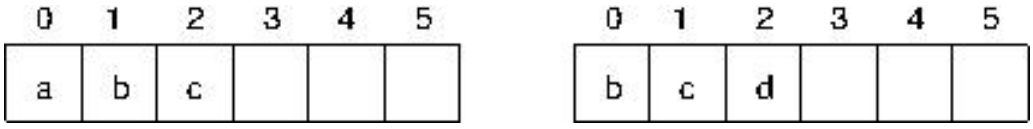
1. Praktikum "Algorithmen und Programmierung II"

Die Klassen der 1. Praktikumsaufgabe

`util.Queue`

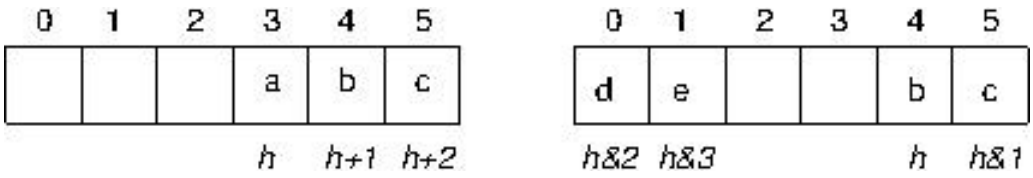
Die Klasse hat die Aufgabe Daten zu speichern und bei Bedarf in der Reihenfolge zurückzugeben, in der diese gespeichert wurden (Methoden `put` und `get`). Zusätzlich verfügt sie über eine Reihe von Methoden, mit denen man ihren Zustand abfragen kann. Die Methode `reduceCapacitiy` hat eine Sonderstellung: Sie sorgt dafür, dass die Queue nicht mehr Speicherplatz benötigt als nötig (aus Performancegründen, wird dies nicht dauernd nachgehalten).

Implementierung: Innerhalb der Klasse werden die Daten in dem Array `data` gespeichert. Das Array ist mindestens so groß wie die Anzahl der gespeicherten Daten. Diese Anzahl wird in der Variablen `size` festgehalten. Für die Größe des Arrays (im Beispiel 6) verwende ich hier den Buchstanben *N*. Die folgende Abbildung zeigt eine Lösungsvariante:



Nach dieser Idee werden neue Daten (wie beim Stack) in aufsteigenden Array-Plätzen gespeichert. Queue-Elemente sollen in einer First In First Out (FIFO) Reihenfolge entnommen werden. Auf der rechten Seite ist ein Element (a) entnommen worden und ein neues (d) hinzugekommen. Diese Lösung hat den großen Nachteil, dass große Datenmengen bewegt werden müssen.

Es geht besser, wie die folgende Abbildung zeigt:



Zunächst einmal werden die Daten, wie bisher, auf fortlaufenden Plätzen gespeichert. Bein Entnehmen eines Elements „rücken“ die nachfolgenden Elemente aber nicht auf. Statt dessen merkt man sich nur den Anfang des gültigen Bereichs (hier die Variable *h;* im Programm `head`). Als nächstes wird das „a“ von Platz 3 entnommen. Der Anfang verschiebt sich dann auf die Stelle $h + 1$.

Diese Taktik ist effizient, zwingt aber zu der Überlegung, wie man die frei gewordenen Plätze am Anfang des Arrays nutzen kann. Auf der rechten Seite ist die Lösung dargestellt: Wenn man am rechten Rand (N-1) angelangt ist, fängt man wieder bei 0 an. Dieses zyklische Hochzählen habe ich durch $h \,\&\, i := (h + i) \,\%\, N$ dargestellt. Der nächste freie Platz ist in dem Beispiel rechts gleich $(h + size) \,\%\, N = (4 + 4) \,\%\, 6 = 2$.

Das Erhöhen von `h` (im Programm `head`) erfolgt natürlich ebenfalls mit der Modulo-Formel.

Wir verschieben nicht mehr die Daten, sondern nur unser „Bezugssystem“.

Den gültigen Bereich kennen wir durch die Variable `size`. Für den Garbage-Collector werden die nicht genutzten Plätze mit **`null`** belegt.

Die sogenannte „Restklassenarithmetik“ wird nicht nur bei Datenstrukturen verwendet. Sie ist ein zentrales Element in Verschlüsselungsalgorithmen!

`util.QueueTest`

Die Methoden der Klasse stellen einfache Testfälle dar. Ihr Aufruf geschieht durch das Framework JUnit. Für jede der Testmethoden meldet JUnit den erstem auftretenden Fehler.

Beim Testen sollten Sie systematisch vorgehen. Einige Fehler sind Folgefehler. Daher sollten Sie mehr oder weniger der Reihenfolge der Tests in den Testquellen folgen und die einfachsten Probleme zuerst betrachten (die Reihenfolge der JUnit-Ausgabe weicht hiervon leider ab).

`statistics.Generator`

Diese Klasse erzeugt die Testdatei `input.txt`. Sie können den Inhalt und die Größe der Textdatei beeinflussen. Die Kasse muss vor `statistics.Main` ausgeführt werden!

`statistics.Main`

Diese Klasse veranlasst mithilfe der Klasse `statistics.Statistics` die statistische Auswertung der Daten in der Datei `input.txt`.

`statistics.Statistics / statistics.StatisticsTest`

Diese Klasse `Statistics` liest die Testdaten ein und enthält die Statistikfunktionen. Zwei kleinere Funktionen sollen von Ihnen ergänzt werden.

Hinweis: Die Queue und das Array `data` speichern die Daten unter dem Datentyp `Object`. In Wirklichkeit handelt es sich bei allen eingelesenen Daten aber um Objekte der Klasse `Double`. Dies muss beim Verwenden der aus der Queue entnommen Objekte dann auch angegeben werden. Der Cast `(Double)` stellt gewissermaßen die „verloren gegangene“ Typdeklaration wieder her.

Grundsätzlich ist die Verdopplung `Double` – `double` ein früher Entwurfsfehler von Java. Da in Java Zahlen keine Objekte sind, ist es nötig Zahlen in Objekten zu „verpacken“, wenn sie in Behältern für beliebige Objekte gespeichert werden sollen. Anschließend muss man sie dann wieder „auspacken“. Dies kann manuell geschehen:

```
q.put(Double.valueOf(3.14));
Double dx = (Double) q.get();
double x = dx.doubleValue();
```

oder in vereinfachter Schreibweise (Autoboxing) angegeben werden:

```
q.put(3.14);
double x = (Double) q.get();.
```