**Team**: 03, Sebastian Diedrich – Murat Korkmaz

**Aufgabenaufteilung**:

* Aufgaben, für die Teammitglied 1 verantwortlich ist:

1. Skizze

Dateien, die komplett/zum Teil von Teammitglied 1 implementiert/bearbeitet wurden:

* Aufgaben, für die Teammitglied 2 verantwortlich ist:

1. Skizze

Dateien, die komplett/zum Teil von Teammitglied 1 implementiert/bearbeitet wurden:

**Quellenangaben**: Vorlesung am 03.12.15

**Bearbeitungszeitraum**:

**Aktueller Stand**:

**Skizze**: (ab Seite 2)

**Skizze Aufgabe 3:**

**Aufgabe: 3.1**

Ziel: Zahlengenerator erweitern

Angaben zur Implementation:

* Das Trennungssymbol zwischen den einzelnen Zahlen soll ein Leerzeichen sein. Es sollen nur positive Zahlen erzeugt werden. Es soll die Möglichkeit bestehen eine Liste mit Duplikaten und ohne Duplikate von Elementen zu erstellen.
* Auch der beste und schlimmste Fall (Zahlen sind sortiert vs. Zahlen sind umgekehrt sortiert) soll mit einer beliebigen Anzahl von Zahlen weiterhin generierbar sein.
* Die Auswahl der Erzeugungsmöglichkeiten soll mittels einem Parameter erfolgen.

Vorgaben für die Implementation:

* Semantische Vorgabe:
  1. anzahlZahlen x pfad x enum -> Datei  
     Datei enthält die gewünschte Anzahl von Elementen, die in der Weise angeordnet sind, wie durch das Enum definiert wurde und ggf. Duplikate. Die Datei wird unter dem „pfad“ gespeichert.
  2. Größe der Zahlenwerte  
     - Dublikate: Zahlenwerte sollen im Bereich von **0 bis 1.000** liegen.  
     - Keine Dublikate: Zahlenwerte sollen im Bereich **0 bis anzahlZahlen+500** liegen.
* Syntaxtische Vorgabe:
  1. Name der Klasse: SortNum
  2. Name der Methode: sortNum
  3. Enum-Parameter und Aufruf der Methode:

RANDOM\_WITH\_DUBLICATES -> Zufallszahlen ohne Dublikate

RANDOM\_WITHOUT\_DUBLICATES -> Zufallszahlen mit Dublikaten

BEST\_CATE -> Zahlen aufsteigend (1..Anzahl)

WORST\_CASE -> Zahlen absteigend (Anzahl..1)

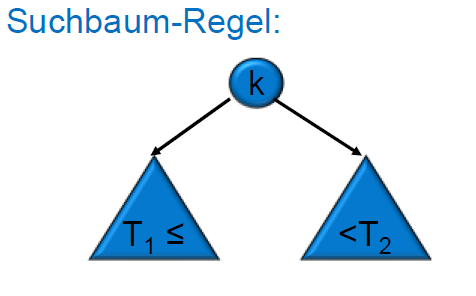
*Aufruf der Methode:*   
Sortnum.sortNum(**int** anzahlZahlen, **String** path, **Parameter** parameter)

* 1. Endung der Datei: .dat

**Aufgabe: 3.2**

Ziel: AVL-Baum implementieren als ADT

Angaben zur Implementation:

1. Ein AVL-Baum (nach **A**delson-**V**elskii und **L**andis) ist ein binärer Suchbaum, der   
   höhenbalanciert ist. Dabei muss folgendes Kriterium (ggf. durch Rotationen) erfüllt werden:  
   Für jeden Knoten *v* gilt, dass sich die Höhe des rechten Teilbaumes ***h*(*Tr*)** von *v* und die Höhe des linken Teilbaumes ***h*(*Tl*)** von *v* um maximal 1 unterscheiden.  
   (siehe auch Skript von Prof. Klauck – AVL-Bäume, Seite 3 ff)
2. Betrachtet man einen Knoten des Baumes, so sind alle linken Folgeknoten („Linkes Kind“) kleiner oder gleich groß in ihrem Wert und alle rechten Folgeknoten („Rechtes Kind“) größer in ihrem Wert.

