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 Datenstruktruen und Thirdparty-Quellen.

Zeitbedarf: ca. 240min.

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 1.0 (FS 2017)

1 Einfache Hashtabelle (bzw. Hashset) (ca. 30')

1.1 Lernziele

- Verstehen wie eine Hashtabelle grundsätzlich funktioniert.
- Eine Hashtabelle mit statischer Grösse selber 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 Elemente einfügen und wieder entnehmen können. Dazu müssen wir Elemente natürlich auch suchen können.

c.) Für unsere Versuche möchten wir Elemente einfügen, welche einfach nachvollziehbare Hashwerte verwenden. Sie können dafür die Klasse **`Integer`** verwenden ➔ prüfen Sie deren verhalten! Oder Sie modifizieren beispielsweise die bereits bekannte Klasse **`Allocation`** (aus der Übung E0) so, dass **`equals()`** (und somit auch **`hashCode()`**) nur noch auf der Startadresse basiert.

d.) Beginnen Sie mit der Implementation. Halten Sie das Ganze bewusst sehr übersichtlich und beschränken Sie sich z.B. auf eine Grösse von **`10`** Elementen. Wie berechnen Sie aus dem Hashwert der eingefügten Elemente den Array-Index?
Tipp: Geben Sie diese mit der überschriebenen **`toString()`**-Methode aus, damit Sie den Inhalt des Arrays einfach kontrollieren können.

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

2 Hashtabelle mit Kollisionen (ca. 45')

2.1 Lernziele

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

2.2 Grundlagen

Diese Aufgabe basiert auf dem Input D21 und der Aufgabe 1.

2.3 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 nach, ob Ihre Implementation 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 (was unerwünscht, aber erlaubt ist, weil wir keine perfekten Hashes haben können), 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 uns wenn nötig linear einen Platz suchen. 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 den schlechtesten möglichen Fall sowohl beim Einfügen der maximalen Anzahl als auch bei der Entnahme (!).

f.) Experimentieren Sie mit der Grösse der Datenstruktur. Wie gross muss sie sein, damit die Probleme von e scheinbar von selber verschwinden?

g.) Implementieren Sie eine Methode, welche alle enthaltenen Elemente ausgibt. Sie können dafür wieder einen Iterator implementieren, Sie können es aber auch einfacher realisieren, denn es geht nur um die Logik: Worauf muss man bei Hashtabellen achten die auf Arrays basieren?

h.) Haben Sie bemerkt, dass bei Entfernen von Elementen innerhalb der Arrays «Lücken» in den Folgen entstehen können, die durch die Sondierungen bei Kollisionen entstanden sind? Was hat das für Konsequenzen, wenn Sie danach weitere Elemente suchen oder entfernen wollen? Wie lösen Sie dieses Problem?

3 Hashtabelle mit Buckets (Listen für Kollisionen) (ca. 45')

3.1 Lernziele

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

3.2 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.

3.3 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 nun damit 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 wird komplizierter?

c.) Implementieren Sie die Liste mit Hilfe von kleinen Listen. Verwenden Sie dafür einfache, kleine 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 2 und beobachten Sie das Verhalten! Experimentieren Sie auch hier mit der Vergrösserung der Datenstruktur.

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

4 Einfache Performance-Messung und Analyse (ca. 30')

4.1 Lernziele

- Einfache Messung / Kontrolle der Performance einer Hashtabelle.
- Nachvollziehen von Abläufen und Operationen auf einer Hashtabelle.

4.2 Grundlagen

Diese Aufgabe basiert auf dem Input D31 und den Aufgabe 2 oder 3.

