Selbstausbalancierender Binärbaum

Matrikelnummer: 9026481

1. Kurze Zusammenfassung der Aufgabe

Ein binärer Baum ist eine Baum-Daten Struktur aus der Informatik. Er besteht aus einer Struktur die neben einem, oder mehreren Nutzwerten aus einem Verweis auf 2 Elemente desselben Typs, mit kleineren Werten links und größeren (gleichen) Werten rechts besteht.

Ein großer Vorteil des Binärbaum besteht darin, dass er im Vergleich zu verketten Listen wesentlich besser dafür geeignet ist größere Datenmengen sortiert zu speichern und gezielt Werte in diesem Baum zu finden.

Damit diese Schnelligkeit gewährleistet ist muss jedoch sichergestellt sein, dass der Baum ausbalanciert und nicht entartet ist. Dieses Problem wird anhand eines kleinen Beispiels veranschaulicht:

Im Extremfall würde eine Eingabe von Zahlen in aufsteigender- oder absteigender Reihenfolge den Binärbaum in eine einfach verkettete Liste überführen.

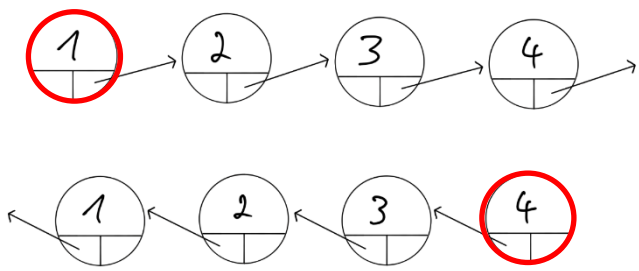

Abbildung 1

(Anmerkung: Die Kreise stehen für ein Strukturelement des Baumes. Die Zahl ist der Nutzwert, die zwei kleinen Flächen darunter stehen für die Zeiger auf Elemente links und rechts vom jetzigen Element. Ist kein Zeiger im Feld eingehängt ist dieser ein NULL Zeiger. Der rote Kreis markiert die Wurzel des Baumes).

Der obere Baum würde aus der folgenden Eingabe resultieren: 1, 2, 3, 4. Der untere entsteht aus denselben Zahlen in umgekehrter, absteigender Reihenfolge.

Das Problem hierbei ist, dass die Vorteile eines Binärbaums in diesem Fall verschwinden. Wenn der Anwender das Element 4 sucht, würde er im oberen Fall einmal den gesamten Baum durchsuchen müssen, bei 4 Eingabewerten mag das noch unproblematisch sein, aber umso größer die Anzahl der Elemente umso länger würde im Extremfall eine Suche dauern.

Im Vergleich dazu die Bäume mit denselben Zahlen in ausbalancierter Form:

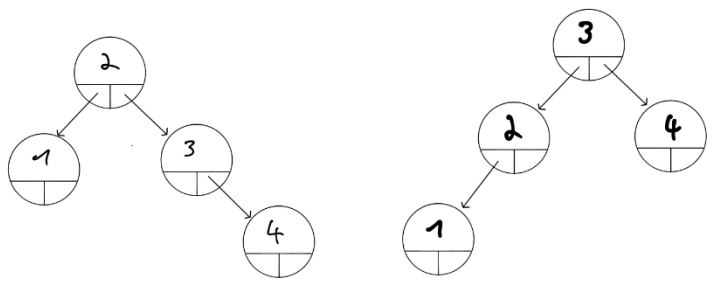

Abbildung 2

Bereits an diesem kleinen Beispiel erkennt man die Vorteile eines ausbalancierten Baumes:
Um zu den am weitest entfernten Elementen zu kommen benötigt man jetzt nur noch maximal 2 Schritte, während es vorher 3 waren.

Im Folgenden wird ein Algorithmus beschrieben, welcher einen beliebig großen, komplexen, entarteten Baum ausbalanciert. Zusätzlich wird noch eine Funktion beschrieben, welche den Baum ausgibt und dazu den Pfad zu jedem Element mit ausgibt.

2. Übersicht

Um das Programm möglichst simpel und gut wartbar zu entwickeln wurde für alle relevanten Aspekte eine eigene Funktion geschrieben. So besteht das finale Programm aus 8 Unterfunktionen.

