SE1, Aufgabenblatt 13

Softwareentwicklung I – Wintersemester 2016/17

Mengen mit Hashing implementieren; Sortieren und Stack

Moodle-URL: uhh.de/se1

Feedback zum Übungsblatt: uhh.de/se1-feedback

Projektraum ..... Softwareentwicklung 1- WiSe 2016/17

Ausgabewoche ..... 26. Januar 2017

Kernbegriffe

Eine Menge unterscheidet sich von anderen Sammlungstypen primär dadurch, dass sie keine Duplikate zulässt. Im Umgang mit einer Menge ist die zentrale Operation die Nachfrage, ob ein gegebenes Element bereits enthalten ist (*istEnthalten*). Bei einer Liste ist diese Operation nicht effizient realisierbar, da das gesuchte Element an jeder Position stehen kann; eine Listen-Implementation muss somit im Schnitt die Hälfte aller Elemente überprüfen, bis das gewünschte Element gefunden ist (vorausgesetzt, es ist in der Liste enthalten), der Aufwand zur Laufzeit ist also proportional zur Länge der Liste (formaler: die Komplexität von *istEnthalten* auf einer Liste ist *O(n)*).

Um festzustellen, ob ein Element in einer Sammlung enthalten ist, muss es ein Konzept von Gleichheit geben. Für Java wird hierfür die Operation `equals` verwendet, die jeder Objekttyp anbietet. Wenn ein Objekttyp keine eigene Definition von Gleichheit implementiert, erfolgt automatisch eine Überprüfung auf Referenzgleichheit. Für Java gilt außerdem: Sind zwei Objekte laut `equals` gleich, dann müssen beide als Ergebnis der Operation `hashCode` den gleichen Wert liefern, d.h. wenn `a.equals(b)`, dann `a.hashCode() == b.hashCode()`.

Die effizientesten Implementationen von Operationen wie *istEnthalten* basieren auf so genannten *Hash-Verfahren*. Die Elemente werden dabei in einem Array von *Überlaufbehältern* gespeichert. Dieses Array bezeichnet man auch als *Hash-Tabelle*. Jedes Element kann nur in einem dieser Behälter vorkommen. Eine *Hash-Funktion* bildet ein Element auf einen ganzzahligen Wert ab. Dieser Wert wird geeignet auf einen Index in der Hash-Tabelle abgebildet, so dass beim Einfügen, Löschen und Aufsuchen eines Elements der richtige Behälter verwendet wird.

Eine gute Hash-Funktion sollte die Elemente möglichst gleichmäßig über die Hash-Tabelle „verschmieren" – Im Idealfall enthält jeder Überlaufbehälter maximal ein Element. Bei einer solchen Auslastung ist der Aufwand für das Auffinden eines Elements nicht mehr von der Kardinalität der Menge abhängig, sondern setzt sich konstant aus der Indexberechnung und dem indexbasierten Zugriff auf einen Überlaufbehälter zusammen.

Interfaces können als Spezifikationen angesehen werden, die das Verhalten einer Implementation stark festlegen (siehe `Set` und `List`). Pragmatisch werden sie häufig nur zur syntaktischen Festlegung einer Schnittstelle benutzt, die einen Dienst mit relativ großen Freiheitsgraden bieten kann.

Sortieren ist eine klassische Problemstellung in der Informatik. Es existieren verschiedene *Sortieralgorithmen* (*Sortierverfahren*) mit unterschiedlicher Laufzeitkomplexität. Sehr häufig müssen Elemente an ihrem aktuellen Speicherort (*in situ* oder *in place*) in eine geordnete Reihenfolge gebracht werden. *Bubble-Sort* ist ein sehr einfaches Sortierverfahren für diese Situation mit quadratischem Aufwand ($O(n^2)$). *Quick-Sort* hingegen gilt als ein sehr schneller Sortieralgorithmus, der im günstigsten Fall eine Komplexität von *O(n log(n))* aufweist, im ungünstigen Fall aber zu $O(n^2)$ degenerieren kann.

Ein *Stack* (im Deutschen auch Stapel oder Kellerspeicher genannt) ist eine Sammlung von Elementen, die nach dem LIFO (Last In First Out) Prinzip verwaltet werden, d.h. zuletzt abgelegte Elemente werden zuerst wieder entnommen (wie etwa bei einem Tablettstapel). Ein Stack definiert im wesentlichen vier Operationen: *push* legt ein Element auf den Stack, *pop* entfernt ein Element vom Stack, mit *peek* kann das oberste Element abgefragt werden, ohne es zu entfernen, und *isEmpty* prüft, ob der Stack leer ist.

Lernziele

Das Prinzip von Hash-Verfahren verstehen, noch sicherer mit Arrays umgehen, weitere Anwendungen von Interfaces kennen. Sortierverfahren, die mit Vertauschungen arbeiten, verstehen und auf Basis ihrer abstrakten Beschreibung implementieren können, noch sicherer mit Listen umgehen können, Stacks sowohl implementieren als auch anwenden können.

Aufgabe 13.1 Wortschatz und Hash-Tabelle

13.1.1 Im vorgegebenen BlueJ-Projekt *Hashing* befindet sich unter anderem das Interface `Wortschatz`, das den möglichen Umgang mit einer Menge von Wörtern beschreibt. Schaut es euch in der Dokumentationsansicht gut an, denn in der nächsten Aufgabe sollt ihr es mit einer Klasse `HashWortschatz` implementieren. Es ist hilfreich, sich zuerst Gedanken über mögliche Testfälle zu machen und diese zu programmieren; dies steigert das Verständnis der Funktionalität eines Wortschatzes und erleichtert das Implementieren. In dieser Aufgabe ist ein JUnit-Testgerüst vorgegeben. Kommentiert die Testmethoden und füllt die Rümpfe.