Folgendes soll immer gelten (Invarianten):

1. funktional
   * Das Einfügen von Elementen passiert nur auf Blattebene
   * Das Löschen kann auf jeder Ebene erfolgen
   * Nach diesen Operationen muss ggf. Rotiert werden, damit der Baum wieder balanciert ist
2. technisch
   * Rekursive Struktur, auf der lokal gearbeitet werden kann. Sonst verlieren wir die logarithmische Komplexität
   * Der AVL-Baum enthält folgende Objektmengen:

* value: Wert des Knotens
* adtTreeSmaller, adtTreeBigger: linkes und rechtes Kind des Knotens
* smallerTreeHeight, biggerTreeHeight: aktuelle Höhe des linken und rechten Kindsknoten

Folgende Operationen sollen bereitgestellt werden (semantische Signatur):

* *create*: ein leeren ADT-AVLTree erstellen   
  („nichts“ -> avlTree)

Fehlerbehandlung: ignorieren (es wird kein Fehler geworfen)

* *isEmpty*: Abfrage, ob der ADT-AVLTree kein Knoten enthält

(avlTree -> Wahrheitswert)   
Fehlerbehandlung: ignorieren

* *high*: Höhe des (Teil-)Baumes

(avlTree -> Zahl)   
Fehlerbehandlung: ignorieren

* *insert*: Ein Knoten wird an richtiger Position an dem AVLTree eingehängt, wobei das Element dem value des neu erstelten Knotens darstellt.  
  (avlTree x elem -> elem)

Fehlerbehandlung: 0 zurückgeben (Typ: elem)

*…*

Syntaxtische Vorgaben:

Dateiname: AVLTree.jar

Klassenname: AVLTree

Anwendung der oben genannten Operationen:

*createS*: ADTStack.createS()

*push*: <Objektname>.push(elem)

*pop:* <Objektname>.pop()

*top:* <Objektname>.top()

*isEmptyS*: <Objektname>.isEmptyS()

**Aufgabe: 2.2**

Quellen:

* <http://users.informatik.haw-hamburg.de/~klauck/AlguDat/TIB3-AD-skript.pdf> (S. 57 ff)
* Vorlesung vom 04.11.2015 bei Prof. Dr. Klauck

Ziel: Sortieren einer Zahlenreihe basierend auf dem Sortieralgorithmus: **Insertion-Sort**

Allgemeiner Ablauf:

Pseudo-Code aus dem Vorlesungsskript von Prof. Dr. Klauck (S.58):

public void insertionSort(int startPos, int endPos){

for ( int i = startPos; i <= endPos ; i++ )

{

int j = i;

Datensatz<T> t = a[i];

int k = t.key;

while(a[j-1].key > k)

{

a[j] = a[j-1];

j = j-1;

}

a[j] = t;

}

}

Anpassung des Pseudo-Codes für das folgende Beispiel:

* Key-Abfrage nicht nötig, da Elemente Zahlen sind
* Erweiterung der while-Bedingung (j>1), um eine sichere Terminierung zu gewährleisten

Erläuterung an einem Beispiel:

1. Gegeben sei eine Zahlenreihe *Z* = [5,3,6,1,2,7,4]. Begonnen wird mit dem Index *i* = 2 (in unserem Beispiel die Zahl 3).
2. Gespeichert wird *i* in *j* und Z[i] in *k*. Dieses ist nötig, da ggf. eine höher-wertige Zahl an diese Position verschoben wird. Im weiteren Verlauf wird nun beginnend bei der Zahl mit dem *Index j-1,* *k* mit dieser verglichen. Wobei j nicht kleiner als 2 sein darf, da j-1 minimal den Index der ersten Zahl der Zahlenreihe darstellen kann.
3. Sollte *j* nicht kleiner als 2 sein, gehe zu (4), sonst zu (6)
4. Sollte Z[j-1] größer als *k* sein, gehe zu (5), sonst zu (6)
5. Es wird die Zahl mit dem *Index j-1* an die Position *j* kopiert und *j* wird um 1 verkleinert:   
   *Z* = [5,**5**,6,1,2,7,4]

Gehe zu (3)

1. k wird nach Z[j] gespeichert:   
   *Z* = [**3**,5,6,1,2,7,4]
2. Der Index i wird um 1 erhöht und ab (2) wiederholt bis einschließlich i = n (n = Anzahl der Zahlen).