1. void eingabe(baum **wurzel); //Vorlage
2. void ausgabe(baum *zgr); //Vorlage
3. void ausgabe_mit_pfad(baum *zgr, char *path);
4. baum* spin_right(baum* y, baum* x);
5. baum* spin_left(baum* y, baum* x);
6. baum* balance(baum *wurzel);
7. int hoehe(baum *lb);
8. int max(int a, int b);

Die Struktur Baum wurde übernommen, lediglich mit *einem typecast struct Baum baum* vereinfacht. Die 2 vorgegebenen Funktionen *einlesen()* und *ausgeben()* wurden komplett übernommen.

Die Funktion *ausgabe_mit_pfad()* wird genutzt um den gesamten Binärbaum rekursiv auszugeben, wobei zu jedem ausgegebenen Element auch der Pfad zu ihm hin ausgegeben wird.

Die restlichen Funktionen werden für das Balancieren des Baumes benötigt. Aufgerufen werden sie alle von der rekursiven Funktion *balance()* welche die Logik zum Balancieren des gesamten Baumes enthält.

3. Ansatz zur Umsetzung

3.1 Grundlagen

Beim Einsortieren eines Wertes in den Baum wird immer zwangsweise garantiert, dass links vom derzeitigen Element nur Elemente mit kleinerem Wert kommen, während rechts die Werte größer/ gleich sind. Diese Eigenschaft kann man ausnutzen, um den Baum auszubalancieren.

Dies geschieht durch Drehung zweier Knoten. Wie genau eine solche Drehung funktioniert soll an einem Beispiel veranschaulicht werden. Grundsätzlich wird zwischen 2 Grunddrehungen unterschieden. Die Links- und die Rechtsdrehung. Die Linksdrehung wird angewandt, wenn der Baum rechts höher ist und die Linksdrehung, wenn der Baum rechts höher ist.

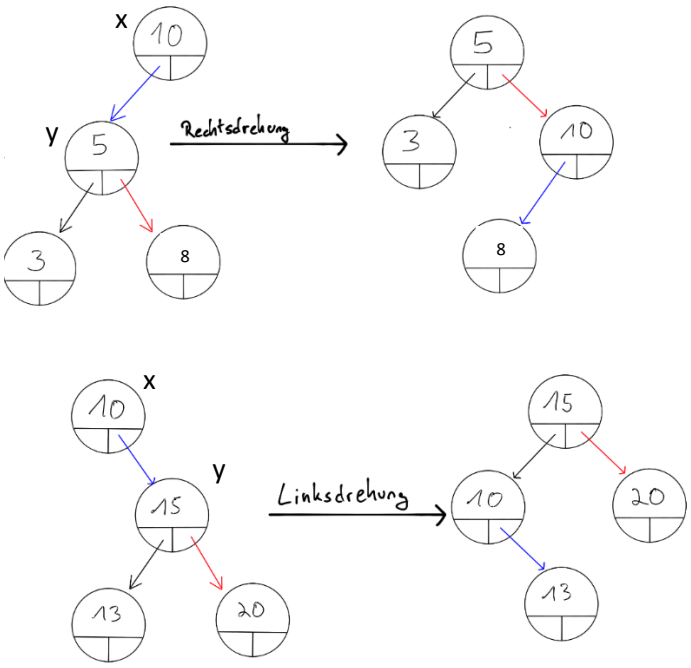

Abbildung 3

Oben im Bild eine Rechtsdrehung, darunter eine Linksdrehung.

Bei einer Drehung werden 2 Elemente (x und y) im Baum „getauscht“, dadurch wird die Balance des Baumes verändert. Im ersten Beispiel (Rechtsdrehung) sieht man einen Baum bestehend aus 4 Elementen: 10, 5, 3, 8. Die 10 ist die Wurzel, alle anderen Elemente befinden sich von ihr auf der linken Seite. Der Baum ist also entartet.

Dies wird im Programm durch Bestimmung der **Höhe** gemessen. Die Höhe ist die Anzahl an Elementen, die ausgehend vom jetzigen Element den längsten Weg nach unten bilden.

