



Abgabe: **17.05.2016** (bis 23:59 Uhr)

Oh my GAD!
Wir wünschen erholsame Pfingstferien :-)

Aufgabe 4.1 (P) Bankkonto-Methode

Diese Tutoraufgabe ist ein Kurztutorial für die Bankkonto-Methode, die in der Vorlesung für dynamische Arrays verwendet wurde. Die zweite Tutoraufgabe veranschaulicht die Theorie an verallgemeinerten dynamischen Arrays.

Sei S eine Menge von Operationen, und bezeichne $T(\sigma)$ (eine obere Schranke für) die Laufzeit einer Operation $\sigma \in S$ (diese Laufzeit kann vom aktuellen Zustand des Objekts, auf dem die Operation wirkt, sowie von Argumenten abhängen). Analog dazu bezeichne $T(\sigma_1, \sigma_2, \dots, \sigma_m) := \sum_{i=1}^m T(\sigma_i)$ (die korrespondierende obere Schranke für) die Laufzeit der Operationsfolge $(\sigma_1, \sigma_2, \dots, \sigma_m)$.

Ziel einer amortisierten Analyse ist, eine möglichst gute obere Schranke für $T(\sigma_1, \sigma_2, \dots, \sigma_m)$ zu finden. Bei der Konto-Methode bestimmen wir dazu eine Funktion $\Delta : S \rightarrow \mathbb{R}$, die folgende Eigenschaften besitzt:

- (i) Für alle legalen (d.h. ausführbaren) Operationsfolgen $(\sigma_1, \dots, \sigma_m)$ gilt $\sum_{i=1}^m \Delta(\sigma_i) \geq 0$.
- (ii) Δ ist möglichst gut gewählt.

Was Eigenschaft (ii) bedeutet, wird später noch spezifiziert. Für $\sigma \in S$ ist $\Delta(\sigma)$ die *Veränderung des Tokenkontos* durch die Operation σ . Wenn $\Delta(\sigma) > 0$, dann zahlt die Operation auf das Konto ein, falls $\Delta(\sigma) < 0$, dann hebt sie vom Konto ab. $A(\sigma) := T(\sigma) + \Delta(\sigma)$ nennen wir dann die *amortisierte Laufzeit* von σ . (In der Vorlesung wird $A(\sigma)$ auch Tokenlaufzeit genannt.) Entsprechend ist $A(\sigma_1, \dots, \sigma_m) := \sum_{i=1}^m A(\sigma_i)$ die amortisierte Laufzeit der Operationsfolge $(\sigma_1, \dots, \sigma_m)$.

Eigenschaft (i) sagt aus, dass das Tokenkonto nie negativ ist. Die Sinnhaftigkeit eines stets nichtnegativen Kontos ergibt sich aus folgender Beobachtung:

$$\begin{aligned} A(\sigma_1, \dots, \sigma_m) &= \sum_{i=1}^m A(\sigma_i) = \sum_{i=1}^m (T(\sigma_i) + \Delta(\sigma_i)) \\ &= T(\sigma_1, \dots, \sigma_m) + \underbrace{\sum_{i=1}^m \Delta(\sigma_i)}_{\geq 0} \geq T(\sigma_1, \dots, \sigma_m) \end{aligned}$$

Die amortisierte Laufzeit der Operationsfolge ist also eine *obere Schranke* für ihre tatsächliche Laufzeit. Jede Tokeneinheit steht für eine gewisse konstante Menge an Laufzeit. Damit



Abbildung 1: Pinguine

wird auch klar, was die (möglicherweise nicht ganzzahlige) Tokenzahl auf dem Konto aussagt: Es ist genau der Wert, um den die amortisierte Laufzeit die durch T gegebene (obere Schranke für die) tatsächliche Laufzeit übersteigt.

Nun zu Eigenschaft (ii). Das wichtigste Ziel ist, dass die amortisierten Laufzeiten $A(\sigma_1, \dots, \sigma_m)$ von Operationsfolgen asymptotisch möglichst klein sind, damit man gute obere Schranken für die tatsächliche Laufzeit von Operationsfolgen erhält.