4.3 Aufgaben

a.) Ergänzen Sie **ausnahmsweise** auf der Klasse die Sie als Datenelement für Ihre Hashtabellen verwendet haben die `equals()`- und die `hashCode()`-Methoden mit einem Logging-Eintrag, so dass Sie sehen, wenn die Methoden aufgerufen werden.
Hinweis: Sollten Sie sich in der Aufgabe 2 bzw. 3 für die Klasse `Integer` als Datenelement entschieden haben, bauen Sie Ihre Lösung entweder auf `Allocation` um, oder implementieren Sie eine eigene kleine Datenklasse, deren `equals()`- und `hashCode()`-Implementationen Sie selber bestimmen können.

b.) Fügen Sie Elemente in Ihre Hashtabelle ein und entfernen Sie sie wieder. Beobachten Sie dabei die Log-Ausgabe. Entspricht diese Ihren Erwartungen? Beziehen Sie auch die verschiedenen, wirklich «interessanten» Szenarien (Kollisionen) mit ein. Wie sieht es dann aus?

c.) Sie können davon ausgehen, dass in der Realität die Methoden `equals()` und `hashCode()` die echten «Zeitfresser» sind! Warum ist das so? Und überlegen Sie: Welche der beiden Methoden wird in der Regel mehr Zeit benötigen?

d.) Simulieren Sie mittels `Thread.sleep(1)` eine längere Bearbeitungszeit von `equals()` und `hashCode()`. So können Sie auf einfache Art Zeitmessungen für die übergeordneten Operationen auf der Hashtabelle durchführen.

e.) Verwenden Sie nun die in den Java Collections vorhandene Implementation `HashSet` und vergleichen Sie die Aufrufe bzw. Zeiten mit Ihrer eigenen Implementation! Beachten Sie, dass Sie die Initialgrösse eines `HashSet`s mittels eine Konstruktors festlegen können. Experimentieren Sie!

5 Performance-Vergleich: Stack-Implementationen (ca. 60')

5.1 Lernziele

- Vergleich der eigenen Implementation mit der Library-Klasse.
- Erfahrungen zu sinnvollen und fairen Messungen

5.2 Grundlagen

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

5.3 Aufgaben

a.) Für eine einigermassen faire Messung müssen wir ein paar Dinge vorbereiten. Wichtig ist, dass wir für alle Messungen die identischen, bereits 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 z.B. im Konstruktor keine unnötigen Aktionen (**`System.out.println(…)`** oder Logging etc.) stattfinden. Erzeugen Sie kleine Objekte (z.B. bei Strings nur wenige Zeichen) – die Operationen werden aufgrund der Objektgrösse nicht schneller oder langsamer, wir benötigen «nur» mehr Speicher.

c.) Zwischenfrage: Warum verwenden wir für die Testdaten einen statischen Array?

d.) Messen Sie nun 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.

Wichtiger Grundsatz: «Wer misst, misst Mist!» war eine sehr prägende Aussage eines geschätzten Dozenten des Autors dieser Aufgabe. Es muss ihnen klar sein, dass Laufzeitmessungen von sehr vielen (Stör-)faktoren betroffen sein können. Sie sollten eine Messung darum immer mehrfach wiederholen, und durch Verändern der Rahmenbedingungen zusätzlich überprüfen, ob diese auch eine Veränderung in die erwartete Richtung provozieren.

e.) Verwenden Sie nun die Klasse **`java.util.Stack`** aus dem Java Collections Framework, welche im Gegensatz zu den aktuelleren Collections synchronisiert und thread-safe 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!

f.) 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).

g.) 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?

6 Optional: Verwendung einer Thirdparty-Datenstruktur

6.1 Lernziele

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

6.2 Grundlagen

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

6.3 Aufgaben

a.) Binden Sie die Library welche Sie am meisten reizt in Ihr Projekt ein. In der Regel finden Sie auf den entsprechenden Projektseiten die «Maven-Koordinaten», welche Sie in Ihrem **`pom.xml`** in die Sektion «dependencies» einfügen können.

b.) Studieren Sie die Beispiele und die Dokumentation. Versuchen Sie die zur Verfügung gestellten Collections zu benutzen und vergleichen Sie den Code mit den Standard-Collections.

c.) Nutzen Sie die Aufgabe 5 als Grundlage und wagen Sie selber mal eine Performance-Messung! Gelingt Ihnen «der Beweis» dass eine bestimmte Collection besser ist als die andere?