Das Element rechts von der Wurzel ist ein NULL Zeiger, hat also die Höhe 0. Das Element links von der Wurzel, die 5, hat eine Höhe von 2 (5(1) → 3/8(2)). Daraus ergibt sich eine Höhendifferenz von 2 zwischen den beiden Seiten der Wurzel. Jetzt wird eine Rechtsdrehung der Elemente 5(y) und 10(x) durchgeführt.

Die neue Wurzel ist die 5, ihr rechter Zeiger verweist nun auf die Ursprüngliche Wurzel (diese muss per Definition größer sein als die 5, sonst wäre zu Beginn die 5 nicht links von der Wurzel eingeordnet

worden). Der Links-Zeiger von Element 10 zeigt jetzt auf die 8, welche vorher rechts von der 5 war. Der Rechtszeiger der 10 ändert sich nicht. (In den Drehungsfunktionen werden immer 2 Baumelement übergeben. X und Y. Y ist das Element, welches nach einer Drehung die neue Wurzel ist und vorher „unten“ war).

Das Resultat ist ein ausbalancierter Baum. Die Höhendifferenz beträgt nun 1 (Rechts 2 – Links 1). Bedingt durch die ungerade Anzahl an Elementen unter der Wurzel ist eine Differenz von 0 nicht möglich.

Analog zur Rechtsdrehung existiert auch die Linksdrehung, wobei das Element Rechts von der Wurzel als neue Wurzel gewählt wird.

3.2 Sonderfälle

Für die meisten Baumstrukturen reichen einfache Links- oder Rechtsdrehungen, um sie auszubalancieren. Es gibt allerdings Situationen, in denen eine Abwandlung der Funktionen notwendig ist, um nicht in eine Endlosschleife von Drehungen zu kommen. Dies wird an einem einfachen Beispiel veranschaulicht:

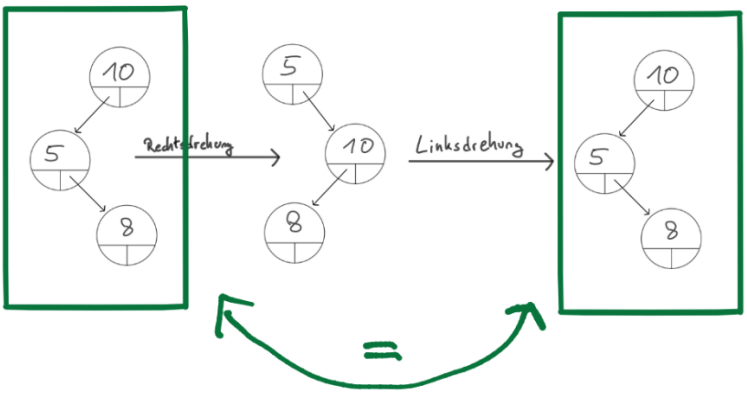

Abbildung 4

Gestartet wird mit dem Baum: 10-5-8. Dieser ist links höher als rechts → Rechtsdrehung. Nach der Drehung liegt der Baum: 5-10-8 vor. Dieser ist rechts höher als links → Linksdrehung. Jetzt ist der Baum identisch dem Ersten Baum, es entsteht eine Dauerschleife aus Rechts- und Linksdrehungen.

Dies kann verhindert werden, wenn vor der ersten Rechtsdrehung eine Linksdrehung der Elemente 5 und 8 durchgeführt wird. Erst danach wird die Linksdrehung durchgeführt. Dieses Vorgehen wird am selben Beispiel demonstriert.