Zu diesem Zwecke ist es oft zielführend, Δ so zu wählen, dass $\max_{\sigma \in S}(A(\sigma))$ möglichst klein ist, d.h. die Operation mit der schlechten amortisierte Laufzeit soll möglichst geringe amortisierte Laufzeit besitzen. Wenn dieses Ziel erreicht ist, ist $\mathcal{O}(m \cdot \max_{\sigma \in S}(A(\sigma)))$ eine obere Schranke für die asymptotische Laufzeit von Worst-Case-Operationsfolgen (wobei die Länge m der Operationsfolgen die asymptotische Variable ist). Bei einer hinreichend guten Wahl von Δ und Analyse ist diese obere Schranke oft nicht schlecht.

Aufgabe 4.2 (P) c-universelles Hashing mit Pinguinen

Die Pinguingattungen (vgl. auch Abbildung 1) {Brillenpinguin, Zwergpinguin, Eselspinguin, Kaiserpinguin, Goldschopfpinguin} sollen in einer Hash-Tabelle der Größe $m = 4$ untergebracht werden. Es seien folgende Hashfunktionen gegeben:

f_1 :	Brillenping.	$\mapsto 4$	Zwergp ping.	$\mapsto 2$	Eselsping.	$\mapsto 2$	Kaiserp ping.	$\mapsto 1$	Goldschopf ping.	$\mapsto 4$
f_2 :	Brillenping.	$\mapsto 3$	Zwergp ping.	$\mapsto 4$	Eselsping.	$\mapsto 2$	Kaiserp ping.	$\mapsto 3$	Goldschopf ping.	$\mapsto 4$
f_3 :	Brillenping.	$\mapsto 2$	Zwergp ping.	$\mapsto 2$	Eselsping.	$\mapsto 4$	Kaiserp ping.	$\mapsto 1$	Goldschopf ping.	$\mapsto 1$
f_4 :	Brillenping.	$\mapsto 1$	Zwergp ping.	$\mapsto 3$	Eselsping.	$\mapsto 3$	Kaiserp ping.	$\mapsto 4$	Goldschopf ping.	$\mapsto 4$
g_1 :	Brillenping.	$\mapsto 1$	Zwergp ping.	$\mapsto 1$	Eselsping.	$\mapsto 3$	Kaiserp ping.	$\mapsto 2$	Goldschopf ping.	$\mapsto 3$
g_2 :	Brillenping.	$\mapsto 2$	Zwergp ping.	$\mapsto 4$	Eselsping.	$\mapsto 2$	Kaiserp ping.	$\mapsto 3$	Goldschopf ping.	$\mapsto 4$
g_3 :	Brillenping.	$\mapsto 4$	Zwergp ping.	$\mapsto 4$	Eselsping.	$\mapsto 1$	Kaiserp ping.	$\mapsto 4$	Goldschopf ping.	$\mapsto 2$
g_4 :	Brillenping.	$\mapsto 3$	Zwergp ping.	$\mapsto 1$	Eselsping.	$\mapsto 2$	Kaiserp ping.	$\mapsto 3$	Goldschopf ping.	$\mapsto 3$
g_5 :	Brillenping.	$\mapsto 4$	Zwergp ping.	$\mapsto 2$	Eselsping.	$\mapsto 2$	Kaiserp ping.	$\mapsto 2$	Goldschopf ping.	$\mapsto 3$

In der Vorlesung haben wir den Begriff der c -universellen Hashfunktionen kennengelernt.

- (a) Geben Sie für die Familie $\mathcal{H}_1 = \{f_1, f_2, f_3, f_4\}$ das kleinste c an, so dass \mathcal{H}_1 c -universell ist.
- (b) Finden Sie eine möglichst kleine Familie $\mathcal{H}_2 \subseteq \{g_1, g_2, g_3, g_4, g_5\}$, die 1-universell ist.

Untermauern Sie Ihre Aussagen mit glaubwürdigen Argumenten.

Aufgabe 4.3 (P) Universelles Hashing mit Chaining

Veranschaulichen Sie Hashing mit Chaining. Die Größe m der Hash-Tabelle ist in den folgenden Beispielen jeweils die Primzahl 11. Die folgenden Operationen sollen nacheinander ausgeführt werden.

