Fakultät IV - Elektrotechnik und Informatik
Einführung in die Programmierung WS 2014/15
Feldmann / Rost / Streibelt / Lichtblau / Deimel / Lucas

Aufgabenblatt 9

letzte Aktualisierung: 22. Januar, 13:54 Uhr

(a78617c36f3e9efc7b2c7aefcb85f7f26d3762a4)

Ausgabe: Mittwoch, 14.01.2015
Abgabe: spätestens Freitag, 23.01.2015, 18:00

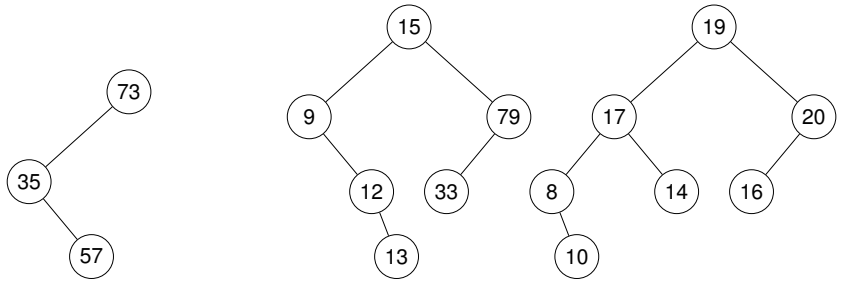
Thema: *AVL Bäume*

Abgabemodalitäten

- Alle abzugebenden Quelltexte müssen ohne Warnungen und Fehler auf den Rechnern des tubIT/IRB mittels `gcc -std=c99 -Wall` kompilieren.
- Abgaben erfolgen prinzipiell immer in Gruppen à 2 Personen, welche in den Tutorien festgelegt wurden. Einzelabgaben sind explizit als solche gekennzeichnet.
- Die Abgabe erfolgt ausschließlich über SVN. Die finale Abgabe
  - für Gruppenabgaben erfolgt im Unterordner `Tutorien/t<xx>/Gruppen/g<xx>/Blatt<xx>/submission/`
  - für Einzelabgaben erfolgt im Unterordner `Tutorien/t<xx>/Studierende/<tuBIT-Login>/Blatt<xx>/submission/`
- Benutzen Sie für alle Abgaben von Programmcode das folgende Namensschema: `introprog_blatt0X_aufgabe0Y_Z.c`, wobei X durch die Blattnummer, Y durch die Aufgabe und Z durch die Unteraufgabe ersetzen ist.
  Beispiel: Aufgabe 1.2 wird zu: `introprog_blatt01_aufgabe01_2.c`
  Für jede Unteraufgabe geben Sie maximal eine Quellcodedatei ab, es sei denn, die Aufgabenstellung erfordert explizit die Abgabe mehrerer Dateien pro Aufgabe.

  Alle anderen Abgaben (Pseudocode, Textaufgaben) benennen Sie wie oben beschrieben. Die zugelassenen Abgabeformate sind PDF, ODT und Text (txt). Auch hier verwenden Sie eine Datei pro Aufgabe, nicht jedoch pro Unteraufgabe.

1. Aufgabe: Eigenschaften balancierter Binärbäume (12 Punkte)



(a) Baum A (b) Baum B (c) Baum C

Abbildung 1: Balancierte und unbalancierte Binärbäume

1.1. Bestimmen der Baumhöhe und Balance (6 Punkte)

Bestimmen Sie die Baumhöhen jedes Unterbaumes und den Balance-Wert jedes Knotens. Welche Bäume sind balanciert, welche nicht? Erklären Sie ihre Schlußfolgerung mit Hilfe der Definitionen:

Definition: `Die Baumhöhe eines Baumes ist die Maximalzahl der Knoten die besucht werden müssen, um von der Wurzel zu einem Kinder-Knoten zu gelangen. Eine Wurzel ohne Kinder hat die Baumhöhe 1, eine nichtexistente Wurzel die Baumhöhe 0.`