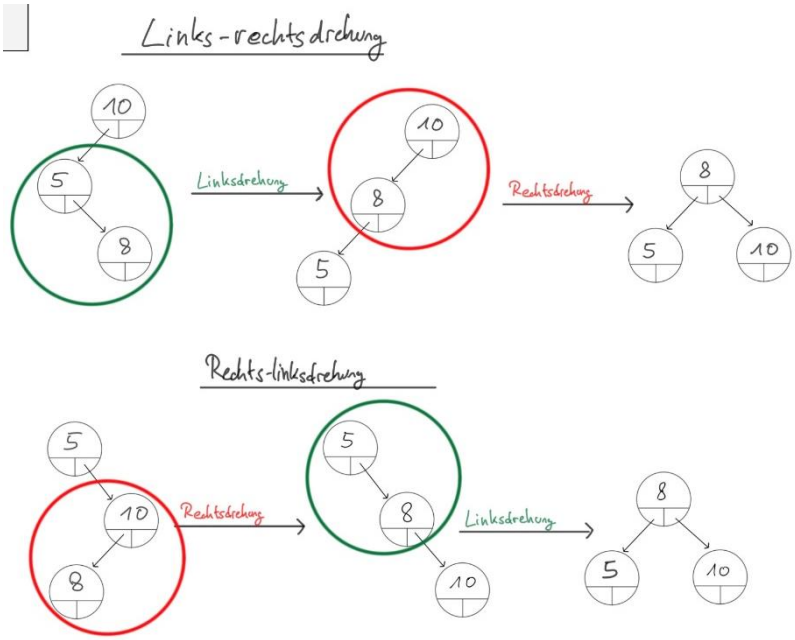

Abbildung 5, oben: Links-rechtsdrehung, unten Rechts-linksdrehung

Dadurch entstehen 2 neue „Drehtechniken“: Die Links-rechtsdrehung, und die Rechts-linksdrehung.

Da die beiden Funktionen aus den Grunddrehungen bestehen wurde für diese keine neue Funktion angelegt, stattdessen wird im *balance()* Algorithmus einfach erst die eine, dann die andere Drehung aufgerufen.

Die letzte Schwierigkeit ist nun noch zu erkennen, wann eine einfache Grunddrehung genügt bzw. wann eine der Sonderformen benötigt wird.

Die Doppeldrehungen müssen dann angewandt werden, wenn die „Innenseite“ (Diejenige Seite, welche näher zur Wurzel ist) des Wurzelnachfolgers höher ist als die Außenseite. Im Beispiel der Links-rechtsdrehung in Abb. 5 wird also zuerst erkannt, dass der Baum links höher ist als rechts. Er muss rechts gedreht werden. Vor der Rechtsdrehung wird noch kontrolliert wie die Höhe des Nachfolgers (die 5) ist. Da die „Innenseite“ mit dem Element 8 eine Höhe von 1 hat, während die „Außenseite“ eine Höhe von 0 hat wird also vor der Rechtsdrehung eine Linksdrehung durchgeführt.

Analog dazu die Rechts-linksdrehung.

3.3 Funktionsweise der *balance()*-Funktion

In den 2 Punkten zuvor wurden die verschiedenen Drehungen dargestellt. Jetzt wird die *balance()*-Funktion erklärt, welche die beschriebenen Techniken anwendet.

Während bei kleinen Bäumen wie oben im Beispiel (Abbildung 1) schon eine Drehung reicht, um den Baum zu ordnen sind bei komplexeren Bäumen mehrere solcher Drehungen an verschiedenen Elementen notwendig. In welche Richtung gedreht wird hängt von der Differenz der Höhen der Nachfolgenden Elemente ab.

Es wird immer Höhe links – Höhe rechts gerechnet. Ist die Differenz kleiner als -1 ist die rechte Seite höher und es muss links gedreht werden. Bei Differenz > +1 ist die linke Seite höher und es muss rechts gedreht werden.

Im Prinzip wird der *balance()* Funktion die Wurzel eines Baumes überreicht, dann wird solange an der Wurzel gedreht bis ein Gleichgewicht (Höhendifferenz <= 1) von linker und rechter Seite vorliegt. Danach wird mittels Rekursion die Funktion nochmal aufgerufen, erst mit dem linken Teilbaum der jetzigen Wurzel und dann einmal mit dem rechten Teilbaum der jetzigen Wurzel. So wird der Baum von oben nach unten, erst links dann rechts, durchbalanciert. Die Rekursion läuft solange, wie die nachfolgenden Teilbäume nur noch NULL Zeiger sind. Es wird also jedes Element einmal als Wurzel der Funktion aufgerufen.

