**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**: 03.12 (6h)

**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)

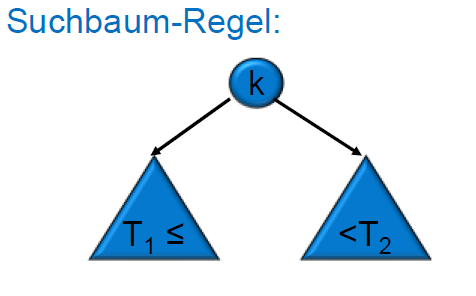
*Aufruf der Methode:*   
Sortnum.sortNum(**int** anzahlZahlen, **String** path, **Parameter** parameter)

* 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 -> avlTree)

Fehlerbehandlung: ignorieren

* *delete*: Ein Knoten wird aus dem AVLTree entfernt

(avlTree x elem -> avlTree)

Fehlerbehandlung: ignorieren

* *print*: Der AVLTree wird in einer png-Datei als Graph angezeigt

(avlTree x pfad x dateiName -> png)

Fehlerbehandlung: ignorieren

Syntaxtische Vorgaben:

Dateiname: AVLTree.jar

Klassenname: AVLTree

Anwendung der oben genannten Operationen:

*create*: AVLTree.create()

*isEmpty*: <Objektname>.isEmpty()

*high:* <Objektname>.high()

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

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

*print*: <Objektname>.print(pfad, dateiName)

**Beschreibung der Rotationsarten und der Durchführung:**

Annahme:  
Wir gehen davon aus, dass der linke Teilbaum eines Knotens die kleineren und der rechte Teilbaum die größeren Elemente enthält.

Definition:

Balance = Differenz der Höhe vom Rechten und Linken Teilbaum (rT – lT)

*d* = Knoten mit Disbalance (Höhe ist +2 oder -2)

*k* = Kindknoten von *d*, der die Disbalance auslöst. Dieser hat die Balance +1 oder -1.

Nachdem das Element eingefügt wurde, läuft man rekursiv den Einfügepfad zurück und prüft bei jedem Knoten die Balance. Sobald man eine Disbalance entdeckt hat, wird das Vorzeichen der Balance von *d* mit der Balance von *k* verglichen. Haben beide das gleiche Vorzeichen, reicht eine einfache Rotation, um den Baum wieder zu balancieren. Sind die Vorzeichen unterschiedlich, liegt eine Problemsituation.

Fall 1: Linksrotation:

Bedingung:

* Balance von d ist +2
* Balance von k ist +1

Für den Fall einer Linksrotation sind beide Balancen positiv.

Ablauf der Linksrotation:   
Der linke Teilbaum von *k* wird der neue rechte Teilbaum von *d*. *d* selbst wird neuer linker Teilbaum von k. Und k nimmt die ehemalige Position von d ein.

Anpassung der Knotenhöhen nach der Rotation:  
Die Höhen aller Teilbäume bleiben gleich. Die Höhe von *d* verringert sich um 2, die Höhe von *k* bleibt unverändert.

Fall 2: Rechtsrotation:

Bedingung:

* Balance von *d* ist -2
* Balance von *k* ist -1

Für den Fall einer Rechtsrotation sind beide Balancen negativ.

Ablauf der Rechtsrotation:   
Der rechte Teilbaum von *k* wird der neue linke Teilbaum von *d*. *d* selbst wird neuer rechter Teilbaum von k. Und k nimmt die ehemalige Position von d ein.

Anpassung der Knotenhöhen nach der Rotation:  
Die Höhen aller Teilbäume bleiben gleich. Die Höhe von *d* verringert sich um 2, die Höhe von *k* bleibt unverändert.

Fall 3: Problemsituation rechts:

Bedingung (Vorzeichen der Balancen sind ungleich, wobei die Balance von *d* negativ ist):

* Balance von *d* ist -2
* Balance von *k* ist +1

Ablauf:   
Es wird zunächst eine Linksrotation auf *k* durchgeführt, im Anschluss eine Rechtsrotation auf *d*.

Anpassung der Knotenhöhen:  
Nach der Linksrotation vermindert sich die Höhe von *k* um 1 und die Höhe des ehemaligen rechten Teilbaums von *k* erhöht sich um eins. Nach der anschließenden Rechtsrotation vermindert sich die Höhe von *d* um 2, alle anderen Teilbäume bleiben in ihrer Höhe unverändert.

Fall 4: Problemsituation links:

Bedingung (Vorzeichen der Balancen sind ungleich, wobei die Balance von *d* positiv ist):

* Balance von *d* ist +2
* Balance von *k* ist -1

Ablauf:   
Es wird zunächst eine Rechtsrotation auf *k* durchgeführt, im Anschluss eine Linksrotation auf *d*.

Anpassung der Knotenhöhen:  
Nach der Rechtsrotation vermindert sich die Höhe von *k* um 1 und die Höhe des ehemaligen linken Teilbaums von *k* erhöht sich um eins. Nach der anschließenden Linksrotation vermindert sich die Höhe von *d* um 2, alle anderen Teilbäume bleiben in ihrer Höhe unverändert.