AD- Aufgabe 3 - Entwurf

Inhaltsverzeichnis

AD- Aufgabe 3 - Entwurf

Allgemeines

Team: 01, Eugen Deutsch, Phillip Schackier

Aufgabenaufteilung:
Die Aufgabe wurde zusammen bearbeitet.
Quellenangaben: AD Skript, sowie Folien
Bearbeitungszeitraum:
28.11.15 - 1 Stunde: Beide
29.11.15 - 3 Stunden: Beide
30.11.15 - 2 Stunden: Beide
01.12.15 - 1 Stunde: Beide

Aktueller Stand: Der Baum und einige Testfälle wurden implementiert. Die Methode print fehlt bisher und die zählenden Varianten des Baumes sind noch nicht implementiert worden.
Änderungen in der Skizze: getLeft und getRight wurden hinzugefügt, print hat einen Parameter hinzubekommen und die Methode importNums wurde hinzugefügt.

Package-/Ordner-Hierarchie und Sichtbarkeiten

adtTree
----- generate
---------- Generator <public class>
---------- GeneratorUnique <public class>
----- AvlTree <public class>
----- AvlTreeTime <public class>
----- AvlTreeAccess <public class>
----- AvlTreeRotation <public class>
----- Benchmark <public class>
----- tests
---------- AvlTreeTest <public class>

Klassen/interface Hierarchien und Sichtbarkeiten

Es gibt keinerlei Vererbungen untereinander.

AD- Aufgabe 3 - Entwurf

Implementationen und Spezifikationen

Benchmark

- **public static void main(String** args**[])**: Hier werden die Tests bezüglich der Laufzeit, der Zugriffe und der Rotationen durchgeführt.

Der Aufbau sieht dabei wie folgt aus:
Zum testen der Laufzeit werden mit Hilfe von **GeneratorUnique** Dateien generiert mit einer jeweils unterschiedlichen Anzahl an Zahlen (z.B. 100, 1.000, 10.000...). Diese Zahlen werden importiert und dann in den Baum eingelesen. Ein insert liefert dabei die benötigte Zeit für das einfügen einer Zahl in den Baum. Bei z.B. 1.000 Zahlen wird die benötigte Zeit von den 1.000 insert's aufaddiert und als Ergebnis exportiert. Das Selbe wird für z.B. 10.000 Zahlen usw. wiederholt und zudem werden linksgeordnette, sowie rechtsgeordnette Zahlenmengen benutzt.

Die Rotationen, sowie die Zugriffe werden unter den gleichen Bedingungen überprüft.

Beim benutzen einer ADT einer anderen Gruppe werden keine neuen Zahlen generiert, um die gleichen Bedingungen zu gewährleisten.

Die Ergebnisse werden in eine *.csv Datei exportiert, wobei die Laufzeiten von Nanosekunden in Millisekunden umgerechnet werden.

Generator

- **public static void sortnum(int** amount**)**: Erstellt eine Datei mit der angegebenen Anzahl an Zufallszahlen, wobei Zahlen auch wiederholt auftauchen können. Die Zahlen liegen zwischen 0 und amount und sind in der Datei als Strings gespeichert und werden mit einem Leerzeichen getrennt. Die Datei heißt "zahlen.dat" und sofern diese schon existiert, wird sie überschrieben. Der Dateiname ist an einer einzigen Stelle im Code änderbar.

Beispiel: 5 3 5 2 0

Bedingungen: amount >= 0
- **public static void sortnumLeft(int** amount**)**: Wie sortnum, allerdings sind die generierten Zahlen von links nach rechts geordnet in der Datei.

Beispiel: 0 2 3 5 5
- **public static void sortnumRight(int** amount**)**: Wie sortnum, allerdings sind die generierten Zahlen von rechts nach links geordnet in der Datei.

Beispiel: 5 5 3 2 0
- **public static int[] importNums()**: Importiert die Dateien aus “zahlen.dat” in ein Array und gibt dieses zurück.

GeneratorUnique

Wie **Generator**, nur das hier die Methoden keine Duplikate erzeugen. **ImportNums** ist hier nicht vertreten.

AvlTree

- **public static AvlTree create()**: Factory Methode zur Erstellung des Baumes.

- **public boolean isEmpty()**: True, wenn der Baum leer ist, sonst false.

