Praktikum Grundlagen der Programmierung

Prof. Dr. Harald Räcke, R. Palenta, A. Reuss, S. Schulze Frielinghaus WS 2016/17

Übungsblatt 8 Abgabefrist: 19.12.2016 (vor 05:00 Uhr)

Σ Punkte: 19 | Σ Bonuspunkte: 2

Aufgabe 8.1 (P) Variable Begrifflichkeiten

Im Folgenden werden wir uns mit den unterschiedlichen Begrifflichkeiten von Variablen in Java auseinandersetzen.

```
class Foo {
    int x;
    static int y;

    void f(int n) {
        int m;
    }
}
```

In der Klasse `Foo` finden wir 4 unterschiedliche Variablen vor. Die Variablen `x` und `y` werden unter dem Oberbegriff *Membervariable* (im Englischen oft auch *field variable*) geführt. Es wird aber noch zwischen diesen beiden Variablen unterschieden. Die Variable `x` kommt in jeder Instanz der Klasse `Foo` genau einmal vor, d.h. in jedem Objekt der Klasse `Foo`. Variablen mit dieser Eigenschaft werden in Java als *Objektvariablen* bezeichnet. Die Variable `y` mit dem Schlüsselwort `static` hingegen kommt genau einmal in dem gesamten Program vor. Egal wie viele Instanzen/Objekte der Klasse `Foo` existieren. Daher werden Variablen mit dieser Eigenschaft als *Klassenvariablen* bezeichnet. Von der Variable `n` wiederum gibt es so viele Instanzen, wie es Aufrufe von `f` gibt. Die Variable ist auch nur innerhalb der Methode `f` gültig. Solche Variablen werden in Java als *Parameter* bzw. *Parametervariablen* bezeichnet. Zu guter Letzt haben wir noch die Variable `m`. Variablen, die innerhalb einer Methode definiert sind, werden als *lokale Variablen* bezeichnet. Von diesen gibt es, wie für Parametervariablen, so viele Instanzen wie es Methodenaufrufe gibt.

Aufgabe 8.2 (P) Vergleich von Strings

In der Vorlesung wurde bereits besprochen, dass ein `String` kein primitiver Datentyp ist, sondern ein Referenzdatentypen. Die Werte primitive Datentypen können mit == verglichen werden. Werden Referenzdatentypen mittels == verglichen, wird überprüft, ob die beiden Referenzen auf das gleiche Objekt verweisen. Beachten Sie dazu das Programm `CompareObjects.java`, in dem der Vergleich zweier Strings mittels ==, die zwar den gleichen Wert enthalten, jedoch nicht auf das gleiche Objekt verweisen, zu `false` ausgewertet wird.

Schreiben Sie eine Methode `public static boolean equals(String a, String b)`, die die Werte der beiden `String`-Objekte `a` und `b` auf Gleichheit überprüft ohne die `equals`-Methode der Klasse `String` zu verwenden.

Diskutieren Sie im Tutorium, warum `v` im folgenden Codestück zu `true` ausgewertet wird.
`String z = "hallo"; String t = "hal" + "lo"; boolean v = (z == t);`

Diese Aufgabe soll Ihnen zeigen, warum Sie Strings immer mittels der `equals(String t)` Methode der Klasse `String` vergleichen sollten.

Lösungsvorschlag 8.2

Der Vergleich in `String z = "hallo"; String t = "hal" + "lo"; boolean v = z == t;` wird zu `true` ausgewertet, da der Compiler Optimierungen vornimmt und deshalb erkennt, dass der Wert der beiden Strings gleich ist. Deshalb wird nur ein `String`-Objekt für diesen Wert angelegt. „Verschleiern“ wir das ganze aber etwas mehr wie durch das Zeichenweise Kopieren des Strings, reichen die Optimierungen nicht mehr aus, so dass zwei verschiedene `String`-Objekte mit dem gleichen Wert angelegt werden.

Aufgabe 8.3 (P) Stundenplan-Modellierung

Verwenden Sie für diese Aufgabe nur die erlaubten Java-Methoden (siehe Anhang).

In dieser Aufgabe wollen wir einen Stundenplan, in dem Termine eingefügt und wieder gelöscht werden können, simulieren. Ein Termin beginnt an einem Wochentag zu einer bestimmten Uhrzeit (Stunde und Minute) und hat eine bestimmte Dauer (in Minuten). Ein Termin ist also nur an einen Wochentag, nicht an ein konkretes Datum gebunden. Der Stundenplan enthält Termine für eine (generelle) Woche (Montag bis Sonntag).

Schreiben Sie eine öffentliche Klasse `Date`, die die folgenden Objektvariablen und Methoden enthält.