Insert	3, 11, 9, 7, 14, 56, 4, 12, 15, 8, 1
Delete	56
Insert	25

Der Einfachheit halber sollen die Schlüssel der Elemente die Elemente selbst sein.

- a) Verwenden Sie zunächst die Hashfunktion

$$g(x) = 5x \mod m.$$

- b) Verwenden Sie jetzt die Hashfunktion

$$h(x) = \mathbf{a} \cdot \mathbf{x} \mod m$$

nach dem aus der Vorlesung bekannten Verfahren für einfache universelle Hashfunktionen, wobei $\mathbf{a} = (7, 5)$ und $\mathbf{x} = (\lfloor \frac{x}{2^w} \rfloor \mod 2^w, x \mod 2^w)$ für $w = \lfloor \log_2 m \rfloor = \lfloor 3.45\dots \rfloor = 3$ gilt und der Ausdruck $\mathbf{a} \cdot \mathbf{x}$ ein Skalarprodukt bezeichnet. Vergleichen Sie die verwendete Hashfunktion mit der aus der vorigen Teilaufgabe.

Aufgabe 4.4 [4 Punkte] (H) Amortisationsschema

Wir betrachten eine Folge von $m \geq 1$ Inkrement-Operationen iterativ angewandt auf 0, das heißt, dass die k -te Inkrementoperation σ_k die Zahl $k - 1$ zur Zahl k inkrementiert. Wir nehmen an, dass alle Zahlen in *Dezimalschreibweise* geschrieben werden, also mit den Ziffern 0, 1, …, 9. Außerdem nehmen wir (vereinfachend) an, dass die Änderung einer Stelle (von x zu $x + 1$ für alle $x \in \{0, \dots, 8\}$ bzw. von 9 zu 0) Laufzeit 1 hat und die Laufzeit jeder Inkrement-Operation gerade die Anzahl der Stellen ist, die durch die Operation geändert werden.

- (a) Definieren Sie ein zulässiges Amortisationsschema $\Delta : \{\sigma_1, \sigma_2, \dots\} \rightarrow \mathbb{R}$ der Bankkonto-Methode, sodass für jede Inkrementoperation σ_k die amortisierte Laufzeit $A(\sigma_k) = T(\sigma_k) + \Delta(\sigma_k)$ gerade $\frac{10}{9}$ beträgt (hierbei sei $T(\sigma_k)$ die tatsächliche Laufzeit von σ_k).

Sie erhalten einen Teil der Punkte, wenn Sie eine amortisierte Laufzeit herleiten, die zwar mehr als $\frac{10}{9}$ beträgt, aber zumindest konstant ist.

- (b) Zeigen Sie, dass Ihr Amortisationsschema zulässig ist, indem Sie nachweisen, dass das Tokenkonto stets nichtnegativ ist.

Hinweis: Verwenden Sie in Ihrem Amortisationsschema für jede Operation σ_k die Anzahl $S_{x \rightarrow x+1}(\sigma_k)$ der Stellen, die durch σ_k von x zu $x + 1$ geändert werden, sowie die Anzahl $S_{9 \rightarrow 0}(\sigma_k)$ der Stellen, die durch σ_k von 9 zu 0 geändert werden.

Aufgabe 4.5 [5 Punkte] (H) Universelles Hashing mit Chaining

Veranschaulichen Sie Hashing mit Chaining. Die Größe m der Hash-Tabelle ist in den folgenden Beispielen jeweils die Primzahl 19. Die folgenden Operationen sollen nacheinander ausgeführt werden nach dem exakt selben Schema wie in der Tutoraufgabe dieses Blattes.

Insert	128, 55, 60, 3, 19, 1234
Delete	60, 55
Insert	76, 60

Der Einfachheit halber sollen die Schlüssel der Elemente die Elemente selbst sein.

- a) Verwenden Sie zunächst die Hashfunktion

$$g(x) = 3x \mod m.$$

- b) Verwenden Sie jetzt die Hashfunktion

$$h(x) = \mathbf{a} \cdot \mathbf{x} \mod m$$

nach dem aus der Vorlesung bekannten Verfahren für einfache universelle Hashfunktionen, wobei $\mathbf{a} = (3, 1, 3)$ und $\mathbf{x} = (\lfloor \frac{x}{2^w} \rfloor \mod 2^w, \lfloor \frac{x}{2^w} \rfloor \mod 2^w, x \mod 2^w)$ für $w = \lfloor \log_2 m \rfloor = \lfloor 4.24 \dots \rfloor = 4$ gilt und der Ausdruck $\mathbf{a} \cdot \mathbf{x}$ wieder das Skalarprodukt bezeichnet.

