SKIZZE

SKIZZE ZUR AUFGABE 2 AUS DER VORLESUNGSREIHE “ALOGRITHMEN UND DATENSTRUKTUREN”

TEAM 10: ANTON, MESUT UND IGOR

Aufgabenaufteilung

Alle Aufgaben wurden gemeinsam entworfen und bearbeitet.

Quellenangaben

Lehrveranstaltung, Skript

Bearbeitungszeitraum

Datum	Dauer in Stunden
11.11.15	4
14.11.15	5
15.11.15	2
Gesamt	**11**

Aktueller Stand

Die Skizze ist fertig.

Änderungen in der Skizze

-

Skizze:

Es sollen folgende Methoden in Java implementiert werden: Zahlengenerator, Insertionsort und Quicksort als Sortieralgorithmen (Package: „Sorter“, Java Klassen: „NumberBuilder“, „Sorter“, „Benchmark“ – extern als sort.jar verfügbar). Dazu müssen auch JUnit Tests erstellt werden. (Package: „Sorter/tests“, Java Klassen „insertionJUt“, „quickJUt“ – extern als „insertionJUt.jar bzw. quickJUt.jar“ verfügbar). Die verwendete Datenstruktur ist ADTArray aus der letzten Aufgabe. Die Dateien müssen genau diese Namen besitzen, damit sie mit den anderen Gruppen austauschbar bleiben. Es folgen nun die Beschreibungen und Signaturen der Methoden.

NumberBuilder

Die NumberBuilder Klasse dient dazu, um eine Datei (explizit: „zahlen.dat“) mit der gewünschten Anzahl an Zufallszahlen (positive ganze Zahlen) zu erstellen.

Funktionale Vorgabe

- Die Elemente sind vom Typ „ganze positive Zahl“.
- Die Zahlen können wiederholt auftauchen.
- Erstellung einer Datei mit dem Namen „zahlen.dat“.

Technische Vorgabe

- Die gewünschte Anzahl der Zufallszahlen kann ≥ 0 sein.
- Besteht die Datei schon, wird sie folglich überschrieben.

Public Operationen (semantische Signatur/syntaktische Signatur)

generateSortNum: int → file

(Input: Integer count | Output: File file)

Mit Hilfe dieser Methode erzeugen wir eine Datei mit der angegebenen Anzahl an Zufallszahlen.

sortNumLeft: int → file

(Input: Integer count | Output: File file)

Die Methode arbeitet generell wie die Methode generateSortNum, allerdings mit der Bedingung, dass die generierten Zahlen von links nach rechts geordnet sind in der erstellten Datei.

sortNumRight: int → file

(Input: Integer count | Output: File file)

Die Methode arbeitet generell wie die Methode generateSortNum, allerdings mit der Bedingung, dass die generierten Zahlen von rechts nach links geordnet sind in der generierten Datei.

importNumFile: string → array

(Input: String filename | Output: ADTArray array)

Bei dieser Methode wird der angegebene Pfad bzw. Dateiname der erstellten Datei mit den Zufallszahlen von generateSortNum() importiert in einem ADTArray zurück.

Sorter

In der Klasse „Sorter“ sind die Sortierungsalgorithmen „Insertionsort“ und „Quicksort“ implementiert.

Public Operationen (semantische Signatur/syntaktische Signatur):

insertionSort: array × anfang × ende → array

(Input: ADTArray array × int pos × int pos | Output: ADTArray array)

Technische Vorgabe

- Die Startposition fängt an der zweiten Stelle des Arrays an.
- Wir merken uns diese Position.
- Solange der Wert von dem Element aus der gemerkten Position kleiner, als der Wert von der gemerkten Position – 1 ist, werden diese Elemente miteinander getauscht. Bei diesem Durchlauf wird das aktuelle Element des Zeigers mit den Elementen davor verglichen indem wir die Position wieder dekrementieren, bis das Element die richtige Position im Array erreicht hat.

- Nach diesem Zyklus wird der Zeiger des Arrays um eine Position erhöht, dadurch iterieren wir durch das ganze Array.

