Abgabe: **22.05.2016** (bis 23:59 Uhr)

Aufgabe 5.1 (P) Dynamische Arrays

Wir betrachten die Konto-Methode angewandt auf dynamische Arrays für beliebige Werte α und β mit $\alpha > \beta > 1$. Hierbei beschränken wir uns auf den analytisch etwas einfacheren Fall, dass $\alpha, \beta \in \mathbb{N}$. Die Analyse für nicht ganzzahlige α und β ist ähnlich. [1]

Man erinnere sich, dass dynamische Arrays über die Methoden `pushBack` (= Einfügen eines Elements am Ende der „Liste“) und `popBack` (Löschen des letzten Elements der „Liste“) verwaltet werden. Im Folgenden bezeichne n stets die aktuelle Anzahl der Elemente im Array, und w die Größe des Arrays. Unter bestimmten Voraussetzungen rufen `pushBack` und `popBack` die Methode `reallocate` als Untermethode auf:

Wenn bei `pushBack` *vor* der Einfügung des Elements das Array voll ist (d.h. $n = w$), dann wird die Methode `reallocate` aufgerufen, die ein neues Array der Größe βn anlegt und alle alten Elemente in das neue Array kopiert. Anschließend wird das neue Element eingefügt.

Wenn bei `popBack` nach der Löschung des letzten Elements der Füllstand des Arrays nur noch maximal $1/\alpha$ beträgt (d.h. $\alpha \cdot n \leq w$), dann wird die Funktion `reallocate` aufgerufen, die ein neues Array der Größe βn anlegt und alle Elemente in das neue Array kopiert. Wir fassen `reallocate` als eigenständige Methode auf.

Es ist nicht schwierig zu sehen, dass die tatsächliche Laufzeit von `pushBack` und `popBack` konstant ist (d.h. durch eine Konstante nach oben beschränkt), und die Laufzeit von `reallocate` durch $\mathcal{O}(1) + c \cdot n$ beschränkt ist, wobei c eine Konstante ist und n die Zahl der kopierten Elemente. Wie in der Vorlesung ist es legitim, $c = 1$ zu setzen, indem wir annehmen, dass wir die Laufzeit in einer Einheit messen, die gerade der Laufzeit einer einzelnen Kopieroperation entspricht. Wir erhalten damit:

$$\begin{aligned} T(\texttt{pushBack}) &\in \mathcal{O}(1) \\ T(\texttt{popBack}) &\in \mathcal{O}(1) \\ T(\texttt{reallocate}) &= \mathcal{O}(1) + n, \text{ wobei } n = \text{Anzahl der kopierten Elemente} \end{aligned}$$

Ziel dieser Aufgabe ist der Nachweis mit einer amortisierten Analyse, dass die Laufzeit von Operationen der Länge m auf einem zu Beginn leeren dynamischen Array in $\mathcal{O}(m)$ liegt (zu Beginn hat das Array Größe 1).

Dafür legen wir fest, dass `pushBack` $\beta/(\beta - 1)$ Token auf das Konto einzahlt, `popBack` $\beta/(\alpha - \beta)$ Token einzahlt, und `reallocate` n Token vom Konto abhebt, wenn n Elemente

[1] In einem solchen Fall muss die Größe des neuen Arrays mithilfe der Aufrundungsfunktion als $\lceil \beta \cdot n \rceil$ definiert werden, was eine entsprechende Anpassung der Funktion Δ erfordert und die Analyse etwas komplizierter macht. In der Literatur werden in solchen Fällen eigentlich notwendige Auf- oder Abrundungen nicht selten einfach weggelassen, um die Analyse zu vereinfachen. Die Analyse ist dann zwar nicht mehr absolut präzise, aber zumindest größenordnungsmäßig „gut genug“ und man erkennt die dahinterstehende Idee. Der Leser ist dann gefordert, die Details selber zu ergänzen. Auf einen Blick zu erkennen, wann solche Vereinfachungen möglich sind, erfordert etwas Erfahrung.

kopiert werden, also:

$$\begin{aligned}
\Delta(\texttt{pushBack}) &= \beta/(\beta-1) \\
\Delta(\texttt{popBack}) &= \beta/(\alpha-\beta) \\
\Delta(\texttt{reallocate}) &= -n, \text{ wobei } n = \text{Anzahl der kopierten Elemente}
\end{aligned}$$

(a) Zeigen Sie, dass dieses Amortisationsschema zulässig ist, indem Sie zeigen, dass das Tokenkonto zu jedem Zeitpunkt nichtnegativ ist.

