Team: 5, Marc Kaepke, Constantin Wahl

Aufgabenaufteilung:

1. Gemeinsame Erarbeitung der Skizze

Quellenangaben: Vorlesungsskript

Bearbeitungszeitraum:

Gemeinsam:

18.11.15 => 2,5 Stunden

Aktueller Stand: Skizze angefertigt und mit Praktikumsgruppe geteilt

Änderungen in der Skizze: zusätzliches Package für ADTs, neue Klasse NumGenerator und jeder Algorithmus in eigener Klasse

Skizze:

Packagestruktur (unterhalb vom src-Package):

adt.implementations.AdtArrayImpl

adt.implementations.AdtListImpl

adt.interfaces.AdtArray

adt.interfaces.AdtList

sort.algorithms.insertionSort

sort.algorithms.quickSort

sort.algorithms.insertionSortRuntime

sort.algorithms.quickSortRuntime

sort.algorithms.insertionSortAccessCount

sort.algorithms.quickSortAccessCount

sort.MethodPivot

sort.NumGenerator

sort.tests.insertionSortTests

sort.tests.quickSortTests

Klassendefinition:

AdtArrayImpl

Die Implementierung des ADTArray vom ersten Praktikum.

AdtListImpl

Die Implementierung des ADTList vom ersten Praktikum.

AdtArray

Das Interface des ADTArray vom ersten Praktikum.

AdtList

Das Interface des ADTList vom ersten Praktikum.

NumGenerator

- public NumGenerator()
  - Konstruktor (default), um keine static-Methoden verwenden zu müssen
- public void sortNum(String filename, int quantity)
  - erzeugt eine zufällige Zahlenreihe und speichert diese als Datei ab
- public void sortNum(String filename, int quantity, boolean desc)
  - desc = true => erzeugt eine zufällige Zahlenreihe, absteigend sortiert
  - desc = false => erzeugt eine zufällige Zahlenreihe, aufsteigend sortiert
  - speichert die Zahlenreihe als Datei ab
- public AdtArray readNum(String filename)
  - liest die Datei ein und speichert sie in einem ADTArray Objekt

insertionSort

- public insertionSort
  - Konstruktor (default), um keine static-Methoden verwenden zu müssen
- public AdtArray insertionSort(AdtArray numbers, int startIndex, int endIndex)
  - startIndex und endIndex geben den Index-Bereich im AdtArray numbers der aufsteigend (größer werdend) sortiert werden soll

quickSort

- public quickSort
  - Konstruktor (default), um keine static-Methoden verwenden zu müssen
- public AdtArray quickSort(AdtArray numbers, MethodPivot pivot)
  - Sind weniger als 12 Elemente (innerhalb der Rekursion) zu sortieren, ist insertionSort zu verwenden, d.h. ab einer Anzahl von 12 Elementen arbeitet das Programm rekursiv und verzweigt bei weniger als 12 Elementen (für genau diese Elemente!) auf insertionSort
    - ⇨ Wenn eine Teilliste im quickSort weniger als 12 Element hat, wird auf dieser Teilliste insertionSort aufgerufen

insertionSortRuntime

- public insertionSortRuntime
  - Konstruktor (default), um keine static-Methoden verwenden zu müssen
- public double getRuntime()
  - Getter-Methode um die gemessene Laufzeit zurückzugeben.

Wie insertionSort, nur um Laufzeitmessungen ergänzt. Die Algorithmen geben die Laufzeit nicht zurück, die geschieht über die Getter-Methoden.

quickSortRuntime

- public quickSortRuntime
  - Konstruktor (default), um keine static-Methoden verwenden zu müssen
- public double getRuntime()
  - Getter-Methode um die gemessene Laufzeit zurückzugeben.

Wie quickSort, nur um Laufzeitmessungen ergänzt. quickSort wird abzüglich der Zeit die insertionSort benötigt gemessen.

Die Algorithmen geben die Laufzeit nicht zurück, die geschieht über die Getter-Methoden.

insertionSortAccessCount

- public insertionSortAccessCount
  - Konstruktor (default), um keine static-Methoden verwenden zu müssen
- public int getReadCount()
  - Getter-Methode um die Lesezugriffe zurückzugeben
- public int getWriteCount()
  - Getter-Methode um die Schreibzugriffe zurückzugeben
- public int getAccessCount()
  - Getter-Methode um die gesamten Zugriffe (Lese- und Schreibzugriffe) zurückzugeben

Wie insertionSort, nur um Zugriffzählungen ergänzt. Lese- und Schreibzugriffe sind getrennt zu erfassen.

Die Algorithmen geben die Zugriffe selbst nicht zurück, dies geschieht durch die Getter-Methoden.

quickSortAccessCount

- public quickSortAccessCount
  - Konstruktor (default), um keine static-Methoden verwenden zu müssen
- public int getReadCount()
  - Getter-Methode um die Lesezugriffe zurückzugeben
- public int getWriteCount()
  - Getter-Methode um die Schreibzugriffe zurückzugeben
- public int getAccessCount()
  - Getter-Methode um die gesamten Zugriffe (Lese- und Schreibzugriffe) zurückzugeben

Wie quickSort, nur um Zugriffzählungen ergänzt. Lese- und Schreibzugriffe sind getrennt zu erfassen. quickSort wird abzüglich der Lese- und Schreibzugriffe von insertionSort erfasst.

Die Algorithmen geben die Zugriffe selbst nicht zurück, dies geschieht durch die Getter-Methoden.

MethodPivot

Die Klasse ist ein Enum mit folgenden Werten:

- LEFT
  - Das pivot ist die Zahl, die ganz links in der Liste steht
- RIGHT
  - Das pivot ist die Zahl, die ganz rechts in der Liste steht
- MEDIANOF3
  - Das pivot wird ermittelt; indem das erste und letzte Element und ein Element aus der Mitte genommen wird, das MEDIAN aus diesen drei Elementen ist das pivot
  - Bsp: 3, 8, 5, 6, 1 –> MEDIAN von 3, 5, 1 -> pivot = 3
- RANDOM
  - Das pivot ist eine zufällige Zahl (aus der Liste, die zu sortieren ist)

quickSortTests und insertionSortTests (aber getrennte Klassen -> Austauschbarkeit)

- Die Tests benutzen das JUnit-Framework
- Die Tests sollen selbsterklärend benannt sein -> deutlich machen, welche Methode getestet wird und welchen Fall sie abdeckt
- Es sollen Grenzfälle geprüft werden, wie:
    - eine bereits aufsteigend sortierte Liste
    - eine bereits absteigend sortierte Liste
    - eine leere Liste, die sortiert werden soll
    - eine Liste mit einem Element die sortiert werden soll
    - eine Liste mit hunderten Elementen, die alle identisch sind
    - eine Liste mit hunderten Elementen, die zufällig sind