|  |
| --- |
| Dokumentation |
| Zu Aufgabenblatt 02 aus der Vorlesungsreihe „Algorithmen und Datenstrukturen“ |
|  |
|  |
|  |

Dokumentation

Zu Aufgabenblatt 02 aus der Vorlesungsreihe „Algorithmen und Datenstrukturen“

# Aufgabe 1

Das entsprechende Interface, das die geforderten Funktionen bereitstellt, heißt „QuadMatrix“. Im Anhang dieses Dokumentes ist die vollständige JavaDoc zu unserem entwickelten Projekt.

Es stellt außerdem folgende Funktionen bereit:

**getSize()** gibt die Länge einer Seite der Matrix zurück. Da es eine quadratische Matrix ist, lässt sich daraus auch wieder die maximale Anzahl der möglichen Werte berechnen lässt.

**init()** erstellt eine neue Instanz einer gegebenen Matrix-Implementation. Da es drei verschiedene Implementationen gibt, müssen wir den Implementationstyp der Ausgabe bei bspw. einer Addition auf den Implementationstyp der Eingabe anpassen. Dies wird durch **init()** ermöglicht.

**equals()** ermöglicht den Vergleich von Matrizen. Dies ist für JUnit-Tests erforderlich, und wird bei der Verwendung der Matrizen von Nutzen sein.

**space()** gibt den verwendeten Platz der einzelnen Instanz wieder. Diese Funktion wird zur Auswertung und Vergleich der einzelnen Implementationen genutzt.

**time()** gibt die bislang benötigte Zeit der einzelnen Instanz wieder. Diese Funktion wird zur Auswertung und Vergleich der einzelnen Implementationen genutzt.

**timeReset()** setzt den Zeitzähler einer Instanz auf 0 zurück. Wird eher zu Testzwecken nützlich sein.

**setGen(int m, int n, double x)** stellt eine Methode bereit, um Matrizen schnell zu befüllen. Diese Methode ist ausdrücklich nicht zum generellen Gebrauch zu nutzen. Sie setzt voraus, dass man keine vorhandenen Werte mit einer 0 überschreibt und wird genutzt, um später die nötigen Testwerte berechnen zu können.

# Aufgabe 7

## Teilaufgabe b)

**Experiment:** Erstellen Sie *t* Matrizen mit einer Größe von n x n (n sei die letzten 4 Stellen einer unserer Matrikelnummern). Es soll der durchschnittliche Platzverbrauch der jeweiligen Implementation berechnet werden. *t* soll dabei so dimensioniert werden, dass die ersten 5 Stellen der Werte „stabil“ sind.

**Erwartung:** Der Platzverbrauch der Implementation (iii) sollte deutlich höher sein, da jedes Element in 3er Tupeln gespeichert wird = [Spalte,Zeile,Wert].

**Ergebnis:** Die Auswertung der Ergebnisse ergab folgendes Diagramm, das zeigt, dass die Implementation (iii) einen deutlich höheren Platzverbrauch aufweist. Das Ergebnis entspricht der Erwartung, dass der Platzverbrauch der Implementation (ii) zu 2/3 dem Platzverbrauch der Implementation (iii) entspricht.

## Teilaufgabe c)

QuadMatrixArrayImpl – Implementation (i) : space(p,n) = n2 , da hier das Array in jedem Fall mit Werten instanziert wird.

QuadMatrixArrayListImpl – Implementation (ii) : space(p,n) = n\*p\*n\*2 , da hier 2er Tupel (Spaltenindex und der zugehörige Wert) gespeichert werden, bei denen der Wert nicht 0 ist.

QuadMatrixListImpl – Implementation (iii) : space(p,n) = n²\*p\*3, da hier 3er Tupel (Zeilenindex, Spaltenindex und der zugehörige Wert) gespeichert werden, bei denen der Wert nicht 0 ist.

## Teilaufgabe d)

Implementation (i) : Der Platz-Overhead beträgt genau n2 , da das Array bereits mit 0’en vorbefüllt wird.

Implementation (ii) und (iii) : Der Platz-Overhead beträgt 0, da die Implementationen nicht mit Werten vorbefüllt werden. Sie sind von Anfang an leer.

# Aufgabe 8

## Teilaufgabe b)

**Experiment:** Gleicher Experimentaufbau wie bei Aufgabe 7b, nur dass in diesem Experiment der Zeitverbrauch der einzelnen Implementationen gemessen wird.

**Erwartung:** Dieses Experiment sollte zeigen, dass die Implementation (i) im Verhältnis am schnellsten ist. Denn (ii) und (iii) müssen jedes Mal prüfen, ob der einzufügende Wert ein 0 ist und ob es an der einzufügenden Stelle schon etwas gespeichert ist. Dabei muss der Algorithmus die Stelle erst finden, und wenn sie vorhanden ist, dann löschen. Dies benötigt sehr viel Rechenzeit (je nachdem wie groß die Matrix letztendlich wird).

**Ergebnis:** Im folgenden sind die Diagramme der Durchschnittsmesswerte und der Streuung der Messwerte zu sehen. Sie zeigen, dass auch hier die Implementationen (ii) und (iii) sehr langsam sind im Verhältnis zu (i).

Bei ersten Versuchen die Messreihe für (iii) zu erstellen, wurden die Berechnungszeiten derartig lang, dass in 9h nur für t=2000 ein Messwert erstellt werden konnte. Nach Erstellung und Nutzung der Methode **setGen** im Zufallsmatrixgenerator konnten die Messwerte nun in deutlich weniger Zeit berechnet werden.

## Teilaufgabe c)