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

Definition:

1. `Jeder Baum der Höhe 0 (ohne jegliche Knoten) ist balanciert.`
2. `Wenn alle Unterbäume balanciert sind, und die Baumhöhe des linken Kindes sich von der des rechten Kindes um nicht mehr als 1 unterscheidet, dann ist der Baum balanciert.`
3. `Ansonsten ist der Baum nicht balanciert.`

Geben Sie Ihre Lösung der Teilaufgabe in einem Textdokument mit einem der folgenden Namen ab: `introprog_blatt09_aufgabe01_1.{txt|odt|pdf}`

1.2. Rebalancieren eines Baumes (6 Punkte)

AVL-Bäume benutzen zwei Operationen (Linksrotation und Rechtsrotation), um unbalancierte Binärbäume in balancierte Binärbäume zu überführen. Geben Sie für alle Bäume an, in welcher Reihenfolge welche Knoten rotiert werden müssen, um einen balancierten AVL Baum zu erhalten. Wie sehen die Bäume nach jeder Rotation aus?

Geben Sie Ihre Lösung der Teilaufgabe in einem Textdokument mit einem der folgenden Namen ab: `introprog_blatt09_aufgabe01_2.{txt|odt|pdf}`

Hinweis: Es gibt viele Programme, die zur einfachen Visualisierung von Bäumen verwendet werden können, z.B. OpenOffice Draw, Microsoft Powerpoint, Microsoft Visio, Inkscape, Graphviz, usw.

2. Aufgabe: Implementieren eines AVL Baumes (28 Punkte)

Ihr Programm soll einzelne Werte von der Standardeingabe einlesen, und diese in einem AVL Baum speichern. Implementieren Sie dazu die Links- und Rechtsrotation, sowie das Balancieren als Funktionen auf einem Binärbaum. Das Eingabeformat zu Erstellung des Baumes ist dem der vorigen Übungsaufgabe ähnlich:

Listing 1: Eingabeformat (`Eingabedatei_Blatt09_01`)

```
+ 33
+ 22
+ 11
+ 5
+ 1
+ 100
+ 110
+ 143
+ 54040
p -2
p -1
```

`+ X` Die Zahl **`X`** soll in den Suchbaum eingefügt werden.

`p -1` Es sollen alle Zahlen im Suchbaum in 'in-order' Reihenfolge (linker Unterbaum, Elternknoten, rechter Unterbaum) in einer Zeile ausgegeben werden.

`p -2` Der Baum soll auf korrekte Balancierung getestet werden. Ist der Baum balanciert, gibt das Programm in einer Zeile `1` zurück, ansonsten `0`.

Hinweis: Sie dürfen Quelltextfragmente aus ihrer vorherigen Übungsabgabe bei der Implementierung wiederverwenden.

Tipp: Die Codevorgabe enthält die Funktionen `AVL_is_balanced_node(AVLNode* node)` und `AVL_test_balance(AVLTree* avlt)`, mit deren Hilfe Sie testen können, ob ihr Baum (oder bestimmte Unterbäume) balanciert ist.

2.1. main-Methode (3 Punkte)

Schreiben Sie die main-Methode, so dass Eingabedateien richtig geparst werden und die entsprechenden Funktionen aufgerufen werden:

```
AVL_test_balance(AVLTree* avlt)
AVL_in_order_walk(AVLTree* avlt)
AVL_insert_value(AVLTree* avlt, int value)
```

2.2. Ausgabe von Elementen im AVL Baum (2 Punkte)

Implementieren Sie die Funktion `AVL_in_order_walk(AVLTree* avlt)`, welche sämtliche Werte im binären Suchbaum in 'in-order' Reihenfolge auf `stdout` (der Konsole) ausgibt. Beachten Sie, dass bei 'in-order' Reihenfolge die Elemente immer aufsteigend geordnet sind. Die Elemente sollen durch ein einzelnes Leerzeichen getrennt sein; beenden Sie die Ausgabe auf der Kommandozeile durch einen Zeilensprung (`\n`).

