Team: 03, Sebastian Diedrich – Murat Korkmaz

Aufgabenaufteilung:

- Aufgaben, für die Teammitglied 1 verantwortlich ist:
    1. Skizze
    2. sortNum
    3. sortNum Tests
    4. AVLTree Tests

  Dateien, die komplett/zum Teil von Teammitglied 1 implementiert/bearbeitet wurden:
    1. sortNum
- Aufgaben, für die Teammitglied 2 verantwortlich ist:
    1. Skizze
    2. AVLTree Implementation

  Dateien, die komplett/zum Teil von Teammitglied 2 implementiert/bearbeitet wurden:
    1. AVLTree Implementation

Quellenangaben: Vorlesung am 03.12.15, Skript AD (Prof. Klauck)

Bearbeitungszeitraum: 03.12 (8h), 04.12 (7h)

Aktueller Stand:

- Skizze Version 1 (Rückmeldung durch Prof. offen)
- SortNum fertig implementiert
- AVLTree: create, isEmpty, high, insert sowie print sind fertig implementiert

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:
  - 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.
  - 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:
  - Name der Klasse: SortNum
  - Name der Methode: sortNum
  - Enum-Parameter und Aufruf der Methode:

RANDOM_WITH_DUBLICATES	->	Zufallszahlen ohne Dublikate
RANDOM_WITHOUT_DUBLICATES	->	Zufallszahlen mit Dublikaten
BEST_CASE	->	Zahlen aufsteigend (1..Anzahl)
WORST_CASE	->	Zahlen absteigend (Anzahl..1)

    Aufruf der Methode:
    Sortnum.sortNum(anzahlZahlen, path, parameter)
  - Endung der Datei: .dat

Aufgabe: 3.2

Ziel: AVL-Baum implementieren als ADT

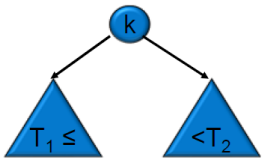
Quelle: http://users.informatik.haw-hamburg.de/~klauck/AlguDat/TIB3-AD-skript.pdf
(S. 91 ff)

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.

Suchbaum-Regel:



Folgendes soll immer gelten (Invarianten):

I) funktional
- Das Einfügen von Elementen passiert nur auf Blattebene
- Das Einfügen von Duplikaten ist verboten
- Das Löschen kann auf jeder Ebene erfolgen
- Nach Einfüge- und Löschoperationen muss ggf. Rotiert werden, damit der Baum wieder balanciert ist

II) 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
- high: höhe des Knotens

Folgende Operationen sollen bereitgestellt werden (semantische Signatur):

- *create*: einen leeren AVLTree erstellen
  („nichts“ -> avlTree)
  Fehlerbehandlung: ignorieren (es wird kein Fehler geworfen)

- *isEmpty*: Abfrage, ob der AVLTree keinen Knoten enthält
  (avlTree -> Wahrheitswert)
  Fehlerbehandlung: ignorieren

- *high*: Höhe des (Teil-)Baumes bzw. des Knotens
  (avlTree -> Zahl)
  Fehlerbehandlung: ignorieren

- *insert*: Ein Element (als Knoten) wird in den AVLTree eingehängt
  (avlTree x elem -> avlTree)
  Fehlerbehandlung: ignorieren

- *delete*: Ein Element (als Knoten) wird aus dem AVLTree entfernt
  (avlTree x elem -> avlTree)
  Fehlerbehandlung: ignorieren

- *print*: Der AVLTree wird in einer png-Datei als Graph gespeichert
  (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 (+2 oder -2)

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

Allgemeines Vorgehen:

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 (Fall 1 und 2), um den Baum wieder zu balancieren. Sind die Vorzeichen unterschiedlich, liegt eine Problemsituation (Fall 3 und 4) vor.

Anpassung der Knotenhöhen:
Die (neue) Höhe ergibt sich aus folgender Berechnung: max(li Kind, re Kind) +1

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.

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.

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*.

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*.

Einfügen eines Elementes in den AVLTree:

Definitionen:

e = Einzufügendes Element, *e.value* = Wert des Elementes
a = aktueller Knoten der betrachtet wird, *a.value* = Wert des Knotens

Algorithmus:

1. Befindet sich kein Knotem im AVL-Baum, wird *e* zum Wurzelknoten des Baumes und der Algorithmus ist beendet. Sonst gehe zu 2.
2. Beginne beim Wurzelknoten des AVL-Baumes
3. Ist *e.value* kleiner als *a.value* betrachte linken Kindknoten von *a*, ist er größer, betrachte rechten Kindknoten von *a*.
4. Befindet sich kein Kind an dieser Stelle, füge neues Element ein. Sonst gehe zu Schritt 3, bis das Element eingefügt wurde.
5. Setze die Höhe von *e* auf 1.
6. Laufe den Einfüge-Pfad (bis einschließlich der Wurzel des AVL-Baumes) zurück und aktualisiere jeweils die Höhe von *a* und prüfe ob eine Disbalance vorliegt.
7. Sollte eine Disbalance in *a* vorliegen, rotiere nach den oben beschriebenen Verfahren und passe die Höhen der Knoten erneut an.