Hinweis: Bezeichne n_1 die Zahl der Elemente *unmittelbar* nach einem `reallocate` (im Falle von `pushBack` also noch vor der Einfügung des neuen Elements). Machen Sie sich klar, dass das Array *zu diesem Zeitpunkt* Größe $w_1 := \beta \cdot n_1$ hat, und $w_1 - n_1$ Positionen frei sind. Das nächste `reallocate` wird erst dann aufgerufen, wenn für die Zahl n der Elemente entweder $n = w_1$ oder $\alpha n \leq w_1$ gilt.

(b) Zeigen Sie, dass unter diesem Amortisationsschema die amortisierte Laufzeit jeder Operation in $\mathcal{O}(1)$ liegt, und folgern Sie, dass die Worst-Case-Laufzeit für Operationsfolgen der Länge m in $\mathcal{O}(m)$ liegt.

Aufgabe 5.2 (P) Hashing mit Linear-Probing

Veranschaulichen Sie Hashing mit Linear Probing. Die Größe der Hashtabelle ist dabei jeweils $m = 13$. Führen Sie die folgenden Operationen aus:

Insert	$16, 3, 12, 17, 29, 10, 24$
Delete	16
Insert	$5, 1, 14$
Delete	10
Insert	10
Delete	1

Verwenden Sie die Hashfunktion

$$h_3(x) = 3x \mod 13.$$

Bei dieser Aufgabe sind die Schlüssel der Elemente die Elemente selbst.

Beim Löschen soll die dritte Methode aus der Vorlesung verwendet werden, d.h. die Wiederherstellung der folgenden Invariante: Für jedes Element e in der Hashtabelle mit Schlüssel $k(e)$, aktueller Position j und optimaler Position $i = h_3(k(e))$ sind alle Positionen $i, (i+1) \bmod m, (i+2) \bmod m, \ldots, j$ der Hashtabelle belegt. Bei dieser Aufgabe soll keine dynamische Größenanpassung der Hashtabelle stattfinden.

Aufgabe 5.3 (P) Double-Hashing

Doppel-Hashing ist eine Methode zur Kollisionsbehandlung. Bei Kollisionen kommt eine Sondierungsfunktion zum Einsatz, die eine sekundäre Hashfunktion beinhaltet:

$$s(x, i) = i \cdot h_2(x), i \in \mathbb{N}_0$$

Diese Sondierungsfunktion wird angewendet, falls der durch die primäre Hashfunktion $h_1(x)$ berechnete Index bereits besetzt ist. Dabei wird i beginnend bei 0 bei jedem Versuch um 1 erhöht. Die vollständige Hashfunktion lautet dann:

$$h(x,i) = (h_1(x) + s(x,i)) \mod m$$

Verwenden Sie im Folgenden die Hashfunktionen

$$h_1(x) = (3x + 1) \bmod m$$
$$h_2(x) = (1 + (x \bmod (m-1)))$$

a) Geben Sie die vollständige Hashfunktion $h(x,i)$ für eine Tabelle der Länge $m = 13$ an.

b) Veranschaulichen Sie schrittweise das Einfügen der Schlüssel $12, 23, 13, 56, 26, 45, 24, 94, 42$ in eine Hashtabelle der Länge $m = 13$.

Aufgabe 5.4 (P) Zusatzaufgabe: Der betrunkene Übungsleiter

Wir betrachten einen torkelnden Übungsleiter an einer Kletterwand, der die folgenden beiden Operationen durchführen kann:

- `hoch`,
- `runter(int` k`)`.

Die Starthöhe des Übungsleiters beträgt 0 Meter. Durch die Operation `hoch` steigt der Übungsleiter von seiner aktuellen Position aus genau einen Meter höher. Diese Operation hat die Laufzeit 1. Durch die Operation `runter(int` k`)` fällt der Übungsleiter von seiner aktuellen Position aus exakt $\min\{h, k\}$ Meter nach unten, wobei h die aktuelle Höhe des Übungsleiters ist. (Das bedeutet, dass der Übungsleiter nie tiefer als seine Starthöhe sinken kann). Wir nehmen hierbei an, dass k stets eine nicht-negative natürliche Zahl ist. Die Operation `runter(int` k`)` hat die Laufzeit $\min\{h, k\}$.

Zeigen Sie mithilfe der Bankkonto-Methode, dass die amortisierten Laufzeiten der Operationen `hoch` und `runter(int` k`)` in $\mathcal{O}(1)$ liegen.

