SKIZZE

SKIZZE ZUR AUFGABE 3 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
27.11.15	2
29.11.15	2
Gesamt	**4**

Aktueller Stand
Die Skizze ist fertig.

Änderungen in der Skizze
-

Skizze:
Es sollen folgende Klassen in Java implementiert werden: Zahlengenerator, AVL-Baum und Benchmark (Package: „ADT“, Java Klassen: „NumberBuilder“, „AVLTree“, „Benchmark“). Dazu müssen auch JUnit Tests erstellt werden. (Package: „ADT/avltests“, extern als „avltJUt.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.

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

Funktionale Vorgabe
- Die Elemente sind vom Typ „ganze positive Zahl“.

Technische Vorgabe
- Die gewünschte Anzahl der Zufallszahlen muss ≥ 1 sein.
- Erstellung einer Datei mit dem Namen „zahlen.dat“.
- Besteht die Datei schon, wird sie folglich überschrieben.

Public Operationen (semantische Signatur/syntaktische Signatur)

generateSortNum: int → file

(Input: Integer count, Boolean duplicateEnabled | Output: File file)

Mit Hilfe dieser Methode erzeugen wir eine Datei mit der angegebenen Anzahl an Zufallszahlen. Die Variable `duplicateEnabled` entspricht der Möglichkeit, die zufälligen Zahlen mit oder ohne Duplikate zu erzeugen. Diese Variable wird auch bei den `sortNumLeft` und `sortNumRight` Methoden verwendet.

sortNumLeft: int → file

(Input: Integer count, Boolean duplicateEnabled | 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, Boolean duplicateEnabled | 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.

AVL Baum

Ein binärer Baum heißt AVL-Baum, wenn für jeden Knoten p gilt, dass sich die Höhen des linken und rechten Teilbaums höchstens um 1 unterscheiden.
In der Klasse „`AVLTree`“ sind die sämtlichen Methoden, wie z.B. insert, delete, high und rotate implementiert.

Funktionale Vorgabe

- Die Elemente des Baumes sind vom Datentyp Integer und grösser Null.
- Unterschied der Höhe zwischen linkem und rechtem Teilbaum darf nicht höher als 1 betragen.

Technische Vorgabe

- Intern wird der ADT mit Hilfe einer Liste rekursiv implementiert.

Objektmengen

- elem (das Element (Knoten oder Blatt) eines AVL-Baums), avltree (Das Objekt vom Typ AVLTree), png (die grafische Datei, die einem Baum entspricht)

Public Operationen (semantische Signatur/syntaktische Signatur):

create: ∅ → avltree

(Input: ∅ | Output: AVLTree avltree)

Mit Hilfe der Methode `create` erzeugen wir eine neue Instanz der Klasse `AVLTree`.

isEmpty: avltree → bool

(Input: AVLTree avltree | Output: Boolean isEmpty)

Mit `isEmpty` überprüfen wir die Höhe des Baumes und liefern einen Boolean Wert zurück. Es entspricht `true`, wenn es keine Elemente in einem Baum vorhanden sind und sonst `false`.

high: avltree → int

(Input: AVLTree avltree | Output: Integer height)

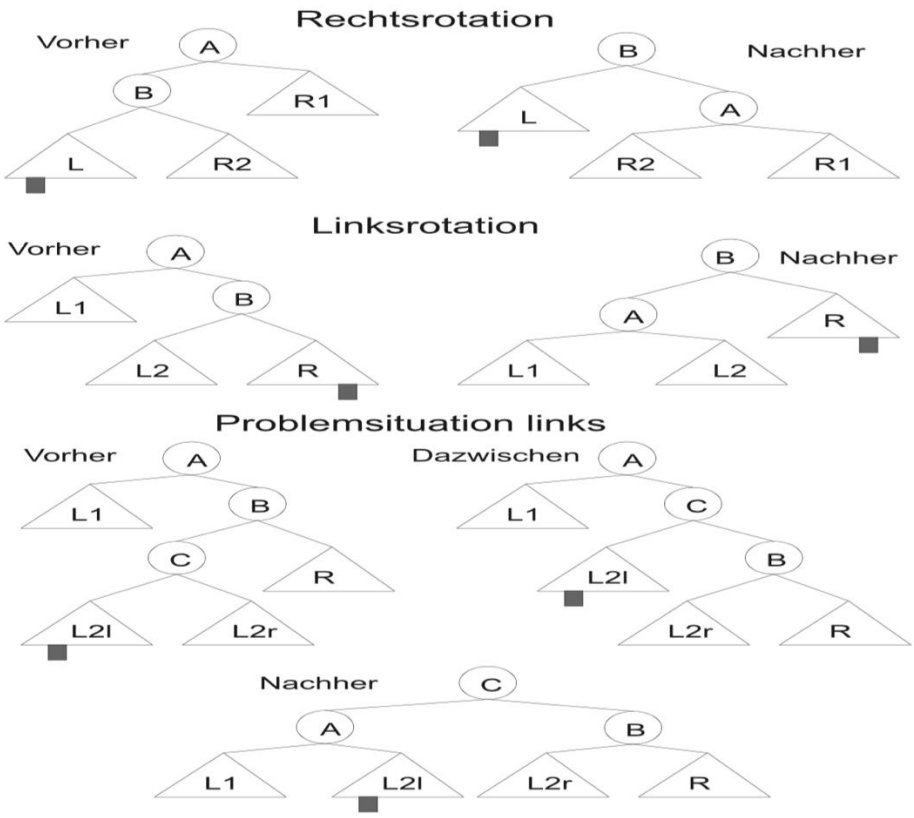
Die Länge des längsten Pfades eines AVL-Baumes wird ermittelt, indem wir uns jeweils die Teilbäume links und rechts von der Wurzel aus ansehen.

