FAU, Informatik 2, AUD-Team
aud@i2.cs.fau.de

9. Übung

Abgabe bis 03.07.2017, 10:00 Uhr

Einzelaufgabe 9.1: Stapel und Warteschlange — 9 EP

Implementieren Sie die *generischen* ADTs `Stack<E>` und `Queue<E>` gemäß AuD-Vorlesung (siehe Folien 10-41 bzw. 10-47) mit folgenden Besonderheiten:

- Die Operationen des ADTs $Stack$ werden alle als Klassenmethoden mit den jeweiligen Signaturen (**ops**) aus der Vorlesung implementiert. Die Klasse bekommt zusätzlich einen `private`n Konstruktor, den Sie nach Belieben nutzen können. Behandeln Sie `null` als Aktualparameter für das erste Argument wie einen *leeren* $Stack$.
- Bei $Queue$ sind hingegen alle Operationen als Instanzmethoden umzusetzen, d.h. anstelle des ersten Arguments in den Signaturen ist `this` zu verwenden und anstelle der Operationen $create$ tritt der „*default*“-Konstruktor der Klasse.
- Die Klassen dürfen *keine* anderen Methoden haben, als die in den ADTs deklarierten **ops**. Nur `Stack` bekommt eine zusätzliche Methode `Stack<E> reverse(Stack<E> s)`, die einen neuen Stapel erzeugt, der die Elemente von `s` in umgekehrter Reihenfolge enthält.
- Die Klasse `Stack` muss *genau zwei* `private` Attribute haben: Eines vom Typ `Stack` und eines vom generischen Typ der Klasse.
  Die Klasse `Queue` muss *genau ein* `private`s Attribut haben, das vom Typ `Stack` mit gleichem generischen Typ wie `Queue` sein muss.
- Alle Operationen müssen „*wirkungsfrei*“ sein, d.h. alle Instanzen der beiden Klassen sind jederzeit *unveränderliche Objekte*.
- Sie dürfen *keine* Klassen aus der Java-API verwenden!

Einzelaufgabe 9.2: GenericDoublyLinkedCircularPriorityQueueWithComparableOrComparator — 12 EP

Implementieren Sie eine neue Klasse mit der prinzipiellen Semantik einer Prioritätswarteschlange namens `GenericDoublyLinkedCircularPriorityQueueWithComparableOrComparator`, die von `AbstractGenericDoublyLinkedCircularPriorityQueueWithComparableOrComparator` abgeleitet ist und deren Methoden die Kommentare der Oberklasse erfüllen.
Wichtig: Beachten Sie bitte unbedingt, dass Ihre Unterklasse ***keine*** weiteren Attribute oder innere Klassen deklarieren darf. Außerdem dürfen Sie aus der Java-API ***keine*** anderen Klassen oder Methoden benutzen, abgesehen von der bereits durch die Oberklasse erzwungenen Verwendung von `Comparator` bzw. `NoSuchElementException`.

Einzelaufgabe 9.3: Streuung - Theorie — 14 EP

„Director's Cut“ der Klausuraufgabe 6 vom 02.08.2012

In dieser Aufgabe sollen Sie mit verschiedenen Verfahren die ASCII-Zeichen $v_1, v_2, \ldots, v_8$ mit der folgenden primären Streufunktion $h_P(v_k)$ bzw. mittels Doppel-Streuung jeweils in eine initial leere Streutabelle einfügen. Geben Sie Ihre Lösung als `Hashing.pdf` ab.

k ▶	1	2	3	4	5	6	7	8
v_k	'g'	'C'	'8'	'@'	'G'	'*'	'+'	'2'
$h_P(v_k)$	2	6	3	3	2	5	6	5

$[h_P(v_k) := (v_k + 3) \,\%\, 8\,]$ ■

a) Fügen Sie $v_1, v_2, \ldots, v_8$ in dieser Reihenfolge zunächst mit der primären Funktion $h_P(v_k)$ in eine Streutabelle der Größe 8 nach folgendem Muster ein. Kollisionen sind dabei mit einer verketteten Liste aufzulösen und in der Tabelle als „$v_x \mapsto v_y \mapsto v_z \mapsto \perp$“ darzustellen.

Bucket	v_k (`verkettete Liste`)
0	
...	
7	

[„$\mapsto\perp$“ stellt das Ende der Liste („Nullzeiger“ als „allerletzter“ Eintrag) dar.]

b) Fügen Sie nun $v_1, v_2, \ldots, v_8$ ebenfalls in dieser Reihenfolge mit $h_P(v_k)$ in eine *neue* Streutabelle der Größe 8 ein. Lösen Sie Kollisionen diesmal aber mittels *linearem Sondieren mit Schrittweite* $+3$ auf. Tragen Sie in eine Tabelle nach folgendem Muster ein, welche Indizes in welcher Reihenfolge sondiert wurden und ob die Sondierung aufgrund einer Primär- ($\overset{P}{\mapsto}$) oder Sekundärkollision ($\overset{S}{\mapsto}$) erfolgen musste.

