TU München, Fakultät für Informatik
Lehrstuhl III: Datenbanksysteme
Prof. Alfons Kemper, Ph.D.

Übung zur Vorlesung *Einführung in die Informatik 2 für Ingenieure (MSE)*

Alexander van Renen (renen@in.tum.de)
http://db.in.tum.de/teaching/ss16/ei2/

Blatt Nr. 6
Dieses Blatt wird am Montag, den 30. Mai 2016 besprochen.

Teil 1: Objektorientierte Modellierung (in UML) und Programmierung (in Java)

Der erste Teil der Vorlesung zu Java und UML ist mit diesem Übungsblatt abgeschlossen. Jetzt ist also *die* Gelegenheit, den Stoff zu wiederholen. Dies erspart Ihnen nicht nur Lernaufwand am Ende des Semesters, sondern Sie haben außerdem noch die Möglichkeit in dieser letzten Übungsstunde zu Java offene Fragen zu klären. Zur Inspiration diese (unvollständige) Liste:

- UML
  - Klassendiagramme, Objektdiagramme
  - Assoziationen (Multiplizitäten, Aggregation vs. Komposition, Navigation)
  - Operationen (Konstruktoren, Beobachterfunktionen, Mutatoren, Access Modifier)
  - Vererbung (Generalisierung, Spezialisierung)
- Java
  - Umsetzung von UML in Java
  - Gemeinsame Unterobjekte
  - Typisierung
  - Werte vs. Objekte
  - Klassenattribute vs. Objektattribute
  - Methoden-Überladung
  - Vererbung und dynamisches Binden
  - Rekursion
  - Java Collections Framework (Set, List, Queue, Map)
  - Generische Typen
  - Iteration (for each loop, Iteratoren)
- Datenstrukturen
  - Komplexitätsangaben für Operationen ($\mathcal{O}(1)$, $\mathcal{O}(\log n)$, $\mathcal{O}(n)$)
  - Menge (Set), Verkettete Liste (LinkedList), Keller (Stack)
  - Binäre Suchbäume (Degenerierung, AVL-Baum)
  - Hashing (Double Hashing, Hashing with Chaining, Hashing with Linear Probing)

Aufgabe 1: Hashing with Chaining

In dieser Aufgabe wollen wir unsere eigene generische Hashtabelle implementieren, mit der man Werte unter Schlüsseln ablegen kann. In der Vorlesung haben Sie Double Hashing, Hashing with Chaining und Hashing with Linear Probing kennen gelernt. Wir werden in dieser Aufgabe *Hashing with Chaining* für unsere Hashtabelle verwenden.

Abbildung 1 zeigt ein Beispiel. Unsere Hashtabelle soll eine ArrayList[1] von verketteten Listen für die Kollisionsbehandlung verwenden. Außerdem soll sie wie das Vorbild `HashMap` die Methode `put(K key, V value)` zum Ablegen eines Wertes vom generischen Typ `V` unter einem Schlüssel vom generischen Typ `K` anbieten (im Beispiel ist `K=String` und `V=Integer`). Als Hashfunktion können Sie wie in der Abbildung `hashCode()` verwenden, diese Methode ist praktischerweise für jedes Objekt in Java definiert. Der `hashCode` modulo der Größe der ArrayList gibt den Index in die ArrayList und damit die verkettete Liste, in die das neue Element eingefügt wird. Für die verkettete Liste können Sie `LinkedList` aus Java Collections verwenden. Die Hashtabelle soll die Methode `get(K key)` zum Abfragen des Wertes zu einem Schlüssel unterstützen.