- **public int high()**: Gibt die Höhe des Baumes zurück, wobei diese wie folgt definiert ist: Ein Blatt hat die Höhe 1, der Teilbaum der dieses Blatt als Nachfolger hat, hat die Höhe 2 und der darüberliegende Baum entsprechend die Höhe 3. Auf diese Weise lässt sich bis zur Wurzel ermitteln, welche Höhe der jeweilige Knoten hat.

- **public void insert(int** elem**)**:
  Fügt das Element dem Baum hinzu, wobei folgendermaßen vorgegangen wird:

  1) Ist das Element kleiner/gleich dem eigenen Element?
     a) Ja -> Habe ich einen Linken Teilbaum?
        - Ja -> Führe die Schritte beginnend bei 1 für diesen durch.
        - Nein -> Erstelle einen neuen Linken Teilbaum, der elem als Wert hat.
     b) Nein -> Habe ich einen Rechten Teilbaum?
        - Ja -> Führe die Schritte beginnend bei 1 für diesen durch.
        - Nein -> Erstelle einen neuen Rechten Teilbaum, der elem als Wert hat.
  2) Berechne und prüfe die Balance ausgehend vom Vorgänger des eingefügten Teilbaums bis hoch zur Wurzel.

AD- Aufgabe 3 - Entwurf

Die Balance wird dabei folgendermaßen sichergestellt:

1) Jeder Teilbaum weiß seine Balance (Rechter_Baum.höhe-Linker_Baum.höhe) und diese wird bei einem **insert** und einem **delete** aktualisiert.
2) Prüfe ob einer der folgenden Fälle besteht:
   a) Der obere Baum hat eine Balance von -2 und der untere -1 -> Rechts-Rotation
   b) Der obere Baum hat eine Balance von +2 und der untere +1 -> Links-Rotation
   c) Der obere Baum hat eine Balance von -2 und der untere +1 -> Doppelte Linksrotation
   d) Der obere Baum hat eine Balance von +2 und der untere -1 -> Doppelte Rechtsrotation

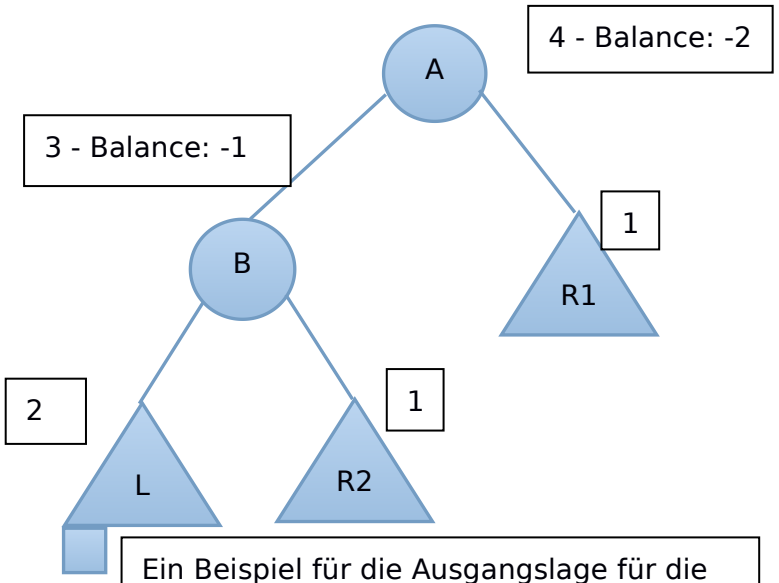
3) Die Rotationen funktionieren nach dem folgenden Prinzip:

rechts:
tempRight = B.right
B.right = A
A.left = tempRight

links:
tempLeft = B.left
B.left = A
A.right = tempLeft

doppel links:
B.rotateLeft
A.rotateRight

doppel rechts:
B.rotateRight
A.rotateLeft



Ein Beispiel für die Ausgangslage für die Rechtsrotation. Die Benennung wurde den Folien, bzw. dem Skript entnommen.

Beim Rotieren, wie beim Löschen ist es wichtig sicherzugehen, dass z.B. der Knoten A nicht unbedingt die Wurzel ist. Es muss somit sichergestellt werden, dass z.B. beim Tausch der Vorgänger von A die Referenz ändert und nicht einfach nur die Teilbäume von A dem Ersatz übergeben werden.

