



Aufgabenblatt 11

letzte Aktualisierung: 12. February, 09:37 Uhr
(6e9d53946a9fa5c262889dc822fcaeb95a81da95)

Ausgabe: Freitag, 02.02.2018
Abgabe: spätestens Mittwoch, 14.02.2018, 18:00

Thema: AVL-Bäume

Abgabemodalitäten

- Alle abzugebenden Quelltexte müssen ohne Warnungen und Fehler auf den Rechnern des eecsIT mittels `gcc -std=c99 -Wall` kompilieren.
 - Für die Hausaufgaben im Semester gibt es ein eigenes Subversion-Repository (SVN). Bitte die Hinweise auf Blatt 1 dazu beachten!
 - Die Abgaben für Blatt 1 bis 4 erfolgen als Einzelabgaben. Ab Blatt 5 erfolgen Abgaben in festen Gruppen à 2 Personen.
 - Die Gruppen werden vor Ausgabe von Blatt 5 gebildet. Solltet ihr keine Gruppe gebildet haben, werdet ihr einer Gruppe zugewiesen.
 - Alle Abgaben müssen an der dafür vorgesehenen Stelle die Namen aller Gruppenmitglieder bzw. bei Einzelabgaben des Autors/der Autorin enthalten!
 - Die Abgabe erfolgt ausschließlich über unser SVN im Abgaben-Ordner. Die finale Abgabe
 - für Gruppenabgaben erfolgt im Unterordner
Tutorien/t<xx>/Gruppen/g<xx>/Abgaben/Blatt<xx>/
 - für Einzelabgaben erfolgt im Unterordner
Tutorien/t<xx>/Studierende/<tuBIT-Login>/Abgaben/Blatt<xx>/
- WICHTIG:** Die Abgabeordner werden immer von uns erstellt, sonst kommt es zu Konflikten! Benutzt zum Aktualisieren `svn up` im obersten Verzeichnis des Repositories!
- Benutzt für alle Abgaben die in der jeweiligen Aufgabe angegebenen Dateinamen (die von uns vorgegebenen Dateien haben das Wort "vorgabe" im Dateinamen, du musst die Datei mit deiner Lösung also entsprechend umbenennen.)
 - Gib bei den Programmieraufgaben den C-Quellcode ab und achte darauf, dass die Datei mit .c endet
 - Bei Textaufgaben sind, wenn nicht anders angegeben, .txt Dateien zugelassen, die als Plaintext-Datei zu speichern sind. (Keine Word-Dateien umbenennen, etc.!)

Definitionen

Abweichend von der Definition bei binären Suchbäumen, verwenden wir zur Bestimmung der Höhe im Kontext der AVL-Bäume folgende Definition:

Definition: Die Höhe eines Blattes (d.h. der Knoten hat keine Kinder) im AVL-Baum ist 1. Anderenfalls ist die Höhe eines (inneren) Knotens x (d.h. x hat mindestens ein Kind) definiert als:

$$\text{Höhe}(x) = 1 + \max(\text{Höhe}(l[x]), \text{Höhe}(r[x]))$$

In AVL-Bäumen werden Balancierungs-Operationen an einem Knoten x nur angewandt, wenn die Höhen der Kinder ($l[x], r[x]$) um mehr als 1 unterscheiden. Wir benutzen auch folgende Definition:

Definition: Der Balance-Wert eines Knoten ist die Baumhöhe des linken Kindes minus der Baumhöhe des rechten Kindes.

Ein AVL-Baum ist somit balanciert, sofern als Balance-Werte nur -1 , 0 , und 1 vorkommen.

1. Aufgabe: Handsimulation AVL-Baum (1 Punkt)

In dieser Aufgabe soll die Einfügen-Operation auf AVL Bäumen geübt werden.

Füge die Elemente 100, 50, 25, 10, 37, 32, 200 nacheinander in einen initial leeren AVL-Baum ein und dokumentiere alle Zwischenzustände des Baumes sowie die benötigten Operationen, um den Baum zu balancieren. Die Abgabe erfolgt im SVN unter der Datei `introprog_build_avl.txt`. Die auszufüllende Vorgabe findest du im SVN unter `introprog_build_avl_vorgabe.txt`. Die Syntax zur Darstellung der Bäume ist dabei analog zu vorherigen Aufgaben:

