AD - Aufgabe 1 - Entwurf

Inhaltsverzeichnis

AD - Aufgabe 1 - Entwurf

Allgemeines

Team: 01, Eugen Deutsch, Phillip Schackier

Aufgabenaufteilung:

1. Eugen: insertionsort und sortnum
2. Phillip: quicksort und importnums
3. Beide: Entwurf, Hierarchie und Tests, sowie allgemeine Verbesserungen am Code des anderen.

Quellenangaben: Keine

Bearbeitungszeitraum:

03.11.15 - 2 Stunden: Beide
04.11.15 - 1 Stunde: Beide
05.11.15 - 1 Stunde: Eugen - 3 Stunden Phillip
06.11.15 - 2 Stunden: Eugen - 1 Stunde Phillip
07.10.15 - 3 Stunden: Beide
08.10.15 - 1 Stunde: Eugen

Aktueller Stand: Die Basisalgorithmen und die mit Zeitrückgabe, sowie die komplette Generatorklasse, sind implementiert und getestet worden. Lediglich der Benchmark selbst und die mit Zugriffsmessung fehlen noch.

Änderungen in der Skizze: Es wurden sortnumLeft sowie sortnumRight beigefügt, sowie importNums.

Package-/Ordner-Hierarchie und Sichtbarkeiten

sort
----- Sorter <public class>
---------- PivotMethod <public interface>
----- Generator <public class>
----- Benchmark <public class>
----- tests
---------- sortTest <public class>

Klassen/interface Hierarchien und Sichtbarkeiten

Es gibt lediglich Utility Klassen, was für uns bedeutet, dass sie nicht instanzierbar sind und nur als Ordner für die Methoden dienen und einige Einstellungsmöglichkeiten für diese bieten.

AD - Aufgabe 1 - Entwurf

Implementationen und Spezifikationen

Benchmark

- **public static void main(String** args**[])**: Erstellt mit Hilfe von sortnum, sortnumLeft und sortnumRight Zufallszahlen, importiert diese und führt jeweils insertionsort und quicksort damit durch. Folgende Fälle bestehen dabei:
    - 12, 100, 500, 1.000, 5.000, 10.000, 20.000, 50.000 Elemente
    Jeweils ungeordnet, links geordnet, rechts geordnet

Die Messergebnisse werden in eine *.csv Datei gespeichert. Quicksort bietet dabei jeweils 3 Messergebnisse: Einmal für sich selbst, einmal für Insertionsort und beides zusammen.

Generator

- **public static void sortnum(int** amount**)**: Erstellt eine Datei mit der angegebenen Anzahl an Zufallszahlen, wobei Zahlen auch wiederholt auftauchen können. Die Zahlen liegen zwischen 0 und amount und sind in der Datei als Strings gespeichert und werden mit einem Leerzeichen getrennt. Die Datei heißt “zahlen.dat” und sofern diese schon existiert, wird sie überschrieben. Der Dateiname ist an einer einzigen Stelle im Code änderbar.

Beispiel: 5 3 5 2 0

Bedingungen: amount >= 0
- **public static void sortnumLeft(int** amount**)**: Wie sortnum, allerdings sind die generierten Zahlen von links nach rechts geordnet in der Datei.

Beispiel: 0 2 3 5 5
- **public static void sortnumRight(int** amount**)**: Wie sortnum, allerdings sind die generierten Zahlen von rechts nach links geordnet in der Datei.

Beispiel: 5 5 3 2 0
- **public static AdtArray importNums(String** filename**)**: Importiert die Zahlen, die mit sortnum exportiert wurden und gibt diese in einer AdtArray zurück.

AD - Aufgabe 1 - Entwurf

Sorter

- **public static void insertionsort(AdtArray** array**, int** begin**, int** end**)**:
Funktionsweise:

1. Starte an der zweiten Stelle des Arrays und merke dir den darin enthaltenen Wert.
2. Merke dir die Position dieser Stelle.
3. Solange der gemerkte Wert aus 1 kleiner ist, als der Wert der in der Array an der Stelle der gemerkten Position minus 1 steht, mache folgendes:
   a) Schiebe den Wert der Array an der Position um einen Index höher.
   b) Dekrementiere die gemerkte Position um 1, bzw setzte mit der nächsten Position fort.
4. Sobald 3 abgeschlossen ist, setzt man den Wert an die gemerkte Position, die sich wahrscheinlich von der ursprünglichen Position unterscheidet.

Somit wird die Array Schritt für Schritt durchlaufen und bei jedem Schritt wird dafür gesorgt, dass an den Positionen darunter alles untereinander geordnet ist.