Hinweise zur Implementierung:

Um eine doppelte Abfrage in der while-Schleife zu verhindern, kann an der Position Z[0] ein Dummy (mit kleinstmöglichem Wert) gespeichert werden, welches als Stopper-Element dient und dazu führt, dass die while-Schleife sicher terminiert.

**Test**

Um die Richtigkeit und die Zuverlässigkeit des Algorithmus zu gewährleisten, sollen umfangreiche JUnit-Tests implementiert werden.

Dateiname des Jar-Files: insertionJUt.jar

Mindestanforderung:

1. Grenzfälle
   1. Zahlenreihe mit n = 2 Zahlen (sortiert, unsortiert und beide Zahlen gleich)
   2. Zahlenreihe mit n = 1.000 Zahlen (umgekehrt sortiert)
      1. Beispiel: (1.000, 999, … , 5, 4, 3, 2, 1)
   3. Zahlenreihe mit n = 1.000 Zahlen (sortiert)
      1. Beispiel: (1, 2, …, 999, 1.000)
2. Belastung
   1. Zahlenreihe mit n > 10.000
   2. Zahlenreihe mit n > 100.000
   3. Zahlenreihe mit n > 200.000

Vorgaben für die Implementation:

* Semantische Vorgabe:
  1. Insertionsort: array x startPos x endPos -> array  
     startPos: dabei handelt es sich um eine positive Zahl, die nicht größer sein darf als die Endposition (endPos)
* Syntaxtische Vorgabe:
  1. Name der Klasse: Insertionsort
  2. Name der Methode: insertionsort
  3. Aufruf der Methode:   
     Insertionsort.insertionsort(ADTArray array, int startPos, int endPos)

**Aufgabe: 2.3**

Quellen:

* <http://users.informatik.haw-hamburg.de/~klauck/AlguDat/TIB3-AD-skript.pdf> (S. 63 ff)
* Vorlesung vom 04.11.2015 bei Prof. Dr. Klauck

Ziel: Sortieren einer Zahlenreihe basierend auf dem Sortieralgorithmus: **Quick-Sort**

Allgemeiner Ablauf:

Pseudo-Code:

void quicksort(int iLinks, int iRechts, Enum pivotAuswahl)

{

int pivot,i,j;

if ( iRechts > iLinks )

{

i = iLinks;

j = iRechts;

int pivotIndex = method\_pivot(pivotAuswahl, i , j);

pivot = Z[pivotIndex];

while(1)

{

while(Z[i] >= pivot && i < iRechts)

{

i++;

}

while(Z[j] < pivot && j > iLinks)

{

j--;

}

if ( j <= i ) break;

//Index-Pivot merken, wenn j <= I noch nicht erfüllt

if(indexPivot == i) indexPivot = j;

swap(i,j);//vertauschen

}

swap(i,indexPivot);//Pivotelement in die Mitte tauschen

quicksort(iLinks,i-1,pivotAuswahl);

quicksort(i+1,iRechts,pivotAuswahl);

}

}

int method\_pivot(Enum pivotAuswahl, int iLinks, int iRechts){

if(pivotAuswahl = links) return iLinks;

if(pivotAuswahl = rechts) return iRechts;

if(pivotAuswahl = random) return random(iLinks .. iRechts)

if(pivotAuswahl = medianOf3) return medianOf3(iLinks, iRechts, (int) iRechts/2)

}

Definitionen:

*Pivot-Element:*

Das Pivot-Element dient als Referenzelement. Vor der Teilung der Zahlenreihe (Rekursiver Aufruf), befinden sich links vom Pivot-Element nur Elemente die kleiner sind und rechts davon nur Elemente die größer oder gleich sind.