- Jeder Knoten im Baum wird nummeriert.
- Die Nummerierung weist der Wurzel den Index 1, seinen Kindern die Werte 2 (links) und 3 (rechts) zu usw. (siehe Abbildung 1).
- Um zu beschreiben, dass der Knoten mit dem Index 5 den Wert 11 enthält, verwenden wir die Notation "5:11".
- Die Angabe mehrerer Elemente erfolgt jeweils auf einer eigenen Zeile.
- Im Unterschied zu den binären Suchbäumen aus Aufgabenblatt 6 verwenden wir an dieser Stelle sowohl für die Indizes als auch für die Werte der Knoten Zahlen. Beachtet also, dass der erste Wert den Index repräsentiert und der zweite den Wert.
- Zusätzlich führen wir in diese Aufgabe noch den Balancewert eines Knotens in [] mit.
- Weiterhin soll die durchgeführte Operation im Multiple Choice Formular angegeben werden. Gebt also an, ob es sich um das Einfügen eines Wertes, eine Rechtsrotation oder eine Linksrotation handelt. Doppelrotationen werden hier in zwei Schritten als Kombination von Rechts- bzw. Linksrotationen dargestellt. Achtet darauf, dass ihr bei Doppelrotationen die jeweiligen Rotationen in der richtigen Reihenfolge gemäß des Pseudo-Codes aus der Vorlesung durchführt.
- Abbildung 2 zeigt den schrittweisen Aufbau eines AVL-Baumes nach Einfügen der Zahlen 23, 55, 11, 9, 10. Die Darstellung dieses Baumes im abzugebenden Format findet ihr in der Datei `introprog_build_avl_beispiel.txt`

Hinweis: Das Textformat ist nur zur Abgabe in Osiris gedacht. Die eigentliche Aufgabe besteht darin, die AVL-Bäume zu zeichnen und nach und nach aufzubauen. Wir empfehlen euch daher, die Aufgabe wirklich durch Zeichnen zu lösen und am Ende eure gezeichneten Schritte in das vorgestellte Textformat zu übersetzen. Zeichnet dazu den Baum jeweils nach dem Einfügen des Elements sowie nach jeder benötigten Rotation. Notiert für jeden Baum (auch jeden Zwischenzustand) den entsprechenden Balance-Wert an jedem Knoten und benennt jeweils die durchgeführten Rotationen analog zu dem Beispiel in Abbildung 2.