Bedingungen für InsertionSort

- Ein leeres Array oder ein Array mit nur einem Element muss ohne Veränderung zurückgegeben werden.
- Als Eingabewert muss die eingegebene Anfangsposition immer kleiner als die eingegebene Endposition sein.
- Ansonsten greift die erste Bedingung.

quickSort: array × method_pivot → array

(Input: ADTArray array × Integer pivot | Output: ADTArray array)

Technische Vorgabe

- Wir setzen uns einen Startpunkt und einen Endpunkt als jeweils die erste und die letzte Position des Arrays (Startindex / Endindex).
- Als nächstes brauchen wir ein Pivot-Element:
  - Parameter 1 entspricht das linkeste Element des Arrays
  - Parameter 2 entspricht das rechteste Element des Arrays
  - Parameter 3 entspricht der Zufallsposition des Arrays
  - Parameter 4 entspricht „Median of 3“. Das linkeste, rechteste und mittlere Element des Arrays werden verglichen und der mittlere Wert als Pivot gesetzt.
- Anschließend iterieren wir durch das Array und vergleichen, ob die restlichen Elemente <= Pivot oder >= Pivot sind, somit sorgen wir dafür, dass die Elemente, die kleiner als der Pivot sind, links und die größer als der Pivot sind rechts stehen. Dieser Algorithmus wird solange durchgespielt, bis es keine tauschbaren Elemente mehr gibt und das Pivot Element auf sich selbst zeigt.
- Dabei geht man bei der funktionsweise wie folgt vor:
- Die Ausführung des Algorithmus hängt von der Position des Pivots ab. Wenn das Pivot Element am Anfang des Arrays steht, so muss man von rechts anfangen und ein Element suchen, welches kleiner als das Pivot-Element ist. Sobald man eins hat, werden diese getauscht und somit steht das Pivot nicht mehr am Anfang des Arrays. Die gleiche Vorgehensweise benutzen wir beim Durchlaufen von links nach rechts auf der Suche nach einem Element, welches größer als das Pivot ist und tauschen diese. Analog dazu verhält sich der Algorithmus, wenn das Pivot rechts steht (nur die Startposition ist dann links).
- Wenn die linke oder rechte Seite vom Pivot noch sortiert werden muss, wird auf beiden Seiten jeweils ein neues Pivot Element bestimmt und der gleiche Algorithmus durchgespielt.

Bedingungen für Quicksort

- Die Wahl des Pivots muss zwei Kriterien erfüllen. 1: >= Startindex 2: <=Endindex.
- Sobald der Quicksort-Alg. beim Sortieren feststellt, dass es weniger als 12 Elemente zu sortieren gilt, soll Quicksort auf Insertionsort verzweigen.

Benchmark

Benchmark ist unsere separate Klasse, die mit Hilfe von generateSortNum (auch sortNumLeft u. sortNumRight) Zufallszahlen generiert, diese importiert und auf denen insertionSort und quickSort ausführt. Dazu werden die Laufzeit von Algorithmen (insertionSort und quickSort getrennt, d.h. also quickSort abzüglich der Zeit, die dort insertionSort benötigt) und auch die Anzahl der lesenden und schreibenden Zugriffe auf das Array (insertionSort und quickSort getrennt, d.h. also quickSort abzüglich der Zugriffe, die insertionSort durchführt) gemessen und als spezielle Datei für weitere Analyse zurückgegeben.

Funktionale Vorgabe

- Erstellung einer Datei mit dem Suffix „.csv“.

Technische Vorgabe

- Besteht die Datei schon, wird sie folglich überschrieben.

Public Operationen (semantische Signatur/syntaktische Signatur):

main: ∅ → file

(Input: keine Parameter | Output: File)

Testen mit JUnit

- Alle obengenannten Methoden aufrufen.
- Es sollen folgende Situationen betrachtet werden: ein Array besitzt keine Elemente, genau ein Element, genau zwei Elemente, mehrere Elemente. Auch sollen schon sortierte Arrays berücksichtigt werden (von links nach rechts und umgekehrt).
- Die fehlerhaften Situationen sollen berücksichtigt werden: z.B. der übergebene Wert von Pivot-Method liegt außerhalb der abgestimmter Reihenfolge (von 1 bis 4) usw.