Date
- weekday : int - starthour : int - startmin : int - duration : int - title : String
+ Date(weekday : int, starthour : int, startmin : int, duration : int, title : String) + getWeekday() : int + getStarthour() : int + getStartmin() : int + getDuration() : int + getTitle() : String + toString() : String

Der Konstruktor soll die Objektvariablen entsprechend der Parameter initialisieren. Überlegen Sie sich eine geeignete Abbildung von Wochentagen (Montag bis Sonntag) auf `Integer` und setzen diese um. Sie können davon ausgehen, dass nur gültige Parameter im Konstruktor übergeben werden. Das bedeutet jedoch nicht, dass ein Termin nicht länger als Sonntag Mitternacht oder länger als eine Woche gehen kann. Die `get`'er Methoden sollen die korrespondierenden Objektvariablen zurückliefern. Die `toString()` Methode soll eine `String`-Repräsentation des Objekts, die alle Objektvariablen enthält, zurückliefern. In dieser `String`-Repräsentation sollen die Wochentage im Wortlaut angegeben werden

und Startzeit und Dauer mit der entsprechenden Einheit (Tage, Stunden, Minuten) angegeben werden. Die genaue Formatierung kann frei gewählt werden.

Schreiben Sie eine öffentliche Klasse `Timetable`, die die folgenden Objektvariablen und Methoden enthält.

Timetable
- dates : DateList
+ Timetable() + addDate(newDate : Date) : boolean + deleteDate(date : Date) : boolean + toString() : String

Schreiben Sie dafür zunächst eine private *innere Klasse* `DateList` mit den folgenden Objektvariablen und Methoden, die eine Liste `Date`-Elementen repräsentiert.

DateList
- info : Date - next : DateList
+ DateList(info : Date) + toString() : String

Der Konstruktor soll die Membervariable `info` mit dem Paramter `info` initialisieren. Die Methode `toString()` soll eine Repräsentation der `Date`-Elemente in der Reihenfolge, wie sie in der Liste enthalten sind, zurückliefern. Sie dürfen in der inneren Klasse `DateList` weitere Methoden hinzufügen, jedoch keine weiteren Membervariablen.

Die Methode `boolean addDate(Date newDate)` der Klasse `Timetable` fügt `newDate` zur Liste `dates` (Objektvariable) hinzu, falls es keine zeitlichen Übereinschneidungen mit bereits vorhanden Terminen in `dates` gibt. Die Methode liefert `true` zurück, wenn `newDate` hinzugefügt werden konnte; andernfalls wird `dates` nicht verändert und `false` zurückgeliefert.

Die Methode `boolean deleteDate(Date date)` der Klasse `Timetable` überprüft, ob es ein `Date`-Element in der Liste `dates` (Objektvariable) gibt, bei dem die Werte der Objektvariablen mit denen des Parameters `date` übereinstimmen. Falls ein solches `Date`-Element gefunden wird, wird dieses aus `dates` entfernt und `true` zurückgeliefert; andernfalls wird `dates` nicht verändert und `false` zurückgeliefert.

Die Methode `toString()` der Klasse `Timetable` liefert eine `String`-Repräsentation der Objektvariablen `dates`, die die Repräsentation der `Date`-Elemente in chronologischer Reihenfolge enthält. Die genaue Formatierung kann frei gewählt werden.

Überlegen Sie in Gruppen oder im gesamten Tutorium, wie mit Terminen, die länger als Sonntag Mitternacht gehen, umgegangen werden kann und setzen Sie mindestens eine dieser Lösungen um.

Lösungsvorschlag 8.3

Im Lösungsvorschlag werden zwei Versionen zum Umgang mit Terminen, die länger als Sonntag Mitternacht dauern, vorgeschlagen.

1. Die Termine werden abgelehnt, also einfach nicht eingetragen.
2. Die Termine werden aufgespalten. Man nimmt an, dass der Termin am Montag fortgeführt wird. Der Termin wird dann in zwei Termine geteilt und eingetragen, falls beide Termine eingetragen werden können. Dies erfordert, dass auch beim Löschen von Terminen überprüft werden muss, ob der Termin ggf. als zwei Termine eingetragen sind. Bei der aktuellen Modellierung kann jedoch nicht mehr unterschieden werden, ob ein solcher Termin, der in `Timetable` durch zwei `Date`-Elemente repräsentiert wird, als ein Termin oder als zwei Termine eingetragen wurde. Es kann also auch nur ein Teil (der bis Sonntag Mitternacht oder der ab Montag) gelöscht werden.

Aufgabe 8.4 (P) Queue via Array

Verwenden Sie für diese Aufgabe nur die erlaubten Java-Methoden (siehe Anhang).

In dieser Aufgabe soll eine Queue durch ein Array simuliert werden.

Erstellen Sie aufbauend auf dem mitgelieferten Programmgerüst `Queue.java` eine öffentliche Klasse `Queue` mit den folgenden Objektvariablen und Methoden.