Hinweis: $2^{2w} = 2^8 = 256$ für $w = 4$.

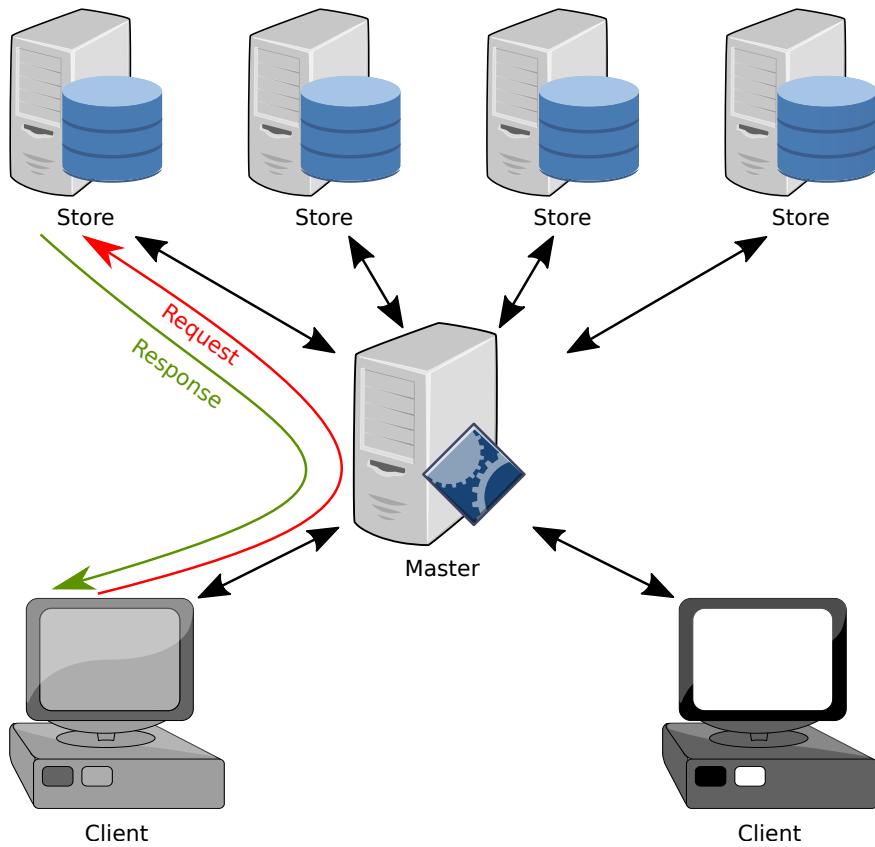


Abbildung 2: Aufbau der verteilten Hashtabelle

Aufgabe 4.6 [11 Punkte] (H) Verteilte Hashtabelle

Eine verteilte Hashtabelle (*distributed hash table, DHT*) erlaubt es, Daten effizient auf im Netzwerk verteilten Instanzen zu speichern. In dieser Aufgabe geht es darum, eine einfache verteilte Hashtabelle in Java zu implementieren. Unsere Hashtabelle soll `String`-Objekte als Schlüssel verwenden und diese auf `int`-Zahlen abbilden.

Unser System besteht aus drei Komponenten - dem *Master*, den *Stores* und den *Clients* (vgl. Abbildung 2). Die Stores verwalten jeweils einen Teil der Daten und können im Netzwerk auf unterschiedlichen Rechnern laufen. Die Clients entsprechen den Nutzern, sie möchten Daten im DHT ablegen oder Daten aus dem DHT auslesen. Der Master schließlich koordiniert die Kommunikation zwischen den Clients und den Stores. Er ordnet jedem *Request* anhand des Schlüssels des abgefragten oder gesetzten Datums den korrekten Store zu. Anschließend leitet er den Request an den Store weiter und wartet auf dessen Antwort. Schließlich leitet er die Antwort des Stores zurück an den Client. Für die Zuordnungen der Requests zu den Stores verwendet der Master *hashing*. Auf diese Weise muss er nicht speichern, welcher Schlüssel auf welchem Store gespeichert wird. Dies ist ein zentrales Konzept eines DHT, da es erlaubt, dass mehr Schlüssel (und so zugeordnete Werte) gespeichert werden als irgendein Rechner alleine verwalten kann.