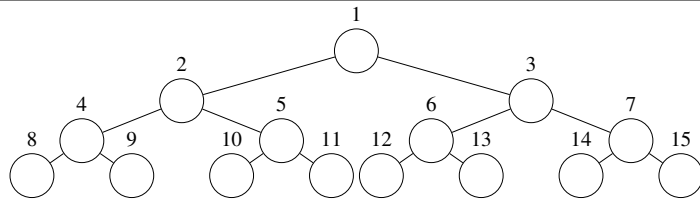


Abbildung 1: Nummerierung der Knoten im Baum

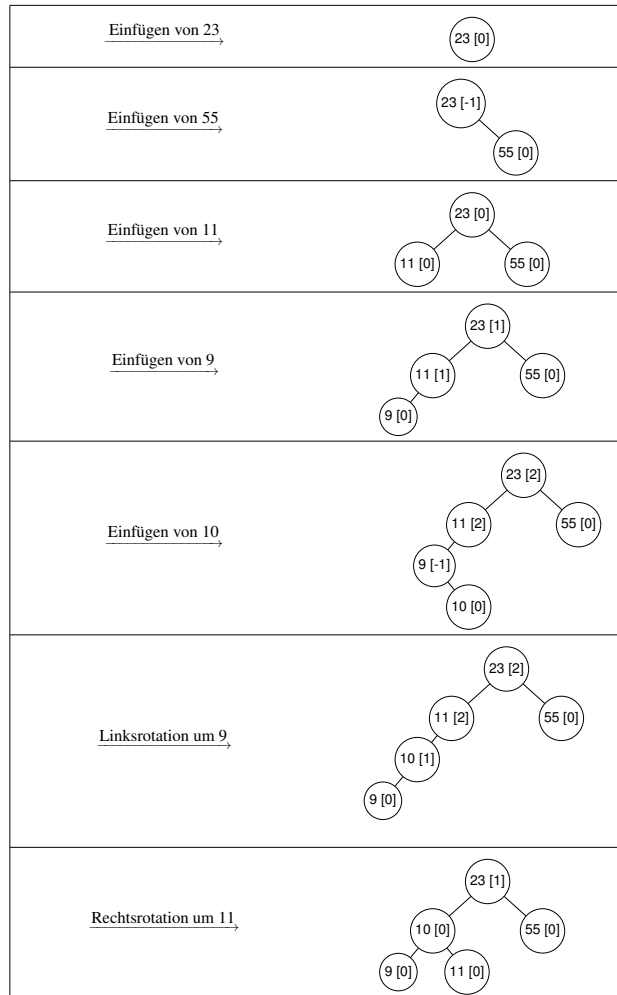


Abbildung 2: Beispiel zum Aufbau eines AVL-Baumes mit Blancewerten

<pre> === SCHRITT 1 === [x] Einfügen: 23 [] Linksrotation: [] Rechtsrotation: [] Nichts mehr zu tun Resultierender Baum: 1:23[0] === ENDE SCHRITT 1 === === SCHRITT 2 === [x] Einfügen: 55 [] Linksrotation: [] Rechtsrotation: [] Nichts mehr zu tun Resultierender Baum: 1:23[-1] 3:55[0] === ENDE SCHRITT 2 === === SCHRITT 3 === [x] Einfügen: 11 [] Linksrotation: [] Rechtsrotation: [] Nichts mehr zu tun Resultierender Baum: 1:23[0] 2:11[0] 3:55[0] === ENDE SCHRITT 3 === === SCHRITT 4 === [x] Einfügen: 9 [] Linksrotation: [] Rechtsrotation: [] Nichts mehr zu tun Resultierender Baum: 1:23[1] 2:11[1] 3:55[0] 4:9[0] === ENDE SCHRITT 4 === === SCHRITT 5 === [x] Einfügen: 10 [] Linksrotation: [] Rechtsrotation: [] Nichts mehr zu tun Resultierender Baum: 1:23[2] 2:11[2] 3:55[0] 4:9[-1] 9:10[0] === ENDE SCHRITT 5 === </pre>	<pre> === SCHRITT 6 === [] Einfügen: [x] Linksrotation: 9 [] Rechtsrotation: [] Nichts mehr zu tun Resultierender Baum: 1:23[2] 2:11[2] 3:55[0] 4:10[1] 8:9[0] === ENDE SCHRITT 6 === === SCHRITT 7 === [] Einfügen: [] Linksrotation: [x] Rechtsrotation: 11 [] Nichts mehr zu tun Resultierender Baum: 1:23[1] 2:10[0] 3:55[0] 4:9[0] 5:11[0] === ENDE SCHRITT 7 === === SCHRITT 8 === [] Einfügen: [] Linksrotation: [] Rechtsrotation: [x] Nichts mehr zu tun Resultierender Baum: === ENDE SCHRITT 8 === === SCHRITT 9 === [] Einfügen: [] Linksrotation: [] Rechtsrotation: [x] Nichts mehr zu tun Resultierender Baum: === ENDE SCHRITT 9 === === SCHRITT 10 === [] Einfügen: [] Linksrotation: [] Rechtsrotation: [x] Nichts mehr zu tun Resultierender Baum: === ENDE SCHRITT 10 === </pre>
---	--

Abbildung 3: Abgabeformat für das Beispiel aus Abbildung 2

2. Aufgabe: Implementieren eines AVL-Baumes (4 Punkte)

In dieser Aufgabe soll ein AVL-Baum implementiert werden. Das resultierende Programm soll dabei die Eingabe entweder von der Konsole oder aus einer Datei einlesen. Folgende Befehle sollen zur Verfügung stehen:

<z> Die Zahl **<z>** soll in den Suchbaum eingefügt werden.

p Gibt den gesamten AVL-Baum aus.

w Gibt alle Knoten in "in-order" Reihenfolge aus.

c Gibt die Anzahl an enthaltenen Knoten aus.

q Beendet das Programm.

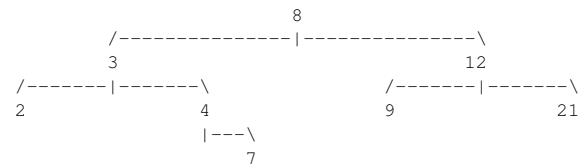
Die Eingabedatei aus Listing 1 führt zur Ausgabe in Listing

Listing 1: Eingabedatei_avl.txt

```
12
2
8
3
21
9
4
7
p
w
c
q
```

Listing 2: Ausgabe des Programms unter der Eingabe der Datei Eingabedatei_avl.txt

```
Füge 12 ein...
Füge 2 ein...
Füge 8 ein...
Füge 3 ein...
Füge 21 ein...
Füge 9 ein...
Füge 4 ein...
Füge 7 ein...
```



```
2 3 4 7 8 9 12 21
Im AVL-Baum sind 8 Elemente enthalten.
```

Während wiederum eine main-Funktion sowie das Parsen der Eingabe von uns bereitgestellt werden, müssen die Funktionen aus Listing 3 implementiert werden.