Queue
- first : int - last : int - arr : int[]
+ Queue() + isEmpty() : boolean + enqueue(x : int) + dequeue() : int + toString() : String

Der Index `first` gibt an, an welcher Position das erste Element der Queue im Array `arr` steht, der Index `last` gibt an, an welcher Position das letzte Element der Queue im Array `arr` steht.

- Für den Fall, dass `first` und `last` den Wert -1 haben, ist die Queue leer.
- Für den Fall, dass `first` $<$ `last` ist, sind die Elemente der Queue `arr[first]` bis `arr[last]`.
- Für den Fall, dass `first` $>$ `last` ist, sind die Elemente der Queue `arr[first]` bis `arr[arr.length-1]` sowie `arr[0]` bis `arr[last]` (in der angegebenen Reihenfolge).
- Für den Fall, dass `first` $==$ `last` ist, enthält die einelementige Queue das Element `arr[first]`.

Der Konstruktor `Queue()` initialisiert die Objektvariablen für eine leere Queue entsprechend der obigen Regeln. Die Objektvariable `arr` wird mit einem Array der Länge zwei initialisiert.

Implementieren Sie die folgenden Methoden.

- Die Methode `isEmpty()` liefert `true` zurück, falls die Queue leer ist; andernfalls wird `false` zurückgegeben.
- Die Methode `enqueue(int x)` fügt den Wert `x` ans Ende der Queue an, d.h. an Position `(last + 1) % arr.length` falls die Länge von `arr` größer als die in der Queue enthaltenen Elemente ist. Entspricht die Länge von `arr` der Anzahl der in der Queue enthaltenen Elemente, wird `arr` ein neues Array zugewiesen, das doppelt so groß ist wie `arr` bisher und die Elemente der Queue in der entsprechenden Reihenfolge ab Index 0 enthält.
- Die Methode `dequeue()` entfernt das erste Element aus der Queue, d. h. `first` wird wie folgt aktualisiert `first = (first + 1) % arr.length`. Ist die Anzahl der nun in der Queue enthaltenen Elemente kleiner oder gleich `arr.length/4`, wird `arr` ein neues Array zugewiesen, das nur halb so groß ist wie `arr` bisher und die Elemente der Queue in der entsprechenden Reihenfolge ab Index 0 enthält.

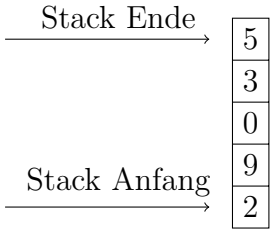
Machen Sie sich vor der Implementierung in Gruppen oder dem ganzen Tutorium mit der Repräsentation der Datenstrukturen und den Operationen vertraut. Veranschaulichen Sie sich die Operationen `enqueue(x)` und `dequeue()` anhand verschiedener Beispiele und betrachten Sie vor allem Randfälle, z.B. `last` entspricht dem letzten Index im Array und es wird `enqueue(x)` aufgerufen usw.

Aufgabe 8.5 (H) Symmetrie ist alles [10 Punkte]

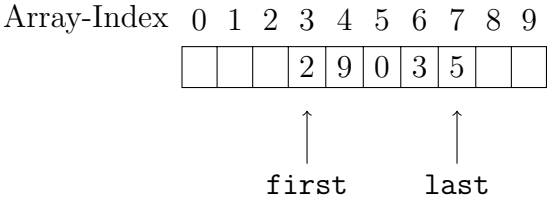
Verwenden Sie für diese Aufgabe nur die erlaubten Java-Methoden (siehe Anhang).

Ein *symmetrischer Stack* ist ein Stapel, auf den nicht nur „oben“ etwas hinzugefügt und entnommen werden kann (wie aus der Vorlesung bekannt), sondern auch „unten“ etwas hinzugefügt bzw. entnommen werden kann. Im Folgenden wollen wir solch einen symmetrischen Stack implementieren. Dabei sollen die Elemente des Stacks in einem Array gespeichert werden. Die Elemente selber sind vom Typen `int` und somit das Array vom Typen `int[]`. Die `int`-Variable `first` gibt den Index des Arrays an, der das Element, welches das „unterste“ Element in dem zu repräsentierenden symmetrischen Stack, enthält. Ähnlich verwenden wir eine `int`-Variable `last` für den Index im Array, welcher das „oberste“ Element in dem zu repräsentierenden symmetrischen Stack enthält.