Aufgabe 5.5 [5 Punkte] **(H) Stapelschlange**

In dieser Aufgabe geht es darum, einen Algorithmus, der eine Queue für `Integer`-Zahlen mittels zweier Stacks implementiert, hinsichtlich seiner Laufzeit zu untersuchen.

```
1  class Stapelschlange {
2    private Stack s1 = new Stack();
3    private Stack s2 = new Stack();
4
5    public void enqueue(int v) {
6      s1.push(v);
7    }
8
9    public int dequeue() {
10     if(s2.isEmpty())
11       while(!s1.isEmpty())
12         s2.push(s1.pop());
13     return s2.pop();
14   }
15 }
```

Die Klasse `Stack` hat dabei folgende Methoden:

Methode	Beschreibung	Laufzeitklasse
`void push(int v)`	legt eine Zahl auf den Stack	$\mathcal{O}(1)$
`int pop()`	nimmt die oberste Zahl vom Stack	$\mathcal{O}(1)$
`boolean isEmpty()`	prüft, ob der Stack leer ist	$\mathcal{O}(1)$

a) [1] Geben Sie die Laufzeitklasse der Worst Case-Laufzeit eines Aufrufs der `dequeue()`-Methode in Abhängigkeit der aktuellen Größe der Queue n in Landau-Notation an.

b) [4] Überlegen Sie sich ein Amortisierungsschema, welches die amortisierte Laufzeit der einzelnen Operationen minimiert. Nennen Sie die amortisierten Laufzeitklassen der `enqueue(int v)`- und `dequeue()`-Methode, die sich aus Ihrem Amortisierungsschema ergeben. Zeigen Sie die Richtigkeit Ihres Amortisierungsschemas (das Tokenkonto darf niemals negativ werden) und der resultierenden Laufzeitklassen.

Aufgabe 5.6 [4 Punkte] **(H) Hashing mit *Linear Probing***

Veranschaulichen Sie Hashing mit Linear Probing. Führen Sie folgende Operationen in der gegebenen Reihenfolge aus:

Insert:	$3, 42, 51, 17, 29, 23, 24$
Delete:	3
Insert:	$18, 40, 14$
Delete:	23
Insert:	23
Delete:	40

Die Größe der Hashtabelle ist $m = 13$. Die Schlüssel der Elemente sind die Elemente selbst. Verwenden Sie die Hashfunktion

$$h(x) = 3x \mod 13.$$

Beim Löschen soll gemäß der dritten Methode aus der Vorlesung die folgende Invariante sichergestellt bleiben: Für jedes Element e in der Hashtabelle mit aktueller Position j und

optimaler Position $i = h(e)$ sind alle Positionen $i, (i+1) \bmod m, (i+2) \bmod m, \ldots, j$ der Hashtabelle belegt. Es soll keine dynamische Größenanpassung der Hashtabelle stattfinden.

Hinweis: Für natürliche Zahlen x gilt $3x \bmod 13 = (3(x \bmod 13)) \bmod 13$.

1. Operation: Insert(3):

0	1	2	3	4	5	6	7	8	9	10	11	12
									3			

2. Operation: Insert(42):

0	1	2	3	4	5	6	7	8	9	10	11	12
									3	42		

3. Operation: Insert(51):

0	1	2	3	4	5	6	7	8	9	10	11	12
									3	42		

4. Operation: Insert(17):

0	1	2	3	4	5	6	7	8	9	10	11	12
									3	42		

5. Operation: Insert(29):

0	1	2	3	4	5	6	7	8	9	10	11	12
									3	42		

6. Operation: Insert(23):

0	1	2	3	4	5	6	7	8	9	10	11	12
									3	42		

7. Operation: Insert(24):

0	1	2	3	4	5	6	7	8	9	10	11	12
									3			

8. Operation: Delete(3):

0	1	2	3	4	5	6	7	8	9	10	11	12

9. Operation: Insert(18):

0	1	2	3	4	5	6	7	8	9	10	11	12

10. Operation: Insert(40):

0	1	2	3	4	5	6	7	8	9	10	11	12

11. Operation: Insert(14):

0	1	2	3	4	5	6	7	8	9	10	11	12

12. Operation: Delete(23):

0	1	2	3	4	5	6	7	8	9	10	11	12

13. Operation: Insert(23):

0	1	2	3	4	5	6	7	8	9	10	11	12

14. Operation: Delete(40):

0	1	2	3	4	5	6	7	8	9	10	11	12

Aufgabe 5.7 [8 Punkte] **(H) Programmieraufgabe: DoubleHashie**

In dieser Aufgabe geht es darum, eine Hashtabelle mit doppeltem Hashing zu implementieren. Ein Programmgerüst des in Java zu implementierenden Programms wird mit diesem Übungsblatt verteilt. Kommentare im gegebenen Quelltext enthalten wichtige Hinweise und Beispiele.