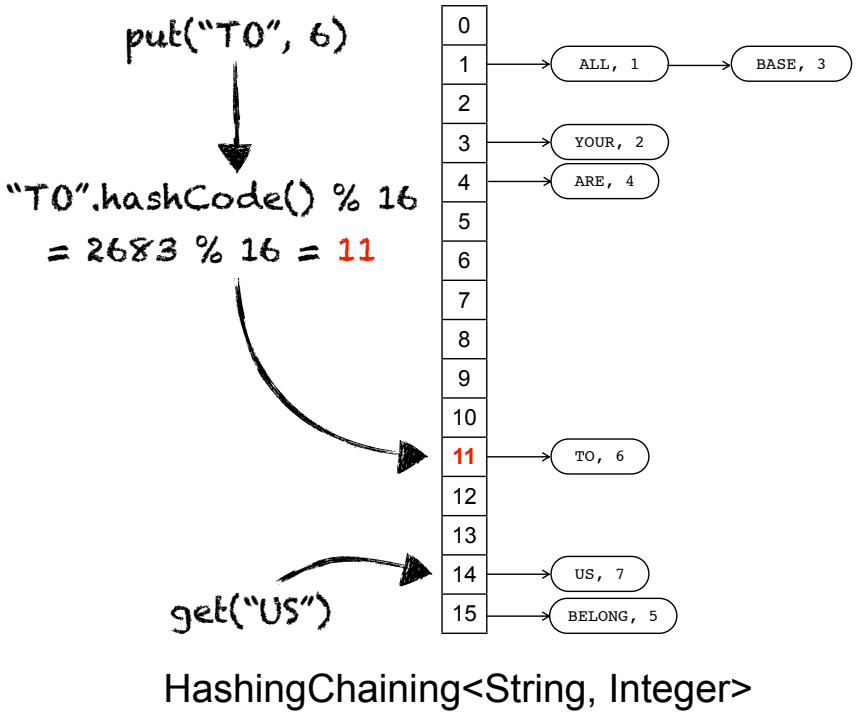

Abbildung 1: Hashing with Chaining verwendet verkettete Listen um Kollisionen zu behandeln

Aufgabe 2: Komplexitätsangaben

Sie haben in der Vorlesung und Übung Komplexitätsangaben in der Landau-Notation (z.B. $\mathcal{O}(n)$) kennen gelernt. Diese geben das asymptotische Laufzeitverhalten von Funktionen an. In dieser Aufgabe wollen wir feststellen, was dies in der Praxis bedeutet. Dafür messen wir die Laufzeit für das Nachschlagen in `HashMap` und `TreeMap`. Welche Laufzeitkomplexität erwarten Sie jeweils in Abhängigkeit von der Eingabegröße und können Sie diese mit Ihren Messergebnissen nachweisen?

[1] Man könnte hier auch ein Array verwenden, aber dann bekommt man Probleme á la *generic array creation*. Generell sollte man Generics und Arrays möglichst nicht kombinieren und stattdessen eine `ArrayList` verwenden.

Tipps zur Messung

Fügen Sie zuerst die Elemente mit `put()` in die Abbildung ein und führen Sie dann 10 Millionen Lookups mit `get()` durch (so viele damit es messbar wird). Die Laufzeit können Sie messen, in dem Sie vor und nach den 10 Millionen Lookups mit `System.currentTimeMillis()` die aktuelle Zeit in Millisekunden seit 1970 abfragen. Die Differenz der beiden Zahlen ergibt entsprechend die Laufzeit in Millisekunden. Anschließend können Sie sich das Ergebnis z.B. in einem Punktediagramm in einem Tabellenprogramm Ihrer Wahl[2] veranschaulichen.

Wenn Sie überprüfen wollen, ob eine Laufzeit logarithmisch ist, sollten Sie die Messungen mit exponentiell ansteigender Eingabegröße durchführen (z.B. mit 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, ..., 1048576 Elementen). Wenn Sie die x-Achse Ihres Diagramms logarithmisch skalieren, wäre diese Laufzeit dann als Gerade erkennbar.

Aufgabe 3: AVL-Bäume

Fügen Sie in einen AVL-Baum nacheinander die folgenden Elemente ein und führen Sie dabei die notwendigen Rotationen durch: 4, 8, 16, 12, 14, 3, 2, 6, 5

[2] Microsoft Excel, Google Docs, Apple Numbers, Open Office Calc, ...