Beispiel. Betrachten Sie den folgenden symmetrischen Stack mit 5 Elementen:

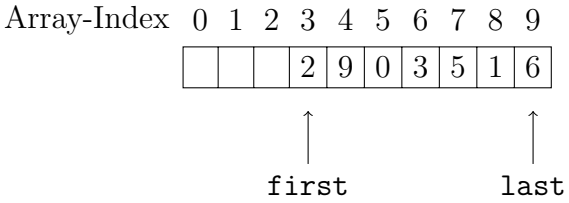


Dieser kann wie folgt als Array z.B. der Länge 10 repräsentiert werden:

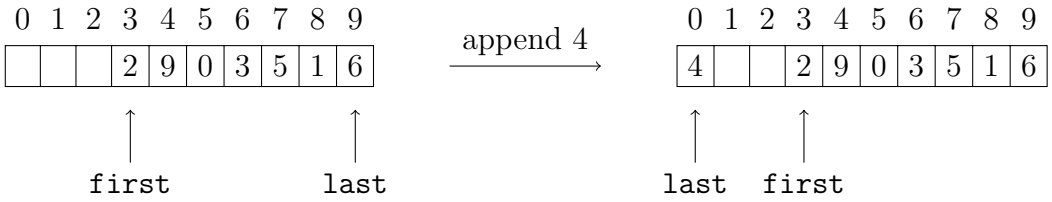


In unserer Repräsentation gibt `first` den Index für das unterste Element und analog `last` den Index für das oberste Element eines korrespondierenden symmetrischen Stacks an. Die Array-Einträge, die nicht zwischen `first` und `last` liegen (in diesem Beispiel also die Einträge mit den Indizes $0, 1, 2, 8, 9$), sind als freie Einträge zu interpretieren. D.h. soll ein weiteres Element an das Ende des Stacks eingefügt werden, dann würde in diesem Beispiel das Element in dem Array an der Stelle 8 eingefügt werden. Analog, wiederum ein weiteres Element an der Stelle 9.

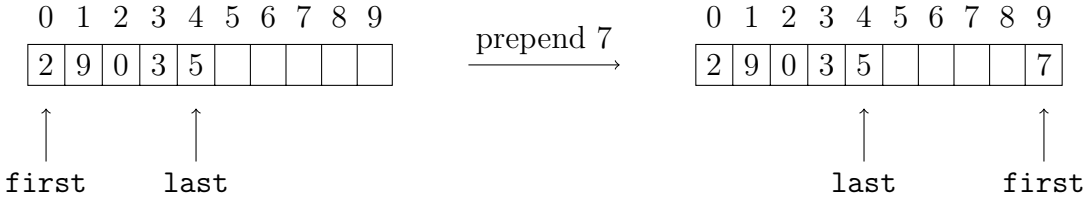
Betrachten Sie nun folgende Beispielkonfiguration:



Der Index `last` gibt das letzte Element in dem Array an. Wollen wir nun noch ein weiteres Element „oben“ auf den Stack legen, dann stellen wir fest, dass wir kein weiteres Element an das Ende des Arrays einfügen können. Anstatt jetzt einfach ein neues Array anzulegen und Elemente zu kopieren, soll unsere Implementierung die noch freien Einträge am Anfang des Arrays nutzen (ähnlich zu einem Array mit zyklischem Zugriff). D.h. wollen wir in diesem Beispiel das Element 4 oben auf den symmetrischen Stack legen, dann sieht das Array danach wie folgt aus:



Analog ist das Problem zu betrachten, wenn ein Element „unten“ an den Stack eingefügt werden soll:

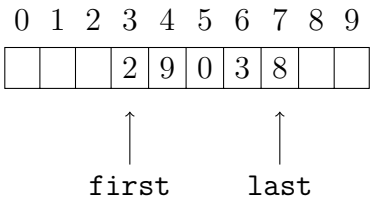


Implementieren Sie einen symmetrischen Stack in der Datei “`SymmetricStack.java`”. Verwenden Sie dazu das gegebene Code-Grundgerüst.

Im Folgenden verwenden wir den Begriff *belegt* für einen Array-Eintrag, der ein Element des zu repräsentierenden Stacks enthält. Ansonsten sprechen wir wie bisher von einem freien Eintrag. In den letzten Beispielen war der Array-Eintrag 4 jeweils belegt.

Der parameterlose Konstruktor soll die Objektvariable `data` mit einem nicht leeren `int`-Array, also einem Array der Länge größer Null, initialisieren und die Objektvariablen `first` und `last` auf `-1` setzen. D.h. ein leerer Stack wird daran identifiziert, dass die Objektvariable `first` den Wert `-1` hat. Implementieren Sie nun die folgenden Methoden:

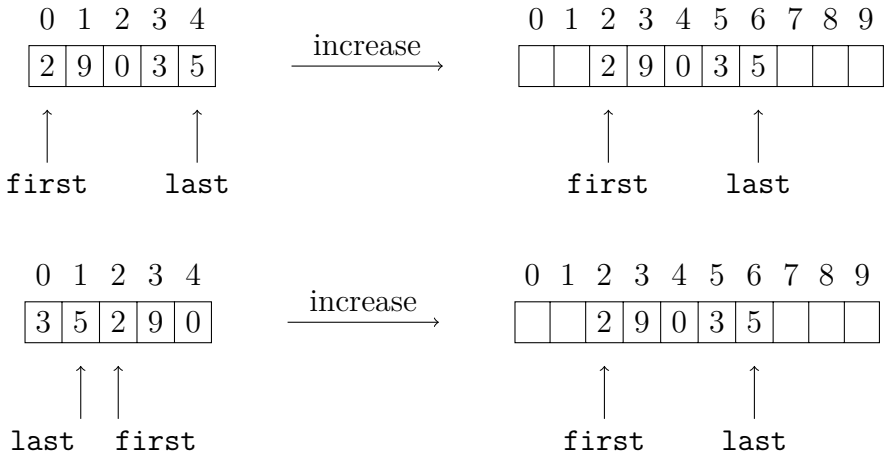
- `public int getNumberOfElements()`
  Gibt die Anzahl der Elemente im Stack zurück, der duch das Array `data` repräsentiert wird. Beispiel: Für folgende Konfiguration liefert die Methode 5 zurück.



- `public boolean isEmpty()`
  Gibt `true` zurück, wenn der Stack, der durch das Array `data` repräsentiert wird, leer ist und andernfalls `false`.

- `public boolean isFull()`
  Gibt `true` zurück, wenn das Array `data` keine freien Einträge hat, also alle Array-Einträge belegt sind, und andernfalls `false`.

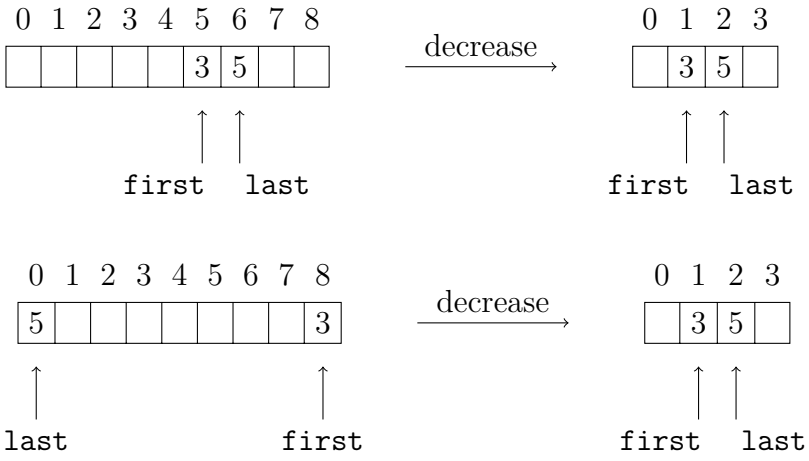
- `public void increase()`
  Die Methode `increase` verdoppelt die Größe des Arrays `data`, sofern alle Array-Einträge belegt sind. Andernfalls verändert sie das Objekt nicht. In dem neuen Array müssen alle alten Einträge enthalten sein, die in dem zu repräsentierenden symmetrischen Stack vorkommen. Außerdem muss die Reihenfolge der Einträge die selbe sein. Wenn die Länge des alten Arrays n war, dann enthält der Array-Eintrag `data[n/2]` das „unterste“ Elemente des Stacks. (Die weiteren Elemente folgen konsekutiv in den darauffolgenden Array-Einträgen.) Ist n ungerade, entspricht `n/2` also $\lfloor n/2 \rfloor$. Intuitiv bedeutet das, dass alle Einträge aus dem alten Array in dem neuen Array „zentriert“ auftauchen. Sprich nach dem Aufruf von `increase` hat das Array `data` „vorne“ als auch „hinten“ freien Platz um weitere Elemente zu speichern. Beispiele:



  Beachten Sie, dass in letzterem Fall, die Element „normalisiert“ in dem neuen Array auftauchen.

- `public void decrease()`
  Die Methode `decrease` verkleinert die Größe des Arrays `data` auf $\lfloor n/2 \rfloor$, sofern kleiner oder gleich $\lfloor n/4 \rfloor$ viele Einträge belegt sind, wobei n die Größe des Arrays

ist. Analog zur Methode `increase` muss die Methode `decrease` die Reihenfolge der im Stack enthaltenen Elemente erhalten, wobei das „unterste“ Element in dem neuen Array an dem Index $\lfloor n/8 \rfloor$ liegen muss. Beispiele:



Beachten Sie, dass auch in letzterem Fall, die Elemente „normalisiert“ in dem neuen Array auftauchen.

