abstrakte Methode, die in allen abgeleiteten Klassen implementiert werden muss:

**emptySet:** Diese Methode muss eine neue, leere Menge des Typs konstruieren, der gerade getestet werden soll. Zum Testen werden immer Elemente des Typs *Integer* verwendet. Die Klasse *SetNaiveTest* enthält bereits ein Implementierung von *emptySet*.

Da die Klasse *SetTest* bereits *contains* testet, könnt ihr euch auf die Methoden *add* und *iterator* beschränken. *addToList* sollte nicht direkt getestet werden, da es die Menge in einen ungültigen Zustand versetzen kann. *iterator* bietet die Möglichkeit, die Reihenfolge der gespeicherten Elemente zu erfahren, denn der Iterator durchläuft die Liste von vorne nach hinten. Damit kann also geprüft werden, ob die erwarteten Umordnungen stattfinden.