3.4 Probleme/ Fehler

Probleme in der Lösung des Algorithmus treten dann auf, wenn im Baum hintereinander Elemente mit dem gleichen Nutzwert auftreten. Das liegt daran, dass die Drehungen nur deswegen funktionieren kann, weil man davon ausgeht, dass Elemente rechts vom aktuellen größer sind und Elemente links vom aktuellem kleiner sind. Bei gleichen Werten ist dies nicht der Fall und es kann passieren, dass nach der Drehung die Regeln nicht mehr eingehalten werden, weil dann zwei gleiche Elemente links von einander sind. Ein Beispiel ist z.B. der folgende Baum: 5-5-5.

```
Eingabe: 5
Eingabe: 5
Eingabe: 5
Eingabe: 0
vor balancing:
Ausgabe Baum: 5   5   5
5 (w)   5 (R)   5 (RR)
nach balancing:
Ausgabe Baum: 5   5   5
5 (L)   5 (w)   5 (R)
```

Abbildung 6

Die Höhe ist zwar jetzt passend, aber links von der Wurzel 5 ist eine 5, was nicht erlaubt ist. Da dieser Fall laut Aufgabenstellung nicht relevant ist wird hier nicht weiter darauf eingegangen, es sollte aber erwähnt werden.

4. ausgabe_mit_pfad()

Neben dem Balancieren des Baumes sollte auch eine Funktion entwickelt werden, welche die Elemente des Baumes in aufsteigender Reihenfolge ausgibt und dabei noch den Weg zum jeweiligen Element von der Wurzel aus beschreibt. Die Ausgabe könnte z.B. so aussehen:

```
1 (LL)   2 (L)   3 (w)   4 (RL)   5 (R)   6 (RR)
```

Abbildung 7

Die Wurzel wird mit einem „w“ markiert, für den Rest gilt: L-Links, R-Rechts.

4.1 Funktionsweise

Der Aufbau der Funktion ausgabe_mit_pfad() ähnelt stark dem Aufbau der vorgegebenen rekursiven Ausgabe Funktion. Sie wird allerdings noch um einen Übergabeparameter, einen Zeiger auf ein Characterfeld erweitert. Dieser String wird genutzt, um den aktuellen Pfad zu speichern.
void ausgabe_mit_pfad(baum *zgr, char *path)

Die Funktion startet mit einem String der Länge 1 Char aus Main, indem nur das String-Abschlusszeichen steht.

Wenn links vom aktuellen Element ein Element ist, wird ein neuer String angelegt mit der Funktion *malloc()*, dessen Länge gleich dem aktuellen String + 1 ist, da jetzt ein zusätzlicher Wegbeschreibender Character gespeichert werden muss. Mit *memcpy()* wird in den neuen String der Inhalt des aktuellen Strings kopiert und mit der Funktion *strcat()* aus *string.h* wird an die Kopie der neue Char (L) angehängt und automatisch das String-Abschlusszeichen eins nach hinten geschoben.

Dies geschieht solange bis das kleinste Element gefunden wurde. Dieses wird dann ausgegeben *(%d für den Wert, %s für den pfad*). Es wird geschaut ob rechts noch Elemente sind und dem rekursiven Aufruf wird ein neuer String übergeben (Erstellen eines neuen Strings, kopieren des aktuellen Pfads, anhängen von „R“). So geht die Funktion den gesamten Baum durch.

Um die Wurzel zu erkennen wird vor jeder Ausgabe kontrolliert, ob das erste Element das String-Abschlusszeichen ist. In dem Fall wird lediglich (w) ausgegeben.

Am Ende der Funktion wird immer der Speicher für den aktuellen String freigegeben, um die Speicherbelegung gering zu halten.

4.2 Probleme

Die Funktion arbeitet einwandfrei, es gibt lediglich ein kleines Problem welches dadurch entsteht, dass der vom Betriebssystem durch *malloc()* reservierte Speicher am Ende immer freigegeben wird. Das hat zur Folge, dass auch der String gelöscht, welcher der Funktion von *main()* aus übergeben worden ist.

Wenn die Funktion also mehrmals aus *main()* aufgerufen werden soll muss für jeden Aufruf der String neu initialisiert werden. Die Logik dazu in die Funktion selbst zu schreiben wäre sehr unvorteilhaft, da die Funktion aufgrund der Rekursion so oft aufgerufen wird wie der Baum Elemente hat.

Da in der Praxis die Funktion aber ohnehin nur gebraucht wird, um zu kontrollieren ob der Baum richtig geordnet ist, sollte dies kein relevantes Problem sein.

5. Fazit und Ausblick

Die beiden Funktionen *balance()* und *ausgabe_mit_pfad()* funktionieren einwandfrei.