- `public void prepend(int x)`
  Die Methode `prepend` fügt unten am Stack das Element `x` hinzu. Falls der Stack leer ist, dann wird das Element `x` an dem Index $\lfloor n/2 \rfloor$ in das Array `data` hinzugefügt. Falls das Array `data` voll ist, dann wird dieses mit der Methode `increase` vergrößert.

- `public void append(int x)`
  Die Methode `append` fügt oben auf den Stack das Element `x` hinzu. Falls der Stack leer ist, dann wird das Element `x` an dem Index $\lfloor n/2 \rfloor$ in das Array `data` hinzugefügt. Falls das Array `data` voll ist, dann wird dieses mit der Methode `increase` vergrößert.

- `public void removeFirst()`
  Die Methode `removeFirst` entfernt das unterste Element aus dem Stack. Des Weiteren soll sichergestellt werden, dass das Array `data` mit der Methode `decrease` verkleinert wird, wann immer nötig.

- `public void removeLast()`
  Die Methode `removeLast` entfernt das oberste Element aus dem Stack. Des Weiteren soll sichergestellt werden, dass das Array `data` mit der der Methode `decrease` verkleinert wird, wann immer nötig.

Hinweis: In Java ist die Division x/y zweier ganzer Zahlen x und y gleich $\lfloor x/y \rfloor$. Beispiel: Nach der Zuweisung `int x = 5/2;` enthält die Variable `x` den Wert `2`.

Lösungsvorschlag 8.5

siehe `SymmetricStackSolution.java`

Korrekturbemerkung:

- getNumberOfElements: Für folgende Fälle gibt es jeweils einen halben Punkt:

  1. Leerer Stack

2. Weder leerer noch voller Stack und `first` $<$ `last`
3. Voller Stack mit `first` $<$ `last`
4. Voller Stack mit `last` $<$ `first`

- prepend: Für folgende Fälle gibt es jeweils einen halben Punkt:

  1. Wenn $0 <$ `first` $<=$ `last` $<$ `data.length` gilt
  2. Wenn $0 =$ `first` $<=$ `last` $<$ `data.length` $- 1$ gilt
  3. Wenn `last` $<$ `first` gilt

  D.h. in beiden Fällen gibt es noch freie Einträge.

- append: Für folgende Fälle gibt es jeweils einen halben Punkt:

  1. Wenn $0 <$ `first` $<=$ `last` $<$ `data.length` $- 1$ gilt
  2. Wenn $0 <$ `first` $<=$ `last` $=$ `data.length` $- 1$ gilt
  3. Wenn $0 <=$ `last` $<$ `first` gilt

  D.h. in beiden Fällen gibt es noch freie Einträge.

- removeFirst: Für folgende Fälle gibt es jeweils einen halben Punkt:

  1. `first` $<$ `last`
  2. `last` $<$ `first`
  3. `first` $=$ `last`

- removeLast: Für folgende Fälle gibt es jeweils einen halben Punkt:

  1. `first` $<$ `last`
  2. `last` $<$ `first`
  3. `first` $=$ `last`

- increase: Für folgende Fälle gibt es jeweils einen halben Punkt:

  1. Wenn das Array voll ist und `first` $<$ `last` gilt
  2. Wenn das Array voll ist un `last` $<$ `first` gilt

- decrease: Für folgende Fälle gibt es jeweils einen halben Punkt:

  1. Wenn der Stack weniger als $\lfloor n/4 \rfloor$ Elemente hat und `first` $<$ `last` gilt
  2. Wenn der Stack weniger als $\lfloor n/4 \rfloor$ Elemente hat und `last` $<$ `first` gilt

Für "off-by-one-error" (deutsch etwa Um-Eins-daneben-Fehler) gibt es auf die gesamte Aufgabe einen Punkt abzug. Ein "off-by-one-error" ist z.B. wenn anstelle des Index i der Index $i \pm 1$ genommen wird.

Aufgabe 8.6 (H) Bankengeheimnis [9 Punkte]

Verwenden Sie für diese Aufgabe nur die erlaubten Java-Methoden (siehe Anhang)!

Im Folgenden wollen wir eine Bankensimulation schreiben. Schreiben Sie eine öffentliche Klasse `Money` im Paket `pgdp` um Geldbeträge darzustellen. Die Klasse muss folgende Membervariable und Methoden bereitstellen:

Money
- cent : int
+ Money() + Money(cent : int) + getCent() : int + addMoney(m : Money) : Money + toString() : String