Fiktives Beispiel zur Demonstration der Syntax: „$7 \overset{S}{\mapsto} 3 \overset{P}{\mapsto} 4$“ bedeutet, dass der erste berechnete Streuwert 7 zu einer Sekundärkollision im bereits belegten Fach 7 führt, weshalb weiter im Fach 3 sondiert wird, das wiederum belegt ist, weshalb es zu einer Primärkollision kommt. Nach erneutem Sondieren wird das leere Fach 4 als Zielfach identifiziert.

v_k	*sondierte Buckets mit Kollisionen*	*Zielbucket*
'g'		
...		
'2'		

c) Beim Doppel-Hashing verwendet man zur Indizierung häufig eine Funktionsschar h_i, die sich aus der primären und sekundären Hash-Funktion (h_P bzw. h_S) zusammensetzt. Fügen Sie die Schlüssel $v_1, v_2, \ldots, v_8$ wieder in dieser Reihenfolge, aber diesmal mittels Doppel-Hashing in eine Streutabelle der Größe 8 ein – bitte beachten Sie dabei folgenden Zusammenhang: $h_0(v_k) = h_P(v_k)$. Geben Sie in der letzten Spalte an, welche Doppel-Hash-Funktion zum Zug kam.

k ▶		*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
	v_k	'g'	'C'	'8'	'@'	'G'	'*'	'+'	'2'
≈.	$h_P(v_k)$	2	6	3	3	2	5	6	5
▼	$h_S(v_k)$	1	0	3	4	4	3	4	4
1	$h_1(v_k)$	4	7	7	0	7	1	3	2
2	$h_2(v_k)$	6	0	3	5	4	5	0	7
3	$h_3(v_k)$	0	1	7	2	1	1	5	4
...	...				...				

■ $[h_P(v_k) := (v_k + 3)\ \%\ 8 \quad | \quad h_S(v_k) := (v_k + 3)\ \%\ 7 \quad | \quad h_i(v_k) = (h_P(v_k) + i \cdot (1 + h_S(v_k)))\%8]$

Bucket	v_k	*Hash-Funktion* (i)
0		
...		
7		

Gruppenaufgabe 9.4: Vergleichbare Punkte — 25 GP

Implementieren Sie eine Unterklasse `Punkt` der vorgegebenen Klasse `AbstrakterPunkt`, die einen Punkt $\mathcal{P}(x, y)$ in einem zweidimensionalen Koordinatensystem repräsentiert.

a) Befolgen Sie dazu die Javadoc-Kommentare in `AbstrakterPunkt` und vervollständigen Sie die Methoden zur Berechnung des *„Euklidischen Abstands“* bzw. der *„Manhattan-Distanz“* sowie die Methode `equals` mit der *konkret vorgegebenen* Äquivalenzrelation für 2D-Punkte.

b) Erweitern Sie die Klasse `Punkt` so, dass sie die Schnittstelle `Comparable<Punkt>` wie folgt implementiert: Wenn der *Euklidischen Abstand* eines Punktes $\mathcal{X}$ vom Ursprung $\mathcal{P}(0, 0)$ kleiner (größer) ist als der eines anderen Punktes $\mathcal{Y}$, dann soll der Punkt $\mathcal{X}$ in der Ordnung der kleinere (größere) und damit `x.compareTo(y) == -1` (`+1`) sein. Außerdem muss diese Ordnung *konsistent* mit `equals` sein, d.h. bei gleichem *Euklidischen Abstand* darf `compareTo` *nicht* 0 zurückgeben: Stattdessen ist der Punkt mit der größeren x-Koordinate und bei gleichen x-Koordinaten derjenige mit der größeren y-Koordinate auch der größere und die Methode muss die jeweilige Koordinatendifferenz (Vorzeichen beachten!) als Indikator zurückgeben.

c) Um Punkte auch anders ordnen zu können, sollen Sie nun die Klasse `PunktVergleicher` erstellen, die die Schnittstelle `Comparator<Punkt>` implementiert. Zusätzlich hat die Klasse einen Konstruktor, dem ein Referenzpunkt $\mathcal{Z}$ (`Punkt z`) übergeben wird. Wenn $\mathcal{Z}$ `null` ist, soll stattdessen der Koordinatenursprung $(0, 0)$ verwendet werden. Der `PunktVergleicher` ordnet die Punkte zwar auch nach dem Abstand zu diesem Punkt $\mathcal{Z}$, wobei aber zunächst nur der Abstand in der x-Komponente der Koordinaten beachtet wird: Punkte, deren Differenz in den x-Koordinaten zu $\mathcal{Z}$ kleiner sind, sind kleiner. Zwischen (nach dieser Regel) gleich weit entfernten Punkten wird nach der natürlichen Ordnung von `Punkt` geordnet, d.h. also nach derjenigen Ordnung, die Sie vorangehend in der Methode `compareTo` implementiert haben. Der Vergleich mit `null` muss nicht funktionieren und darf eine `NullPointerException` werfen. Implementieren Sie auch die zugehörige Methode `Comparator.equals` so, dass sie konsistent mit den Anforderungen der zugehörigen Java-API ist und trotzdem wann immer möglich `true` zurückgibt.

d) Um mehrere Punkte in einem Feld „aufsteigend“ zu sortieren, müssen Sie die Klassenmethoden `sortiere(Punkt p[])` bzw. `sortiereZentrum(Punkt p[], Punkt z)` in der Klasse `Punkt` wie folgt implementieren: Die erste Methode soll das übergebene Feld nach der natürlichen Ordnung von `Punkt` sortieren, während die zweite Methode die Punkte im Feld nach ihrem Abstand zum Punkt `z` (gemäß `PunktVergleicher`) anordnet. Verwenden Sie dazu die entsprechend geeigneten `sort`-Methoden der Java-API-Klasse `java.util.Arrays`.

35 EP + 25 GP = 60 Punkte