Listing 3: introprog_avl_vorgabe.c

```
#include <stdlib.h>
#include <stdio.h> //Ein- / Ausgabe
#include <math.h> //Für die Berechnungen der Ausgabe
#include "avl.h"

// Gibt den gesamten AVL Baum in "in-order" Reihenfolge aus.
void AVL_in_order_walk(AVLTree* avlt)
{
    //Hier Code implementieren!
}

// Diese Funktion führt eine Linksrotation auf dem angegebenen Knoten aus.
// Beachtet: Die Höhen der entsprechenden Teilbäume müssen (lokal)
// angepasst werden.
void AVL_rotate_left(AVLTree* avlt, AVLNode* x)
{
    //Hier Code implementieren!
}

// Diese Funktion führt eine Rechtsrotation auf dem angegebenen Knoten aus.
// Beachtet: Die Höhen der entsprechenden Teilbäume müssen (lokal)
// angepasst werden.
void AVL_rotate_right(AVLTree* avlt, AVLNode* y)
{
    //Hier Code implementieren!
}

//Balanciere den Teilbaum unterhalb von node.
void AVL_balance(AVLTree* avlt, AVLNode* node)
{
    //Hier Code implementieren!
}

// Fügt der Wert value in den Baum ein.
// Die Implementierung muss sicherstellen, dass der Baum nach dem Einfügen
// immer noch balanciert ist!
void AVL_insert_value(AVLTree* avlt, int value)
{
    //Hier Code implementieren!
}

// Löscht den gesamten AVL-Baum und gibt den Speicher aller Knoten frei.
void AVL_remove_all_elements(AVLTree* avlt)
{
    //Hier Code implementieren!
}
```

2.1. Ausgabe von Elementen im AVL-Baum

Implementiere die Funktion `void AVL_in_order_walk(AVLTree* avlt)`, welche sämtliche Werte im binären Suchbaum in “in-order” Reihenfolge auf `stdout` (der Konsole) ausgibt. Beachte, dass bei der “in-order” Reihenfolge, die Elemente immer aufsteigend geordnet sind. Die Elemente sollen durch ein einzelnes Leerzeichen getrennt sein. Nach der Ausgabe aller Elemente muss ein einfacher Zeilensprung (`\n`) folgen (siehe auch Listing 3).

2.2. Links- und Rechtsrotation

Implementiere die Funktionen `void AVL_rotate_left(AVLTree* avlt, AVLNode* x)` und `void AVL_rotate_right(AVLTree* avlt, AVLNode* y)`, welche jeweils die AVL Links- und Rechtsrotation auf dem Knoten `x` bzw. `y` ausführen.

Hinweis: Achte darauf, dass die Pointer richtig vertauscht werden, und dass nach der jeweiligen Rotation in allen Knoten die gespeicherten Baumhöhen stimmen. Beachte weiterhin, dass eventuell der Wurzelknoten des gesamten Baumes angepasst werden muss. Diese Funktionen müssen jeweils eine konstante Laufzeit haben.

2.3. Wiederherstellen der Balance eines AVL-Baumes

Implementiere die Funktionen `void AVL_balance(AVLTree* avlt, AVLNode* node)`, welche den AVL-Baum am Knoten `node` (gegebenenfalls) balanciert.

Hinweis: Diese Funktion muss eine konstante Laufzeit haben. Es soll also wirklich nur der übergebene Knoten balanciert werden.

2.4. Einfügen in den AVL-Baum

Implementiere unter Verwendung der vorherigen Funktionen die Funktion `void AVL_insert_value(AVLTree* avlt, int value)`, welche den Wert `value` in den Baum `avlt` einfügt.

Hinweis: Achte darauf, dass insbesondere die `parent` Pointer richtig gesetzt sind, dass die Suchbaum-Eigenschaft erhalten bleibt und dass der Baum nach dem Einfügen überall balanciert ist. Diese Funktion muss eine Laufzeit von $\mathcal{O}(\log n)$ haben.

2.5. Korrekte Freigabe des Speichers

Implementiere abschließend die Funktion `void AVL_remove_all_elements(AVLTree* avlt)`, welche alle Knoten im Baum löscht. Implementiere die Funktion dabei so, dass der Suchbaum nur einmal durchlaufen wird: Gegeben einen Knoten im Baum wird zuerst der rechte Unterbaum, dann der linke Unterbaum und anschließend der aktuelle Knoten gelöscht. Diese Art der Traversierung wird auch als “post-order” bezeichnet.

Gib den Quelltext in einer Datei mit folgendem Namen ab: `introprog_avl.c`

Hinweis: Kompiliere dein Programm mit `gcc -Wall -std=c99 main_avl.c avl.c introprog_avl.c -lm -g -o introprog_avl`