AD- Aufgabe 3 - Entwurf

- **public void delete(int** elem**)**:

Löscht das Element aus dem Baum und geht dabei wie folgt vor:

1) Suche elem und merke dir, welcher Teilbaum dieses hat. (Suche: Habe ich das Element? Wenn nicht, dann prüfe meine Teilbäume, sonst gebe mich zurück)
2) Wie viele Kinder hat der zu löschende Teilbaum?
   a) 0 -> Lösche es, bzw sorge dafür, dass niemand mehr eine Referenz zu diesem Teilbaum hat.
   b) 1 -> Ersetze den Teilbaum durch sein Kind und führe die Schritte ab 1 für das Kind aus.
   c) 2 -> Suche den Teilbaum mit dem maximalen Wert im linken Teilbaum (left.right.right.right... bis null käme) und tausche die Werte aus. Führe dann die Methode ab 1 beim gefundenen Teilbaum durch.
3) Berechne und prüfe die Balance ausgehend vom Vorgänger des gelöschten Teilbaums bis hoch zur Wurzel wie bei **insert**.

Bedingung: tree.has(elem)

- **public void print(String** filename**)**: Speichert den Graphen als *.png Datei ab.
Bedingung: filename != null, GraphViz installiert

- **public AvlTree getLeft()** und **public AvlTree getRight()**: Da die Struktur eines Baumes darüber entscheidet, ob es sich überhaupt um einen Avl-Baum handelt oder nicht, ist es sehr wichtig diese in den Tests bei verschiedenen Szenarien zu überprüfen. Die Richtigkeit der Struktur wird anhand der Balance geprüft, die sich anhand der Höhe des rechten - die Höhe des linken Teilbaums ergibt, wobei man diesen Test bei allen Unterbäumen durchführen muss. Anhand der vorgegebenen Spezifikation lässt sich die Struktur nur anhand der Ausgabe von **print** überpfrüfen, wobei ein solcher Test nicht automatisierbar ist.
Deswegen ist ein Zugriff von außen auf den linken und rechten Teilbaum wichtig.

- **public boolean equals(Object** o**)**: Wichtig für die Tests, wobei hierbei nicht geprüft wird, ob die Struktur des Baumes richtig ist. Es werden lediglich die Elemente verglichen und ihre Position im Baum macht keinen Unterschied.

AD- Aufgabe 3 - Entwurf

AvlTreeTime

Wie **AvlTree**, nur das hier beim insert die Zeit zurückgegeben wird (**public long insert(int** elem**)**). Die Zeit wird in ns gemessen.

AvlTreeAccess

Wie **AvlTree**, nur das hier beim insert die Anzahl der Zugriffe auf die Elemente gezählt wird.

AvlTreeRotation

Wie **AvlTree**, nur das hier beim insert die Anzahl der Rotationen gemessen wird. Eine Doppelrotation zählt doppelt.

Fehlerbehandlung

Der Baum ist in dieser Form nicht darauf ausgelegt, dass bereits vorhandene Elemente eingefügt werden. Zu Lernzwecken wird dieser Fall nicht blockiert.

Ansonsten werden fehlerhafte Eingaben angenommen und ignoriert. Die entsprechende Methode führt ihren Dienst nicht aus und hinterlässt alles beim alten.

Tests

Getestet werden vor allem die Methoden **insert** und **delete**, wobei hier zum einen geprüft wird, ob die Reihenfolge der einzufügenden Elemente einen Unterschied macht und zudem wird mit Hilfe der zusätzlichen Methoden **getLeft** und **getRight** geprüft, ob die Struktur jederzeit einem Avl-Baum entspricht. **high** und **isEmpty** wird ebenfalls getestet, lässt sich aber ziemlich einfach abhandeln.
Mit den Tests handeln wir unter anderem folgende Situationen ab:

- leerer Baum
- Baum mit einem Element
- Baum mit 3 Elementen ohne Rotation
- Baum mit 3 Elementen mit Rotation
- Baum mit doppelrotation
- Alle Rotationsarten
- Löschen der Wurzel
- Löschen eines Blattes
- Löschen eines Baumes mit einem Kind
- Löschen eines Baumes mit zwei Kindern

AD- Aufgabe 3 - Entwurf

Export

Alles zusammen: ad_avl.jar
Tests: ad_AvlTreeTest.jar
Klassen: ad_AvlTree.jar, ad_AvlTreeTime.jar, AvlTreeAccess.jar, AvlTreeRotation.jar, ad_Benchmark.jar