Die Klasse hat eine private Objektvariable `cent` vom Typen `int` welche einen Euro-Betrag in Cent speichert (die Bankenaufsicht hat von uns gefordert, keine Floatingpoint-Typen zu verwenden, um mögliche Floatingpoint-Fehler auszuschließen). Des Weiteren muss die Klasse einen Konstruktor `Money()` haben, welcher die Variable `cent` auf Null setzt. Ein weiterer Konstruktor `Money(int cent)` soll die Objektvariable `cent` auf den Wert der Argumentvariable `cent` setzen. Die Methode `getCent()` liefert den Wert der Objektvariable `cent` zurück. Die Methode `addMoney(Money m)` erzeugt ein *neues* Objekt `Money` und liefert dieses zurück, in dem die Cent-Beträge `this.cent` und `m.cent` addiert sind. Zum Beispiel: `new Money(101).addMoney(new Money(-100))` liefert ein Objekt, welches äquivalent zu `new Money(1)` ist. Die Methode `toString()` liefert eine Stringrepräsentation des Objekts zurück. Der Cent-Betrag soll als Euro-Betrag ausgegeben werden. Dabei müssen immer zwei Nachkommastellen ausgegeben werden und mindestens eine Vorkommastelle. Als Dezimalstelle dient ein Komma. Des Weiteren muss der Suffix "Euro" lauten, der mit genau einem Leerzeichen auf die Zahl folgt. Das Format ist also folgendes

<Eine-oder-mehr-Vorkommastellen>,<1te-Nachkommastelle><2te-Nachkommastelle> Euro

Beispiel: `new Money(10010).toString()` gleicht dem String `"100,10 Euro"`.

Keine Methode der Klasse `Money` darf das Objekt verändern (Javasprech: "Immutable-Object").

Implementieren Sie nun eine öffentliche Klasse `BankAccount` im Paket `pgdp` mit den folgenden Membervariablen und Methoden:

BankAccount
- bankaccount : int - firstname : String - surname : String - balance : Money
+ BankAccount(accountnumber : int, fname : String, sname : String) + getAccountnumber() : int + getFirstname() : String + getSurname() : String + getBalance() : Money + addMoney(m : Money) : void + toString() : String

Die `get`'er Methoden sollen die korrespondierenden Objektvariablen zurückliefern. Die Methode `addMoney(Money m)` addiert das Geld `m` auf den Kontostand `balance` des jeweiligen Objekts. Die Methode `toString` liefert einen String zurück, der alle Werte aller Objektvariablen wiederspiegelt (die exakte Formatierung ist irrelevant). Der Konstruktor initialisiert alle Objektvariablen anhand der Parameter sowie die Objektvariable `balance` auf ein `Money`-Objekt welches Null Cent repräsentiert.

Als nächstes soll die öffentliche Klasse `Bank` im Paket `pgdp` wie folgt implementiert werden:

Bank
- accounts : BankAccountList
+ newAccount(firstname : String, lastname : String) : int + removeAccount(accountnumber : int) : void + getBalance(accountnumber : int) : Money + depositOrWithdraw(accountnumber : int, m : Money) : boolean + transfer(from : int, to : int, m : Money) : boolean

In einer Liste mit dem Typen `BankAccountList` wollen wir alle Kunden der Bank auflisten und in der Objektvariable `accounts` der Klasse `Bank` speichern. Implementieren Sie hierfür eine private *innere Klasse* `BankAccountList` in der Klasse `Bank` wie folgt:

BankAccountList
+ info : BankAccount + next : BankAccountList

Die Methode `newAccount` legt ein neues Konto an, i.e., also ein neues Objekt vom Typen `BankAccount`. Jedes Konto erhält eine eindeutige Kontonummer. Jedes neu erzeugte Konto wird in die Liste `accounts` eingefügt. Schlussendlich liefert die Methode die Kontonummer von dem neu erzeugten Konto zurück. Das Pendant zu der Methode `newAccount` ist die Methode `removeAccount` welches aus der Liste `accounts` das Objekt mit der passenden Kontonummer löscht, falls es denn existiert. Die Methode `getBalance(int accountnumber)` liefert als Rückgabewert den Kontostand des Kontos mit der Kontonummer `accountnumber`. Falls solch ein Konto nicht in der Liste `accounts` existiert, wird `null` zurückgeliefert. Die Methode `depositOrWithdraw(int accountnumber, Money m)` addiert das Geld `m` auf das Konto mit der Kontonummer `accountnumber`. Falls solch ein Konto nicht in der Liste `accounts` existiert wird `false` zurückgeliefert und andernfalls `true`. Die Methode `transfer(int from, int to, Money m)` überweist von dem Konto mit der Kontonummer `from` auf das Konto mit der Kontonummer `to` `m` viel Geld. Sollte das Konto `to` oder `from` nicht in der Liste `accounts` existieren, dann liefert die Methode `transfer` als Rückgabewert `false` und verändert die Konten nicht. D.h. vor und nach dem Aufruf muss der Kontostand gleich sein. Andernfalls, wenn die Konten existieren, dann muss nach dem Aufruf gelten, dass das Konto `from` `m` viel Geld weniger hat und das Konto `to` `m` viel Geld mehr hat. Die Methode liefert in diesem Fall `true` zurück. Der Informatiker würde davon sprechen, dass die Methode `transfer` mit der gewünschten Eigenschaft *transaktional* ist.