2.3. Links- und Rechtsrotation (8 Punkte)

Implementieren Sie die Funktionen `void AVL_rotate_left(AVLNode* node)` und `void AVL_rotate_right(AVLNode* node)` welche jeweils die AVL Links- und Rechtsrotation auf dem Knoten ausführen. Achten Sie darauf, dass die Pointer richtig vertauscht werden, und dass in allen Knoten die gespeicherten Baumhöhen stimmen.

2.4. Wiederherstellen der Balance eines AVL Baumes (6 Punkte)

Implementieren Sie die Funktionen `AVL_rebalance(AVLTree* avlt, AVLNode* node)` welche den AVL Baum nach Einfügen des Knotens node rebalanciert. Benutzen Sie dabei das Wissen, dass nur die Eltern des eingefügten Knotens balanciert werden müssen, um die Lauftzeitkomplexitätsklasse der Einfügeoperation ($O(\log n)$) nicht zu verändern.

Hinweis: Da `AVL_test_balance(AVLTree* avlt)` eine Laufzeitkomplexität von $O(n)$ besitzt, kann es hier nicht verwendet werden!

2.5. Einfügen in den AVL Baum (6 Punkte)

Implementieren Sie die Funktion `AVL_insert_value(AVLTree* avlt, int value)`, welche den Wert `value` in den Baum `avlt` einfügt. Achten Sie darauf, dass insbesondere die `parent` Pointer richtig gesetzt sind und dass die Suchbaumeigenschaften erhalten bleiben: Im linken Teilbaum sind nur Werte enthalten, die kleiner gleich dem aktuellen Wert sind und der rechte Teilbaum enthält nur Werte, die größer als der aktuelle Wert sind; außerdem muss der Baum balanciert sein.

2.6. Korrekte Freigabe des Speichers (3 Punkte)

Implementieren Sie die Funktion `AVL_delete_tree(AVLTree* avlt)`, welche alle Knoten im binären Suchbaum löscht. Implementieren Sie 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. Nutzen Sie diese Funktion in Ihrer `main()`-Methode, damit der zugewiesene Speicher explizit freigegeben wird.

Geben Sie den Quelltext in einer Datei mit folgendem Namen ab:
`introprog_blatt09_aufgabe02.c`