Implementieren Sie das System nun schrittweise, wie in den folgenden Teilaufgaben beschrieben. Im Ordner `distry-angabe/` sind Vorlagen für die Lösungen der Teilaufgaben zu finden.

- Implementieren Sie zunächst die Klasse `HashString`, die den Hashwert zu einem Schlüssel vom Typ `String` berechnet. Es soll dabei eine Hashfunktion aus der 1-universellen Familie von Hashfunktionen verwendet werden, die in ab Folie 164 der Vorlesung be-

schrieben wird. Bedenken Sie dabei, dass Sie zu Beginn keine Information darüber haben, welche Form die Strings haben werden, die die Clients später als Schlüssel verwenden. Sie müssen die Komponenten des Vektors `a` also sukzessive generieren und speichern, damit Sie zum gleichen Schlüssel stets den gleichen Hashwert berechnen. Bedenken Sie, dass der Hash aus mehreren Threads gleichzeitig berechnet werden kann, die Berechnung muss also *threadsicher* sein. Implementieren Sie die Methode `int hash(String key)` wie im Quelltext vorgegeben; Sie dürfen die Klasse sonst beliebig erweitern. **Hinweis:** Sie dürfen die Klasse `ArrayList` verwenden!

- b) Implementieren Sie nun die Klasse `Client`. Der Client bittet nach dem Start den Benutzer um einen Befehl. Es stehen dabei die folgenden Befehle zur Verfügung:

Befehl	Beschreibung
<code>store key value</code>	Dieser Befehl setzt den Wert zum Schlüssel <code>key</code> auf den Wert <code>value</code> .
<code>read key</code>	Dieser Befehl liest den Wert zum Schlüssel <code>key</code> aus dem DHT aus.

Aus dem Befehl baut der Client ein `Request`-Objekt auf, welches er anschließend an den Master über eine TCP-Verbindung schickt. Nun muss der Client auf eine Antwort des Masters warten. Im Falle eines `store`-Requests bestätigt die Antwort des Masters, dass der Schlüssel im DHT auf den jeweiligen Wert gesetzt wurde. Bei einem `read`-Request antwortet der Master mit dem gelesenen Wert. Befindet sich zum gegebenen Schlüssel kein Wert im DHT, so antwortet der Master mit dem Wert `SerializableOptional.empty` (als Rückgabewert der `getValue()`-Methode von `ReadResponse`).

- c) Implementieren Sie nun die Klasse `Store`. Ein Store verbindet sich nach dem Start mit dem Master. Er nutzt dazu das TCP-Protokoll und eine andere Port-Nummer als die Clients. Anschließend empfängt er in einer Endlosschleife Requests vom Master, die er abarbeitet und beantwortet. Er hält dazu die gespeicherten Daten in einem `java.util.Hashtable`. **Hinweis:** Der Store erhält vom Master nur diejenigen Requests, die tatsächlich für ihn bestimmt sind.
- d) Implementieren Sie schließlich die Klasse `Master`. Der Master wartet nach dem Start zunächst darauf, dass sich alle Stores zu ihm verbinden. Die Stores bleiben über die gesamte Laufzeit mit dem Master verbunden. Ein Verbindungsabbruch soll, nachdem dieser festgestellt wurde, zur Folge haben, dass sich der Master mit einer entsprechenden Fehlermeldung beendet (insbesondere muss der Master also nach der anfänglichen Verbindungsphase nie wieder neue Verbindungen von Stores annehmen). Die Anzahl an Stores ist vorkonfiguriert im Quelltext des Masters. Nachdem sich alle Stores mit dem Master verbunden haben, beginnt dieser auf Anfragen der Clients zu warten. Achten Sie bei Ihrer Implementierung darauf, dass sich mehrere Clients gleichzeitig mit dem Master verbinden können müssen. Achten Sie ferner darauf, dass sich diese Clients dann und nur dann gegenseitig aufhalten, wenn sie auf den gleichen Store zugreifen müssen.