Als Grundlage für das doppelte Hashing ist folgende Funktion gegeben:

$$h(x,i) \quad = \quad (h(x) + i * h'(x)) \bmod m \tag{1}$$

Die Primzahl m bezeichnet die Größe der Hashtabelle, i die Anzahl vorangegangener Hashings dieses Schlüssels.

a) Implementieren Sie zunächst die Klasse `DoubleHashInt`, die den Hashwert $h(x)$ bzw. $h'(x)$ zu einem Schlüssel x vom Typ `Integer` berechnet. Achten Sie darauf, dass Ihre Implementierung garantiert, dass verschiedene Werte von i in (1) zu verschiedenen

Hashwerten führen. Implementieren Sie die Methoden `int hash(String key)` und `int hashTick(String key)` wie im Quelltext vorgegeben; Sie dürfen die Klasse sonst beliebig erweitern.

b) Implementieren Sie nun die Klasse `DoubleHashString`, die den Hashwert $h(x)$ bzw. $h'(x)$ zu einem Schlüssel x vom Typ `String` berechnet. Es soll dabei jeweils eine Hashfunktion aus der 1-universellen Familie von Hashfunktionen verwendet werden, die ab Folie 164 der Vorlesung beschrieben wird. Bedenken Sie, dass Sie zu Beginn keine Information darüber haben, welche Form die Strings haben werden, die in die Hashtabelle eingefügt werden. Sie müssen die Komponenten des Vektors `a` im Skalarprodukt also sukzessive generieren und speichern, damit Sie zum gleichen Schlüssel stets den gleichen Hashwert berechnen. Als Vereinfachung dürfen Sie während der Berechnung des Skalarproduktes annehmen, dass die Hashtabelle eine Größe m' hat, die zur Aufteilung des Strings in Komponenten der Größe eines Buchstaben (16 Bit) passt. Die tatsächliche Größe $m \leq m'$ fließt erst am Ende der Berechnung ein:

$$h(x) \quad = \quad h_a^{m'}(x) \bmod m \tag{2}$$

Achten Sie darauf, dass Ihre Implementierung garantiert, dass verschiedene Werte von i in (1) zu verschiedenen Hashwerten führen. Implementieren Sie die Methoden `int hash(String key)` und `int hashTick(String key)` wie im Quelltext vorgegeben; Sie dürfen die Klasse sonst beliebig erweitern. **Hinweis:** Sie dürfen die Klasse `ArrayList` verwenden!

c) Implementieren Sie die Methode `bool insert(K key, V value)` der Klasse `DoubleHashTable` . Die Methode fügt den Wert `value` mittels doppeltem Hashing für den Schüssel `key` in die Hashtabelle ein. Sie gibt `true` zurück, wenn das Einfügen erfolgreich verlaufen ist; falls die Hashtabelle voll ist, wird `false` zurückgegeben. Bereits zu einem Schlüssel vorhandene Werte sollen überschrieben werden können.

d) Implementieren Sie die Methode `Optional<V> find(K key)`, die nach dem Schlüssel `key` in der Hashtabelle sucht. Die Methode gibt durch den optionalen Rückgabewert genau dann einen Wert zurück, wenn der Schlüssel in der Hashtabelle gefunden wurde.

e) Implementieren Sie die Methode `int collisions()` der Klasse `DoubleHashTable`, die zurückgibt, wieviele Elemente nicht an der *optimalen Position* gespeichert sind. Eine Position ist dann für ein Element *optimal*, wenn sie durch nur einen Aufruf der Hashfunktion ermittelt werden kann.

f) Implementieren Sie die Methode `int maxRehashes()` der Klasse `DoubleHashTable`, die zurückgibt, wie oft die Hashfunktion für ein einzelnes Element während des Einfügens insgesamt maximal berechnet werden musste.

Bitte beachten Sie, dass Sie für eine funktionierende Implementierung die gegebenen Klassen noch an anderen Stellen als innerhalb der jeweiligen Methoden verändern müssen.