Falls euch noch weitere Testfälle einfallen, könnt ihr diese natürlich gerne ergänzen.

13.1.2 **Skizziert, wie eine Hash-Tabelle aufgebaut ist** und erläutert eurem Betreuer bzw. eurer Betreuerin anhand der Skizze, wie diese funktioniert und welche Vorteile diese Lösung gegenüber einer Listenimplementation (ohne Hash-Verfahren) hat.

Aufgabe 13.2 HashWortschatz

13.2.1 Vervollständigt nun die Klasse `HashWortschatz`, indem ihr ein Hash-Verfahren implementiert. Für die Hashwertberechnung steht (über einen Konstruktor-Parameter) eine Implementation des Interfaces `HashWertBerechner` bereit. Die Angabe eines Berechners erlaubt die Verwendung verschiedener Hash-Funktionen. Denkt daran, dass der von der Hash-Funktion gelieferte Wert noch auf die Größe der Tabelle angepasst werden muss, um einen gültigen Index in die Tabelle zu erhalten. Das ist in mehreren Methoden nötig, beispielsweise in `fuegeWortHinzu`. Falls ihr ein und denselben Quelltext in mehreren Methoden benötigt, solltet ihr nicht copy/paste verwenden, sondern die Logik in eine Hilfsmethode auslagern.

Verwendet als Überlaufbehälter Exemplare der mitgelieferten Klasse `WortListe`. Eure Hash-Tabelle soll also ein Array von `WortListen` sein. Die Größe dieser Hash-Tabelle bekommt eure Implementation über den Konstruktor von `HashWortschatz` als zweiten Parameter übergeben.

Tipp: Überlegt euch zu Beginn, welche Zustandsfelder die Klasse `HashWortschatz` benötigt, und beginnt mit der Implementierung des Konstruktors. Beachtet die Kommentare im Interface `Wortschatz`, wenn ihr danach die einzelnen Methoden implementiert.

13.2.2. Implementiert in der Klasse `HashWortschatz` die Operation `schreibeAufKonsole` so, dass sie den Inhalt aller Überlaufbehälter auf der Konsole ausgibt. In der Darstellung soll pro Überlaufbehälter eine Zeile verwendet werden, etwa folgendermaßen:

```
[0]: Hund Katze
[1]:
[2]: Maus
```

Tipp: Über Exemplare der `WortListe` könnt ihr mit der erweiterten for-Schleife iterieren, genauso, wie ihr es bei den Sammlungen der Java-Bibliothek gemacht habt. Dies ist möglich, weil die Klasse `WortListe` das Interface `Iterable<String>` implementiert.

Ruft nun (über einen Rechtsklick auf die Klasse `Startup`) die Methode `visualisiereHashtabelle` auf. Diese erzeugt ein Exemplar eurer `Wortschatz`-Implementation, fügt einige Wörter aus einer kurzen Textdatei in den Wortschatz ein und ruft anschließend `schreibeAufKonsole` auf.

Aufgabe 13.3 Sortieren einfach

13.3.1 Öffnet das Projekt *Sortieren* und erstellt ein Exemplar der Klasse `VisualIntListe`. Daraufhin öffnet sich ein neues Fenster mit weißen Quadraten. Was bedeuten diese Quadrate? Lest euch die Dokumentation der Klasse durch. Ruft anschließend an dem Exemplar die Operation `initialisiereAufsteigend` auf. Welche Änderungen ergeben sich dadurch, und was bedeuten diese?

13.3.2 Studiert die Schnittstelle `Sortierer` und implementiert anschließend den Bubble-Sort-Algorithmus in der vorgegebenen Klasse `BubbleSortierer`.

13.3.3 Um eure Implementation von `BubbleSortierer` interaktiv zu testen, erzeugt ihr ein Exemplar der Klasse `VisualIntListe` und ein Exemplar der Klasse `BubbleSortierer`. Dann ruft ihr an dem Sortierer die Methode `sortiere` auf und übergebt die zu sortierende Liste als Parameter.

Aufgabe 13.4 Sortieren besser

13.4.1 Implementiert den Quick-Sort-Algorithmus (**in place**). Vervollständigt dazu die Klasse `QuickSortierer`, welche ebenfalls das Interface `Sortierer` implementiert. Testet wieder interaktiv. Die Teillisten werden im Quelltext durch die beiden Grenzvariablen `von` und `bis` realisiert. Die öffentliche `sortiere`-Methode sorgt dafür, dass der rekursive Prozess mit passenden Werten für die komplette Liste angestoßen wird. Beachtet, dass die Länge der Teilliste von den Grenzvariablen `von` und `bis` abhängt. Es wäre sinnlos, die Operation `gibLaenge` auf der Liste aufzurufen, da diese immer die Länge der kompletten Liste liefert.

13.4.2 *Zusatzaufgabe*: Experimentiere mit verschiedenen Strategien zur Wahl des Pivot-Elements, zum Beispiel erstes Element, mittleres Element, letztes Element, der Median aus diesen drei Elementen, ein zufälliges Element, der Median aus drei zufälligen Elementen, der Median aus 9 zufälligen Elementen…