insert: avltree x elem→ avltree

(Input: AVLTree avltree, Integer elem | Output: AVLTree avltree)

Ist der Baum leer, wird das Element als Wurzel eingesetzt. Sonst gehen wir von der Wurzel des Baumes aus. Gibt es noch kein Element auf dem aktuellen Platz, wird ein neues Element dort eingefügt. Ist der Wert des einzufügenden Elementes kleiner des aktuellen Wertes, wird der linke Teilbaum rekursiv betrachtet, ansonsten der rechte Teilbaum. Dieses Vorgehen wird wiederholt bis ein leerer Platz gefunden wurde. Anschließend muss die Balance des Baumes überprüft werden. Der Balancefaktor beider Teilbäume darf nicht höher als 1 betragen. Ist es nicht der Fall, muss eine Rotation (links, rechts oder entsprechend die doppelte Rotation) stattfinden, damit die vorgegebene AVL Definition gewährleistet ist.

Das folgende Rotationsverhalten grafisch dargestellt:



delete: avltree x elem → avltree

(Input: AVLTree avltree, Integer elem | Output: AVLTree avltree)

Zuerst wird der ganze Baum nach dem eingegebenen Element durchsucht. Bei einer Löschung eines Elementes im Knoten des Baumes, kopieren wir entweder das linke Element des rechten Teilbaumes und fügen es an der zu löschenden Position ein oder das rechte Element des linken Teilbaumes. Die Auswahl der Elemente ist situationsbedingt abhängig, ist das rechte Blatt-Element nicht vorhanden, nehmen wir dann das linke Blatt-Element als Ersatz. Und umgekehrt. Wenn die beiden vorhanden sind, nehmen wir uns das linke Blatt-Element. Die Suche nach den Blättern erfolgt mit Hilfe der Bottom-up Methode. Ist ein Element ein Blatt des Teilbaumes, wird er entfernt und die Balance des Baumes wieder überprüft. Der Balancefaktor beider Teilbäume darf nicht höher als 1 betragen. Ist es nicht der Fall, muss eine Rotation (links, rechts oder entsprechend die doppelte Rotation) stattfinden, damit die vorgegebene AVL Definition gewährleistet ist.

print: avltree → png

(Input: AVLTree avltree | Output: File tree)

Diese Methode erzeugt dem Baum entsprechende png Datei mit dem Namen „baum.png“. Besteht die Datei schon, wird sie folglich überschrieben.

Benchmark

Benchmark ist unsere separate Klasse, die mit Hilfe von `generateSortNum` (auch `sortNumLeft` u. `sortNumRight`) Zufallszahlen generiert, diese importiert und auf denen `insert` Schritt für Schritt ausführt. Die Laufzeit der `insert`-Methode und die Anzahl der lesenden und schreibenden Zugriffe (inklusive Anzahl der Rotationen) auf die Elemente in dem AVL-Baum bei `insert`-Methode werden separat und automatisiert gemessen. Die Messungen enthalten die Fälle für 10, 50, 100, 500, 1000, 5000, 10000 und 20000 Zahlen in 3 Anordnungen: unsortierte Zahlen (zufällig geordnete), linkssortierte und rechtssortierte Zahlen.

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 benchmark)

Testen mit JUnit

- Alle obengenannten Methoden aufrufen.
- Es sollen folgende Situationen betrachtet werden: ein Baum besitzt keine Elemente, genau ein Element, genau zwei Elemente, mehrere Elemente.
- Die fehlerhaften Situationen sollen berücksichtigt werden: z.B. der übergebene Wert bei `insert` existiert bereits im Baum oder ist im Baum bei `delete` nicht enthalten usw.