Es dürfen keine weiteren Membervariablen erstellt werden, aber beliebige weitere Methoden.

Lösungsvorschlag 8.6

siehe `Bank.java`, `BankAccount.java` und `Money.java`

Korrekturbemerkung:

- `Money` is immutable: 1 Punkt
- `Money.addMoney`: 1 Punkt
- `Money.toString`: 1 Punkt
- `Bank.transfer(x, x, m)`: 1 Punkt (self-transfer)
- `Bank.transfer`: 2 Punkte (transfer succeeds and balances change)
- `Bank.transfer`: 2 Punkte (transfer fails and balances stay the same)
- `Bank.getBalance(doesnt_exist) == null`: 1 Punkt (falls das Konto nicht existiert)

Aufgabe 8.7 (H) Memoization: Ich erinnere mich, also bin ich [2 Bonuspunkte]

Spätestens seit der Aufgabe A6B6 von letzter Woche wissen wir, wie wir die Klammerung bei einem Matrizenmultiplikationsproblem finden können, die die wenigsten Multiplikationen benötigt. Wir wiederholen, bei k gegebenen Matrizen A_i, $1 \leq i \leq k$ ist die minimal benötigte Anzahl an Multiplikationen gleich $f(1, k)$, wenn

$$f(i,j) = \begin{cases} 0 & \text{if } i = j \\ \min\{\, f(i,x) + f(x+1,j) + (d_1(i) \cdot d_2(x) \cdot d_2(j)) \mid i \leq x < j \,\} & \text{if } i < j \end{cases}$$

für beliebige $i, j \in \mathbb{N}$ gilt. Lassen Sie sich für ein Matrizenmultiplikationsproblem die Aufrufparameter der Funktion f ausgeben. Wir beobachten, dass für ein $k > 3$, Funktionswerte $f(i, j)$ für einige $i, j \in \mathbb{N}$ mehrfach berechnet werden. Anstatt den Funktionswert immer und immer wieder zu berechnen, wollen wir den Wert genau einmal berechnen und dann speichern, sodass wir den gespeicherten Wert immer wieder abrufen können, wann immer dieser wieder benötigt wird. Diese Technik haben wir in der Vorlesung kennen gelernt (siehe Folien über Fibonaccizahlen) und ist unter dem Stichwort *Memoization* allgemein bekannt.

Implementieren Sie Ihre Lösung in der Datei `MatrixMultOptMemoization.java`, die eine Lösung zu der Aufgabenstellung A6B6 ist, sodass jeder Funktionswert $f(i, j)$ für $i \neq j$ genau einmal berechnet wird.

Lösungsvorschlag 8.7

siehe `MatrixMultOptMemoizationSolution.java`

Korrekturbemerkung:

- Speichern der Funktionswerte: 1 Punkt
- Abrufen der Funktionswerte: 1 Punkt

sodass jeder Funktionswert $f(i, j)$ für $i \neq j$ genau einmal berechnet wird.

Erlaubte Java-Methoden

MiniJava:

```
public static int readInt()
public static int readInt(String s)
public static int read()
public static int read(String s)
public static String readString()
public static String readString(String s)
public static void write(String output)
public static void write(int output)
public static int drawCard()
public static int dice()
```

returns an integer from the interval $[2, 11]$

returns an integer from the interval $[1, 6]$

String:

`public char charAt(int index)`
Returns the char value at the specified index. An index ranges from `0` to `length() - 1`. The first char value of the sequence is at index `0`, the next at index `1`, and so on, as for array indexing.

Example: `String s = "Hello Students"; char c = s.charAt(7);` saves the character `'t'` in the variable `c`.

`public boolean isEmpty()`
Returns `true` if, and only if, `length()` is `0`.

`public int length()`
Returns the length of this string. The length is equal to the number of Unicode code units in the string.

Example: `String s = "Hello Students"; int l = s.length();` saves the value `14` in the variable `l`.

`public boolean equals(Object obj)`
Compares this string to the specified object. The result is true if and only if the argument is not null and is a String object that represents the same sequence of characters as this object.
Example: `String s = "Hello Students"; boolean v = s.equals("Hello Students");` here `v` is evaluated to `true`.

System:

`System.out.print(x)` prints the object `x` to the console
`System.out.println(x)` prints the object `x` to the console and terminates the line

Selbstgeschriebene Methoden, die selber nur erlaubte Methoden verwenden, sind erlaubt, sofern dies nicht explizit in der Aufgabenstellung verboten wurde. Methoden, die im gegebenen Programmgerüst definiert wurden, dürfen verwendet werden.