Bedingungen: array != null, begin > 0, end <= array.length, begin < end

- **public static void quicksort(AdtArray** array**, MethodPivot** pivot**)**:

Funktionsweise:
1. Wähle einen Start- und einen Endindex. Man nehme eine Datenstruktur, die von 0 bis 20 gefüllt ist, so ist der erste Startindex 0 und der erste Endindex 20.
2. Wähle irgendein Element zwischen den gewählten Indizes als Pivot-Element. Da dieses Element nicht vertauscht werden sollte, ist es am einfachsten, wenn man dieses Element mit dem Element des Endindexes tauscht.
3. Nun sorge dafür, dass alle Elemente der Datenstruktur, die kleiner als das Pivot-Element links davon und alle die größer sind, rechts davon gelegt werden:
   a) Laufe dazu von links nach rechts und suche nach einem Element, dass größer ist, als das Pivot-Element. [Man achte darauf, dass man nicht über den Endindex kommt]
   b) Nun laufe man von rechts nach links und suche ein Element, dass kleiner ist, als das Pivot-Element. [Man achte darauf, dass man nicht unter den Startindex kommt]
   c) Vertausche die beiden Elemente und wiederhohle ab “a”, solange nicht alle Indizes, die zwischen dem Start- und dem Endindex liegen,

durchlaufen wurden.

4. Tausche das Pivot-Element mit dem Index des letzten Tausches und gebe diesen zurück.
5. Starte bei 1 mit veränderten Start- und Endindizes: Einmal vom Startindex bis zur letzten Pivot-Element-Position (nicht einschließlich), die man aus 4 hat und einmal ab der Pivot-Element-Position (nicht einschließlich) bis zum Endindex, sodass man links und rechts vom Pivot jeweils anhand eines neuen Pivots ordnet, bis irgendwann ein Startindex kommt, der größer ist als der Endindex.

Bedingungen: array != null, pivotMethod != null, startIndex <= pivotMethod.getPivot() <= endIndex

Zusätzlich: Sobald zwischen begin und end 12 Elemente oder weniger vorhanden sind, sollen genau diese 12 Elemente von insertionsort sortiert werden und nicht von quicksort. Diese 12 ist dabei keine “magic number”, sondern ein leicht festlegbarer, aber konstanter Wert.

Realisierung des leicht wählbaren Pivot-Elements:
Eine kurze und einfache Möglichkeit ist die Nutzung von Lambdas. Das sähe wie folgt aus:

Intern:

```
@FunctionalInterface
public interface PivotMethod {
    int getPivotIndex(int start, int end);
}

public static void quicksort(AdtArray array, PivotMethod pivotMethod) {
    ...
    int pivotIndex = pivotMethod.getPivotIndex(left, right);
}
```

Extern:

```
AdtArray myArray = AdtContainerFactory.adtArray();
myArray.set(...)
// ...

Sorter.quicksort(array, (start, end) -> start); // Um immer das linkeste Element als
    Pivot zu wählen.
```

AD - Aufgabe 1 - Entwurf

Sorter.quicksort(array, (start, end) -> end); // Um immer das rechteste Element als Pivot zu wählen.

- **public interface PivotMethod { int getPivotIndex(int** start**, int** end**) }**:
  Interface für quicksort.

- **public static long insertionsortTime(AdtArray** array**, int** begin**, int** end**)**:
  insertionsort, mit dem Unterschied dass hier die benötigte Zeit (angegeben in Millisekunden) zurückgegeben wird.
- **public static long[] quicksortTime(AdtArray** array**, MethodPivot** pivot**)**:
  quicksort, aber mit der Rückgabe der benötigten Zeit (in Millisekunden), wobei die Zeit für die angewandten insertionsorts davon getrennt zurückgegeben wird.

- **public static long insertionsortSteps(AdtArray** array**, int** begin**, int** end**)**:
  insertionsort, aber mit der Rückgabe der Anzahl der lesenden und schreibenden Zugriffe (getrennt) auf das Array.

- **public static long[] quicksortSteps(AdtArray** array**, int** begin**, int** end**)**:
  quicksort, aber mit der Rückgabe der Anzahl der lesenden und schreibenden Zugriffe (getrennt) auf das Array, wobei die insertionsorts seperat zurückgegeben werden.

Die messenden Methoden messen allesamt ab dem Methoden Aufruf bis zum Abschluss des letzten Schrittes und berücksichtigen dabei alles dazwischen.

Fehlerbehandlung

Sollten die Bedingungen nicht erfüllt, oder sonst irgendein Fehler auftauchen, so wird die Operation ignoriert und alle Eingaben bleiben unverändert. Es wird unter keinen Umständen ein Fehler geworfen.

Tests

Getestet werden alle öffentlich zugänglichen Methoden. SortNum selbst wird nur oberflächlich getestet, da es schwierig ist zu prüfen, ob die Datei aus Zufallszahlen besteht.

Getestet werden unter anderem folgende Fälle:
- Unerfüllte Bedingungen
- leere Array

AD - Aufgabe 1 - Entwurf

- 1-Element Array
- 2-Element Array
- Eine Mittlere Anzahl an Elementen in der Array
- Eine Array mit vielen Elementen
- Links, rechts und unsortierte Arrays

Export

Alles zusammen: ad_sort.jar
Tests: ad_sortTests.jar
Klassen: ad_Sorter.jar, ad_Generator.jar, ad_Benchmark.jar