Listing 2: Codevorgabe `introprog_blatt09_aufgabe02_vorgabe.c`

```c
#include <stdio.h>
#include <stdlib.h>

// Knoten im AVL Suchbaum, wobei ...
// - left und right auf das linke bzw. rechte Kind zeigt.
// - parent auf denjenigen Knoten verweist, dessen Kind dieser Knoten ist.
// - value der ganzzahlige Wert des Knotens ist.
// - height die Höhe des Baumes unter diesem Knoten (ist immer >= 1)
// Diese Struktur darf nicht verändert werden.
struct AVLNode
{
   struct AVLNode* left;
   struct AVLNode* right;
   struct AVLNode* parent;
   int value;
```

```
16    int height;
17 };
18
19 // Ein Binärer Suchbaum mit einem Wurzelknoten root und der Anzahl an
20 // Elementen im Baum.
21 struct AVLTree
22 {
23    struct AVLNode* root;
24    int numberOfElements;
25 };
26
27 // typedefs, damit man sich das "struct" sparen kann.
28 typedef struct AVLNode AVLNode;
29 typedef struct AVLTree AVLTree;
30
31
32 // Gibt den Unterbaum von node in "in-order" Reihenfolge aus.
33 // Die Ausgabe dieser Funktion muss aufsteigend soriert sein.
34 void AVL_in_order_walk_node(AVLNode* node)
35 {
36    // Hier bitte Ihr Code.
37 }
38
39 // Gibt den gesamten AVL Baum in "in-order" Reihenfolge aus.
40 void AVL_in_order_walk(AVLTree* avlt)
41 {
42     if (avlt != NULL && avlt->root != NULL) {
43         AVL_in_order_walk_node(avlt->root);
44     }
45 }
46
47 // Falls der Baum korrekt balanciert ist, gibt die Funktion
48 // die (nichtnegative) Baumhöhe zurück
49 // Falls der Baum nicht balanciert ist, gibt die Funktion -1 zurück
50 // In dieser Funktion darf das Strukturelement AVLNode.height nicht
51 // verwendet werden
52 int AVL_is_balanced_node(AVLNode* node)
53 {
54    if (NULL == node) {return 0;}   //nichtexistente root node -> baumhoehe 0
55
56    int leftheight = AVL_is_balanced_node(node->left);
57    if (-1 == leftheight) {return -1;}  //checke linken unterbaum
58
59    int rightheight = AVL_is_balanced_node(node->right);
60    if (-1 == rightheight) {return -1;}  //checke rechten unterbaum
61
62     int balance = leftheight - rightheight;
63     if (balance < -1 || balance > 1) {return -1;}
64     //Berechen die eigene Höhe:
65     int myheight = 1 + ((leftheight > rightheight)? leftheight : rightheight)
       ↪ ;
66     return myheight;
67 }
68
```



```
69 // Gibt 1 aus, falls der Baum korrekt balanciert ist, ansonsten 0
70 void AVL_test_balance(AVLTree* avlt)
71 {
72  if (AVL_is_balanced_node(avlt->root) >= 0) {printf("1\n");}
73  else                                       {printf("0\n");}
74 }
75
76 // Diese Funktion führt eine Linksrotation auf dem angegebenen Knoten aus,
77 // ändert dessen Baumhöhe und die des linken Kindes
78 void AVL_rotate_left(AVLTree* avlt, AVLNode* node)
79 {
80    // Hier bitte Ihr Code.
81 }
82
83 // Diese Funktion führt eine Rechtsrotation auf dem angegebenen Knoten aus,
84 // ändert dessen Baumhöhe und die des rechten Kindes
85 void AVL_rotate_right(AVLTree* avlt, AVLNode* node)
86 {
87    // Hier bitte Ihr Code.
88 }
89
90 // Diese Funktion rebalanciert den AVL Baum, nachdem
91 // der Knoten node eingefügt wurde.
92 void AVL_rebalance(AVLTree* avlt, AVLNode* node)
93 {
94    // Hier bitte Ihr Code.
95 }
96
97 // Für den Suchbaum soll nach dem Einfügen die Eigenschaft gelten,
98 // dass alle linken Kinder einen Wert kleiner gleich (<=) und alle
99 // rechten Kinder einen Wert grö ß er (>) haben.
100 //
101 // Zusätzlich soll sichergestellt sein, dass der Baum nach Aufruf
102 // korrekt balanciert ist
103 void AVL_insert_value(AVLTree* avlt, int value)
104 {
105    // Hier bitte Ihr Code.
106 }
107
108 // Löscht den Teilbaum unterhalb des Knotens node rekursiv durch
109 // "post-order" Traversierung, d.h. zurerst wird der linke und dann
110 // der rechte Teilbaum gelöscht. Anschlie ß end wird der übergebene Knoten
111 // gelöscht.
112 void AVL_delete_subTree(AVLNode* node)
113 {
114    // Hier bitte Ihr Code.
115 }
116
117 // Löscht den gesamten Baum AVL und gibt den Speicher aller Knoten frei.
118 void AVL_delete_tree(AVLTree* avlt)
119 {
120    if(avlt != NULL && avlt->root != NULL)
121    {
122       AVL_delete_subTree(avlt->root);
```

```
123         avlt->root = NULL;
124         avlt->numberOfElements = 0;
125      }
126   }
127
128   int main(int argc, char** args)
129   {
130      if (argc != 2)
131      {
132         printf("Nutzung:␣%s␣<Dateiname>\n",args[0]);
133         return 1;
134      }
135      // Hier bitte Ihr Code.
136      return 0;
137   }
```