Lucerne University of
Applied Sciences and Arts

HOCHSCHULE
LUZERN

Informatik

Algorithmen & Datenstrukturen (AD)

Übung: Hashbasierte Datenstrukturen, Performance, Thirdparty-Datenstrukturen (D3)

Themen: Hashbasierende Datenstrukturen, Performanceaspekte, Java Standard Datenstrukturen und Thirdparty-Quellen.

Zeitbedarf: ca. 240 Minuten

Wichtig: Ziel dieser Aufgaben ist es, ausgewählte Datenstrukturen mindestens teilweise selber zu implementieren, um deren Funktion und Aufbau besser zu verstehen sowie das algorithmische Denken und auch das Programmieren zu üben.
In der Praxis vermeidet man eigene Implementationen und verwendet stattdessen möglichst die bereits vorhandenen, erprobten und optimierten Implementationen.

Roland Gisler, Version 1.3.1 (FS 2021)

1 Einfache Hashtabelle (bzw. Hashset) (ca. 60')

1.1 Lernziele

- Verstehen wie eine Hashtabelle grundsätzlich funktioniert.
- Eine Hashtabelle mit statischer Grösse selbst implementieren.

1.2 Grundlagen

Diese Aufgabe basiert auf dem Input **D31**.

1.3 Aufgaben

a.) Wir wollen mit Hilfe eines Arrays eine (statische) Datenstruktur implementieren, welche die Hashwerte der eingefügten Objekte verwendet, um den Zugriff zu beschleunigen. Zur Vereinfachung gehen wir davon aus, dass es keine doppelten Einträge geben darf (Set-Semantik). Welchen Datentyp nutzen wir für die Hashwerte?

b.) Entwerfen Sie als erstes die Schnittstelle: Wir wollen als minimale Funktionalität einzelne Elemente einfügen und wieder entnehmen können. Eine Grundlage dafür ist natürlich, dass wir Elemente auch suchen können.

c.) Für unsere Versuche möchten wir Elemente einfügen, welche möglichst einfach nachvollziehbare Hashwerte verwenden. Sie können dafür z.B. die Klasse `Integer` verwenden (wenn, dann prüfen Sie deren Verhalten!). Oder Sie modifizieren ggf. die bereits bekannte Klasse `Allocation` (aus der Übung **E0**) so, dass `equals()` (und somit auch `hashCode()`) nur auf der Startadresse basieren.

d.) Beginnen Sie mit der Implementation. Halten Sie das Ganze bewusst sehr übersichtlich und beschränken Sie sich z.B. auf eine Grösse von nur `10` Elementen. Wie berechnen Sie aus dem Hashwert der eingefügten Elemente den Array-Index?

Tipp: Überschreiben Sie auf Ihrer Hashtabelle die `toString()`-Methode, damit Sie den Inhalt des Arrays sehr einfach ausgeben und kontrollieren können.

e.) Testen Sie Ihre Implementation. Überlegen Sie sich dafür mindestens drei sinnvolle Szenarien! Wo liegen die Grenzen Ihrer Implementation?

2 Performance-Vergleich: Stack-Implementationen (ca. 90')

2.1 Lernziele

- Vergleich der eigenen Stack-Implementation (von letzter Woche) mit der Library-Klasse.
- Erfahrungen zu sinnvollen und fairen Messungen sammeln.

2.2 Grundlagen

Diese Aufgabe basiert auf dem Input **D32**. Ausserdem benötigen Sie Ihre Stack-Implementation aus den Übungen von **D2**.

2.3 Aufgaben

a.) Für eine einigermassen faire Messung müssen wir ein paar Dinge vorbereiten. Wichtig ist, dass wir für alle Messungen die identischen, bereits zuvor erzeugten Datenelemente in der gleichen Reihenfolge verwenden. Implementieren Sie dazu eine Klasse mit einer (statischen) Methode, welchen Ihnen einen Array von z.B. 100'000 Objekten (ideal: per Parameter konfigurierbar) zurückliefert.

b.) Wenn Sie als Typ eine eigene Klasse (z.B. `Allocation`) verwenden, achten Sie darauf, dass darin keine unnötigen Aktionen (`System.out.println(…)` oder Logging etc.) stattfinden. Erzeugen Sie kleine Objekte (z.B. bei Strings mit nur wenigen Zeichen) – die Objektgrösse hat auf die Laufzeit der Operationen keinen Einfluss, sehr wohl aber auf den Speicherverbrauch!

c.) Messen Sie die Zeit, welche Sie für die Datenerstellung benötigen. Aus der Aufgabe **2** der Übungen E1 kennen Sie bereits die Methode `System.currentTimeMillis()`, welche Ihnen einen Zeitstempel in Millisekunden liefert.

Achtung: «*Wer misst, misst Mist!*» war eine sehr prägende Aussage eines ehemaligen Dozenten des Autors dieser Aufgabe. Werden Sie sich bewusst, dass Laufzeitmessungen von sehr vielen (Stör-)Faktoren verfälscht werden können. Sie sollten darum jede Messung mehrfach durchführen, und durch Veränderung der Rahmenbedingungen zusätzlich überprüfen, ob diese auch eine Veränderung in die erwartete Richtung provozieren.

d.) Verwenden Sie die Klasse `java.util.Stack` aus dem Java Collections Framework, welche im Gegensatz zu den aktuelleren Collections **synchronisiert** implementiert ist. Messen Sie die Zeit, welche benötigt wird, um die (vorbereiteten) Objekte möglichst schnell (mit einem `for`-Loop) in die Datenstruktur einzufügen.
Achtung: Damit es fair bleibt, erzeugen Sie den Stack mit einer initialen Grösse!

e.) Nun treten Sie in Konkurrenz mit ihrer eigenen Implementation! Wiederholen Sie die Messung mit identischen Rahmenbedingungen aber mit Ihrem eigenen Stack!
Und? Haben Sie die Java-Implementation geschlagen? Sind die Resultate plausibel? Erhöhen Sie ggf. die Datenmenge (auf eine Million bis 10 Millionen Objekte).

f.) Nun versuchen Sie es noch mit einer Implementationen von `java.util.Deque`. Überlegen Sie gut welche Implementation und welche Methoden Sie verwenden, konsultieren Sie dazu die JavaDoc. Wer ist der Gewinner?