Die Auswahlbestimmung, wird als Parameter beim Aufruf der Methode „quicksort“ mit übergeben. Je nach Auswahl befindet sich das Pivot-Element links, rechts, wird mittels Zufallszahl bestimmt oder wird durch die Bestimmung des Medians von drei Zahlen ermittelt.

*Start und End Index:*

*iLinks* = Index der Zahl ganz links in der (Teil)Zahlenreihe

*iRechts* = Index der Zahl ganz rechts in der (Teil)Zahlenreihe

*Iteratoren:*

*i = von links nach rechts*

*j = von rechts nach links*

Algorithmus:

Zunächst wird das Pivot-Element bestimmt. Wie dieses geschieht, ist nicht fest vorgegeben.

Der Pseudocode wurde generisch beschrieben und deckt alle 4 beschriebenen Auswahlmethoden ab.

Das aktuelle Pivot-Element wird so lange mit allen anderen Zahlen der (Teil)Zahlenreihe verglichen, bis eine Zahl gefunden wird, die kleiner ist als das Pivot-Element (beginnend bei *iRechts*) und eine Zahl, die größer oder gleich ist, als das aktuelle Pivot-Element (beginnend bei *iLinks*). Dabei bewegen sich *i* und *j* aufeinander zu. Die gefundenen Zahlen werden dann getauscht.

Dieses wird so lange fortgesetzt, bis *i* und *j* „zusammenstoßen“ (bzw: j <= i). Das aktuelle Pivot-Element wird dann mit der Zahl an der Position *i* getauscht.

Nun wird die (Teil)Zahlenreihe in zwei Teile aufgeteilt, wobei das Pivot-Element die „Grenze“ der beiden Teil-Zahlenreihen darstellt. Da das Pivot-Element sich bereits an seiner endgültigen Position befindet, wird es weder der linken noch der rechten Teil-Zahlenreihe zugeordnet.

Der Algorithmus wird nun auf jede Teil-Zahlenreihe angewandt, bis jede Teil-Zahlenreihe nur noch aus einer Zahl besteht. Nun ist die Zahlenreihe sortiert und der Algorithmus endet, da die if-Bedingung (iRechts > iLinks) nicht mehr erfüllt wird.

Hinweis zur Implementierung:

Bei der Erhöhung des Iterators *i* muss beachtet werden, dass *i* nicht größer als iRechts sein darf, da wir uns sonst außerhalb der (Teil)Zahlenreihe befinden.

Bei der Erniedriegung des Iterators jmuss beachtet werden, dass jnicht kleiner als iLinks sein darf, da wir uns sonst außerhalb der (Teil)Zahlenreihe befinden.

* Semantische Vorgabe:
  1. Quicksort: array x method\_pivot -> array  
     method\_pivot: bestimmt die Pivot-Element-Auswahl  
     Folgende Auswahlmöglichkeiten sollen implementiert werden:
     1. Immer ganz links
     2. Immer ganz rechts
     3. Medianof3
        1. Median von folgenden Zahlen:
           1. Zahl an erster Position
           2. Zahl an letzter Position
           3. Zahl an mittlerer Position
     4. Random
* Syntaxtische Vorgabe:
  1. Name der Klasse: Quicksort
  2. Name der Methode: quicksort
  3. Aufruf der Methode: Quicksort.quicksort(ADTArray array, Enum pivotAuswahl)
* Sollte die (Teil)Zahlenreihe weniger als 12 Elemente beinhalten, werden die (Teil)Zahlenreihen mittels Insertionsort sortiert. Deshalb wurde diese Methode um die Parameter, startPos und endPos erweitert.

**Test**

Um die Richtigkeit und die Zuverlässigkeit des Algorithmus zu gewährleisten, sollen umfangreiche JUnit-Tests implementiert werden.