Löschen eines Elementes aus dem AVLTree:

Definitionen:

l = das zu löschende Element, *l.value* = Wert des Elementes
a = aktueller Knoten der betrachtet wird, *a.value* = Wert des Knotens
c = Knoten, der an die Stelle von *l* kopiert wird
kL = Knoten des linken Teilbaumes von *a*
kR = Knoten des rechten Teilbaumes von *a*

Algorithmus:

1. Beginne beim Wurzelknoten des AVL-Baumes
2. Beträgt *l.value* gleich *a.value, gehe zu Schritt 4.*
3. Ist *l.value* kleiner als *a.value*, betrachte den linken Kindknoten von a, ist er größer, betrachte den rechten Kindknoten von *a*. Befindet sich an der ausgewählten Position kein Knoten, ist der Algorithmus beendet (*l* befindet sich nicht im Baum). Sonst gehe zu 2.
4. Ist die gefundene Position ein Blatt, lösche den Knoten und gehe zu 9. Andernfalls fahre mit 5. fort.

5. Merke dir die Position von *l*.

6. Suche im linken Teilbaum von *l* den Knoten mit dem höchsten Wert. Besitzt *l* keinen linken Teilbaum, suche im rechten Teilbaum von *l* nach dem niedrigsten Wert. Der gefundene Knoten wird *c*.

7. Kopiere *c* an die gemerkte Position von *l*.

8. Hat *c* selbst ein Kindknoten, wird dieser an die ehemalige Position von *c* kopiert und der ehemalige Kindknoten von *c* wird gelöscht.

9. Laufe den Suchpfad zurück (beginnend beim Vaterknoten von der ehemaligen Position von *c*), aktualisiere jeweils die Höhe und prüfe bei jedem Knoten des Aufwärtspfades (einschließlich der Wurzel des AVL-Baumes), ob eine Disbalance vorliegt.

10. Sollte eine Disbalance in *a* vorliegen, prüfe und rotiere nach folgenden Verfahren:

    a. Wenn die Löschung in *kL* erfolgte, betrachte die Balance von *kR*.
    Fall 1: Balance ist +1
    Vorgehen: Führe Linksrotation auf *a* aus
    Fall 2: Balance ist -1
    Vorgehen: Führe zunächst eine Rechtsrotation auf *kR* aus und anschließend eine Linksrotation auf *a*
    Fall 3: Balance ist 0
    Vorgehen: Führe Linksrotation auf *a* aus

    b. Wenn die Löschung in *kR* erfolgte, betrachte die Balance von *kL*.
    Fall 1: Balance ist +1
    Vorgehen: Führe zunächst eine Linksrotation auf *kL* aus und anschließend eine Rechtsrotation auf *a*
    Fall 2: Balance ist -1
    Vorgehen: Führe Rechtsrotation auf *a* aus
    Fall 3: Balance ist 0
    Vorgehen: Führe Rechtsrotation auf *a* aus

Anpassung der Knotenhöhen:
Die (neue) Höhe ergibt sich aus folgender Berechnung: max(li Kind, re Kind) +1

Aufgabe 3.3

Tests

Um die Richtigkeit und die Zuverlässigkeit des AVL-Baumes zu gewährleisten, sollen umfangreiche JUnit-Tests implementiert werden. Es muss jeder Knoten auf die richtige Balance geprüft werden.

Dateiname des Jar-Files: avlJUt.jar

<u>Mindestanforderung:</u>

1. Grenzfälle
    a. AVL-Baum mit 1 Knoten
    b. AVL-Baum mit keinem Knoten (isEmpty())
    c. Linksrotation erzwingen:
        - Einfügereihenfolge: 1,2,3
    d. Rechtsrotation erzwingen:
        - Einfügereihenfolge: 3,2,1
    e. Problemsituation links erzwingen:
        - Einfügereihenfole: 20,19,30,18,25,40,41,24,28,23
    f. Problemsituation rechts erzwingen:
        - Einfügereihenfolge: 20,15,21,22,14,18,13,17,19,16
2. Belastung: Einfügen
    a. AVL-Baum mit 500 Knoten
    b. AVL-Baum mit 1000 Knoten
    c. AVL-Baum mit 2000 Knoten
    d. AVL-Baum mit 4000 Knoten
    e. AVL-Baum mit 8000 Knoten
    f. AVL-Baum mit 16000 Knoten
3. Belastung: Löschen (bezogen auf 2.)
    a. 100 Elemente Löschen
    b. 1000 Elemente Löschen
    c. 1999 Elemente Löschen

Aufgabe 3.4 und Aufgabe 3.5

Versuchsaufbau:

Es wird eine Kopie des AVL-Baumes erstellt und um folgende Komponenten erweitert:

- Laufzeit (insert):
    - Zu Beginn des ersten insert() Aufrufs wird die aktuelle Zeit festgehalten. Am Ende des letztens insert() Aufrufs 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)
- Anzahl
    - Anzahl der linken bzw. rechten Rotationen
- Zugriffe
    - Lesen

      Ein Lesezugriff ist definiert, durch das Auslesen eines Knotenwertes bzw. dessen Höhe
    - Schreiben

      Ein Schreibzugriff ist definiert, durch das mutieren des Knotens (Wert, Höhe, linkes Kind sowie rechtes Kind)

Aufgabe 3.6

Messungen:

Mit den in Aufgabe 3.3 und 3.4 implementierten Versionen, sollen aussagekräftige Messungen durchgeführt werden.

Vorgaben für die Messungen:
Anzahl der Knoten mit unterschiedlichen Zahlenwerten im AVL-Baum:

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

Resultate:

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

AVL-Baum									
	500	1000	2000	4000	8000	16000	32000	64000	128000
Laufzeit (ms)									
Rotationen (links)									
Rotationen (rechts)									
Rotationen (gesamt)									
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.