3 Hashtabelle mit Kollisionen (’90)

3.1 Lernziele

- Umgang mit Kollisionen auf Hashtabellen.
- Lineares Sondieren und Tombstones verstehen.
- Implementation einer Hashtabelle verbessern.

3.2 Grundlagen

Diese Aufgabe basiert auf dem Input **D31** und der Aufgabe **1**.

3.3 Aufgaben

a.) Erweitern Sie die Lösung von Aufgabe **1** mit einem Füllstand der Datenstruktur, welchen Sie z.B. mit einer Methode **`size()`** und/oder **`isFull()`** abfragen können. Prüfen Sie nochmal kritisch nach, ob Ihre Implementation zuverlässig verhindert, dass zweimal das gleiche Objekt eingefügt werden kann. Haben Sie dafür überhaupt jemals **`equals()`** aufgerufen?

b.) Nun möchten wir zwei verschiedene Objekte einfügen, die entweder denselben Hashwert haben, oder aber durch die sehr kleine Grösse der Datenstruktur (nur zehn Elemente) auf denselben Index positioniert würden. Entwerfen Sie ein entsprechendes, von Ihren eingefügten Elementen abhängiges Szenario.

c.) Verbessern Sie Ihre Implementation des Einfügens: Wir müssen mit Kollisionen umgehen können, und wenn nötig linear einen Platz sondieren. Wie weit müssen wir suchen? Was passiert, wenn die Datenstruktur voll ist?

d.) Haben Sie daran gedacht, dass sich damit auch das Entnehmen (und Suchen) von Elementen verändern muss? Implementieren und testen Sie auch diese Methoden gewissenhaft.

e.) Wenn alles funktioniert, testen Sie auch explizit die beiden Sonderfälle für das Einfügen und die Entnahme bei einer vollen Hashtabelle. Funktioniert es?

4 Optional: Hashtabelle mit Buckets (Listen für Kollisionen)

4.1 Lernziele

- Hashtabelle welche Kollisionen mit Buckets auflöst verstehen.
- Implementation einer Hashtabelle, kombiniert mit Listen.

4.2 Grundlagen

Diese Aufgabe basiert auf dem Input **D31** und Aufgabe **2**. Zusätzlich können Sie ggf. auf Ihre `Node`[1]-Klasse der Implementation einer einfach verknüpften Liste der Aufgabe **2** von **D2** zurückgreifen.

4.3 Aufgaben

a.) Wenn vermehrt Kollisionen auftreten, hat das lineare Sondieren einige Nachteile. Deshalb verwenden viele Hashtabellen für doppelte Elemente bzw. Kollisionen kleine, einfach verknüpfte Listen. Das macht einiges einfacher und schneller. Aber welche Nachteile kaufen wir uns damit neu wieder ein?

b.) Entwerfen Sie zuerst das Konzept für Ihre modifizierte Hashtabelle: Skizzieren Sie dazu am besten ein kleines, konkretes Szenario auf und überlegen Sie sich den neuen Ablauf der verschiedenen Operationen. Was wird durch die Listen viel einfacher (und schneller), was hingegen wird komplizierter?

c.) Implementieren Sie die Hashtabelle mit Hilfe von Bucket-Listen. Verwenden Sie dafür Listen-Nodes, welche eine einfach verkettete Liste ergeben. Überlegen Sie kurz: Warum ist die Verwendung eines zweidimensionalen Arrays nicht wirklich eine Alternative?

d.) Testen Sie Ihre Implementation möglichst mit den gleichen Szenarien wie bei Aufgabe **3** und beobachten Sie das Verhalten! Experimentieren Sie mit der Vergrösserung der Datenstruktur.

[1] Vielleicht trägt die Klasse in Ihrer Lösung einen anderen Namen, z.B. `Element` etc.

5 Optional: Verwendung einer Thirdparty-Datenstruktur

5.1 Lernziele

- Verwenden von Thirdparty-Libraries.
- Alternative Ansätze kennenlernen und ausprobieren.

5.2 Grundlagen

Diese Aufgabe basiert auf dem Input **D32** und der Aufgabe **5**.

5.3 Aufgaben

a.) Binden Sie eine der im Input erwähnten Libraries in Ihr Projekt ein. Sie dürfen selbst entscheiden, welche Implementation Sie am meisten interessiert. In der Regel finden Sie auf den entsprechenden Projektseiten die «Maven-Koordinaten», welche Sie in Ihrem `pom.xml` in das Element `dependencies` einfügen können.

b.) Studieren Sie die Beispiele und die Dokumentation der verwendete Library und verwenden Sie eine konkrete Collection. Nutzen Sie Beispielsweise die in Aufgabe **2** erstellten Testdaten für Versuche.

c.) Vergleichen Sie die Performance mit einer (fairen) Alternative des JDKs.
Gelingt es Ihnen, einen von der jeweiligen Library versprochenen Vorteil tatsächlich zu beobachten?