Jeder Store verfügt über einen beliebigen, aber nach dem Start festen Index. Ein bestimmter Request x wird vom Master an den i -ten Store weitergeleitet, wobei sich i durch Berechnung des Hashwertes des Schlüssels des jeweiligen Requests (`x.getKey()`) vom Clienten ergibt. Der Master wartet anschließend auf die Antwort des Stores und leitet diese an den Client weiter. Während der gesamten Kommunikation mit dem Store muss ausgeschlossen werden, dass weitere Anfragen an den selben Store verschickt werden. Nach Abschluss der Kommunikation mit dem Store muss dieser sofort wieder für weitere Anfragen zur Verfügung stehen. Der Master speichert zu keinem Zeitpunkt, an welchen Store er einen bestimmten Request weitergeleitet hat.

Stattdessen *vergisst* er alles über einen bestimmten Request nach der Antwort an den jeweiligen Client. Achten Sie darauf, dass die Verbindung zum Client nach der Antwort des Masters geschlossen wird.

Ein Kommunikationsfehler mit einem Client soll nicht dazu führen, dass der Master beendet wird oder nicht mehr ordnungsgemäß funktioniert. Stattdessen soll lediglich eine entsprechende Fehlermeldung auf der Konsole des Masters ausgegeben werden.

Hinweis: Achten Sie darauf, *Race Conditions* zu vermeiden!

Beachten Sie zusätzlich folgende **Hinweise**:

- Die verschiedenen Komponenten des Systems benötigen jeweils eine Konfiguration, die z.B. die zu verwendenden Ports spezifiziert. Als Vereinfachung schlagen wir vor, dass Sie die Konfiguration vollständig in Ihrem Quelltext halten (wie in den Vorlagen angedeutet).
- Sie müssen in dieser Aufgabe Informationen über eine Netzwerkverbindung verschicken. Dies geht am besten durch die Benutzung von *Object Streams*. Die vorgegebenen Request- und Responseklassen sind extra dazu ausgelegt, über Object Streams verschickt zu werden. Als Beispiel sei ein `Socket` mit dem Namen `socket` gegeben. Um ein Objekt `request` über diesen Socket zu verschicken können Sie folgenden Code verwenden:

```
1 ObjectOutputStream out = new ObjectOutputStream(socket.
2     getOutputStream());
3 out.writeObject(request);
4 out.flush();
```

Um dagegen über den Socket ein Objekt `response` vom Typ `IResponse` zu empfangen, genügen die folgenden Zeilen:

```
1 ObjectInputStream in =
2     new ObjectInputStream(socket.getInputStream());
3 IResponse response = (IResponse) in.readObject();
```

Achten Sie darauf, dass die gesendeten Objekte jeweils den Typen haben, den der Empfänger erwartet!

- Die Request- und Responseklassen implementieren eine Variante des sogenannten Visitor-Patterns¹. Sie können so einfach eine Fallunterscheidung nach Typ der Antwort implementieren, indem sie Lambda-Ausdrücke wie folgt verwenden. Im Beispiel zeigen wir, wie man je nach UnterkLASSE von `Response` unterschiedlichen Code ausführen kann:

```
1 ResponseVisitor rv = new ResponseVisitor();
2 rv._( (ReadResponse readResponse) -> {
3     SerializableOptional<Integer> result = readResponse.getValue();
4     if(result.isPresent())
5         System.out.println("Read response with value " + result.get() + " .");
6     else
7         System.out.println("Read response: Unknown key!");
8 }, (StoreResponse storeResponse) -> {
9     System.out.println("Store successful!");
10 });
11 response.accept(rv);
```

- Die Aufgabe setzt die Version 8 von Java voraus.

¹https://en.wikipedia.org/wiki/Visitor_pattern

- Sie dürfen in dieser Aufgabe auf Fehler mit Programmabbruch reagieren. Lediglich bei Kommunikationsfehlern zwischen Master und Client soll der Master anschließend weitere Anfragen von Clients bearbeiten können, also nicht beendet werden oder in einen Fehlerzustand geraten.