SE1, Aufgabenblatt 13

Softwareentwicklung I – Wintersemester 2015/16

Mengen mit Hashing implementieren

MIN-CommSy-URL: ..... https://www.mincommsy.uni-hamburg.de/
CommSy-Projektraum ..... SE1 CommSy WiSe 15/16
Ausgabewoche ..... 21. Januar 2016

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.

Lernziele

Das Prinzip von Hash-Verfahren verstehen, noch sicherer mit Arrays umgehen, weitere Anwendungen von Interfaces kennen.

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 Performancevergleich von Hash-Funktionen

13.3.1 Welche Komplexität hat die Operation `enthaeltWort(String)` im günstigsten Fall? Welche im ungünstigsten Fall? Wovon hängt das ab?

13.3.2 Erstellt eine weitere Klasse, die das Interface `HashWertBerechner` derart realisiert, dass `hashWert` einen konstanten Wert (unabhängig vom übergebenen Wort) zurückgibt. Was bewirkt das?

13.3.3 Ergänzt die Methode `vergleichePerformance` der Klasse `Startup` so, dass sie beide Realisierungen des Interfaces `HashWertBerechner` testet. Um einen messbaren Unterschied feststellen zu können, wird ein sehr großer Wortschatz verwendet.

13.3.4 Erstellt nun mindestens zwei weitere Implementationen von `HashWertBerechner`, welche eigene, sinnvolle Hash-Funktionen bereitstellen. Dazu müsst ihr lediglich „irgendwie“ die Buchstaben des eingegebenen Wortes verwenden, um daraus einen Integerwert zu berechnen.
Bezieht diese Lösungen in eure Messungen mit ein!

Aufgabe 13.4 Hash-Tabelle dynamisch vergrößern

13.4.1 Bisher ist die Größe der Hash-Tabelle statisch festgelegt. Spätestens wenn mehr Elemente eingefügt werden, als Überlaufbehälter vorhanden sind, enthalten einzelne Überlaufbehälter mehr als ein Element. Dies reduziert die Effizienz. Ab einem gewissen *Befüllungsgrad* (Element-Anzahl / Array-Länge) ist es sinnvoll, die Hash-Tabelle dynamisch zu vergrößern. Implementiert dies und vergesst dabei das Rehashen nicht. Der kritische Befüllungsgrad soll im Konstruktor als ganzzahlige Prozentzahl angegeben werden können.