Dateiname des Jar-Files:

* quickJUt.jar

Mindestanforderung:

1. Grenzfälle
   1. Zahlenreihe mit n = 2 Zahlen (sortiert, unsortiert und beide Zahlen gleich)
   2. Zahlenreihe mit n = 1.000 Zahlen (umgekehrt sortiert)
      1. Beispiel: (1.000, 999, … , 5, 4, 3, 2, 1)
   3. Zahlenreihe mit n = 1.000 Zahlen (sortiert)
      1. Beispiel: (1, 2, …, 999, 1.000)
2. Belastung
   1. Zahlenreihe mit n > 10.000
   2. Zahlenreihe mit n > 100.000
   3. Zahlenreihe mit n > 200.000

**Aufgabe 2.4**

Versuchsaufbau:

Es wird eine Kopie der Algorithmus-Implementation erstellt und um folgende Komponenten erweitert:

* Laufzeit:
  1. Zu Beginn des Algorithmus wird die aktuelle Zeit festgehalten. Am Ende des Algorithmus wird die aktuelle Zeit festgehalten und der Betrag der Differenz der beiden ausgegeben. Dieser Betrag entspricht der Laufzeit. (zu beachten ist dabei, dass diese Laufzeit NUR mit Laufzeiten verglichen werden darf, die ebenfalls auf dem gleichen Ausführungssystem gemessen worden sind)
  2. Bei der Laufzeitmessung von Quicksort muss beachtet werden, dass die Laufzeit von Insertionsort abgezogen wird. (Sortierung von Zahlenreihen mit weniger als 12 Zahlen)
* Zugriffe (Lesen und Schreiben)
  1. Lesen

Ein Lesezugriff ist definiert, durch das Auslesen eines Wertes aus der Zahlenreihe.

* 1. Schreiben

Ein Schreibzugriff ist definiert, durch das mutieren der Zahlenreihe oder durch die Belegung einer Variablen mit einem Wert.

Messungen:

Vorgaben für die Messungen: Anzahl der Zahlen der Zahlenfolgen:

1. 500
2. 1.000
3. 2.000
4. 4.000
5. 8.000
6. 16.000
7. 32.000
8. 64.000
9. 128.000
10. Mittels sortNum wird eine zufällig generierte Zahlenfolge nach obigen Vorgaben erstellt.
    1. Messung der Laufzeit mittels Insertionsort
    2. Messung der Laufzeit mittels Quicksort (links)
    3. Messung der Laufzeit mittels Quicksort (rechts)
    4. Messung der Laufzeit mittels Quicksort (MedianOf3)
    5. Messung der Laufzeit mittels Quicksort (random)
11. Mittels sortNum wird eine zufällig generierte Zahlenfolge nach obigen Vorgaben erstellt.
    1. Messung der Schreib- und Lesezugriffe mittels Insertionsort
    2. Messung der Schreib- und Lesezugriffe mittels Quicksort (links)
    3. Messung der Schreib- und Lesezugriffe mittels Quicksort (rechts)
    4. Messung der Schreib- und Lesezugriffe mittels Quicksort (MedianOf3)
    5. Messung der Schreib- und Lesezugriffe mittels Quicksort (random)

*Damit ergeben sich 2x5x9 Messungen.*

Resultate:

Die Ergebnisse der obigen Messungen werden in eine Excel-Tabelle eingetragen:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ***Algorithmus X*** |  |  |  |  |  |  |  |  |  |
|  | 500 | 1000 | 2000 | 4000 | 8000 | 16000 | 32000 | 64000 | 128000 |
| Laufzeit (ms) |  |  |  |  |  |  |  |  |  |
| Lesezugriffe |  |  |  |  |  |  |  |  |  |
| Schreibzugriffe |  |  |  |  |  |  |  |  |  |

Aus den Ergebnissen werden Excel-Graphiken erzeugt, um die Steigung der einzelnen interpolierten Kurven vergleichen zu können.

Die daraus resultierenden Schlussfolgerungen werden zusammen mit den Graphiken in einem PDF dokumentiert.

Mögliche Fragen wären: Welcher Algorithmus ist am schnellsten (gemessen an der Laufzeit). Welcher Algorithmus benötigt am wenigsten Schreib- bzw. Lesezugriffe!