In der Ausgabe Funktion kann durch den Einsatz von Rekursion und variabler Speicherplatz Allokation über *malloc()* ein sehr kompakter und unkomplizierten Weg genutzt werden um den Pfad zu jedem Element auszugeben, dank der Funktion *strcat()* ist das Erweitern des Strings in einer Zeile Code erledigt.

Auch die Funktion *ballance()* kann dank Rekursion sehr elegant einen ungeordneten Baum Stück für Stück balancieren.

Dadurch, dass lediglich Zeiger neu verkettet werden und nicht die Inhalte der Strukturen neu einsortiert werden, ist das Programm sehr performant und auch gut skalierbar um in Zukunft die Struktur um wesentlich mehr Nutzdaten zu erhöhen und so z.B. in einer Datenbank eingesetzt zu werden.

Des Weiteren werden nur dann Änderungen in Form von Drehungen vorgenommen, wenn diese auch nötig sind. Sollte bei der Eingabe z.B. schon ein perfekt ausbalancierter Baum eingegeben werden geht der Lösungsalgorithmus lediglich einmal jedes Element durch, ohne unnötiger Weise den gesamten Baum neu zu verketten, das erhöht die ohnehin schon gute Performance nochmal ein Stück. Die Hilfsfunktion der Höhenbestimmung läuft schon mit 2 Zeilen Code einwandfrei dank Rekursion. Auch die Drehungen sind so simpel, dass hier jeweils 3 Zeilen Code genügen, was die Fehleranfälligkeit im Vergleich zu langen/ komplexen Algorithmen vermindert.

6. Struktogramme

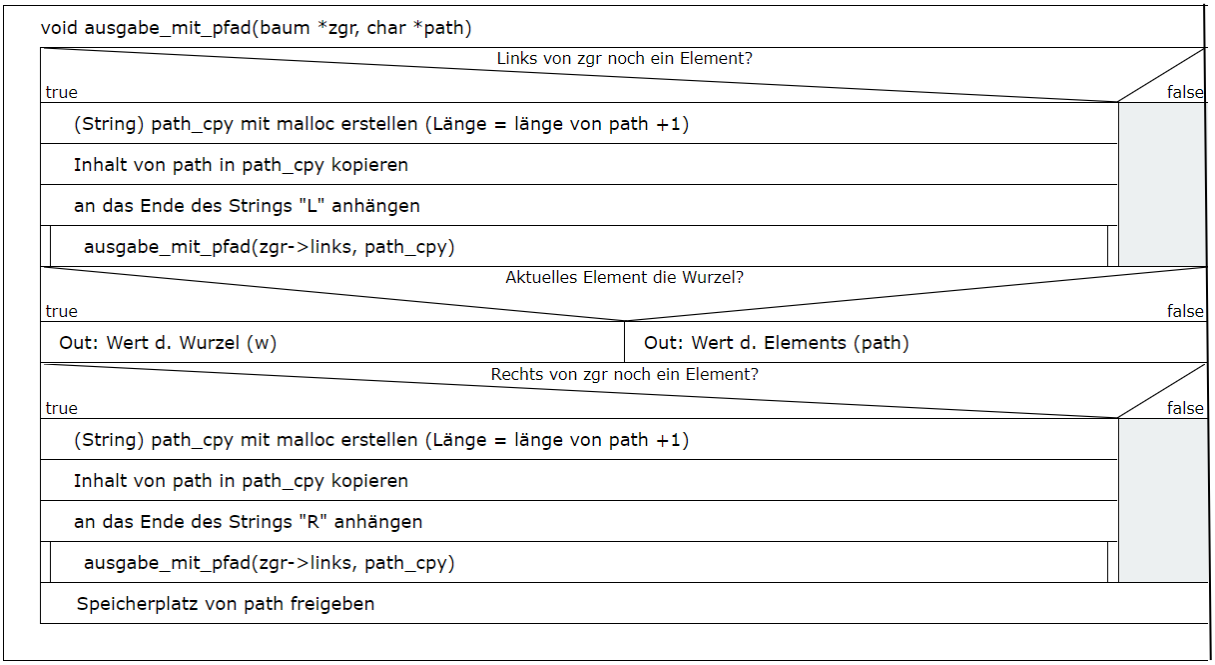

Abbildung 8

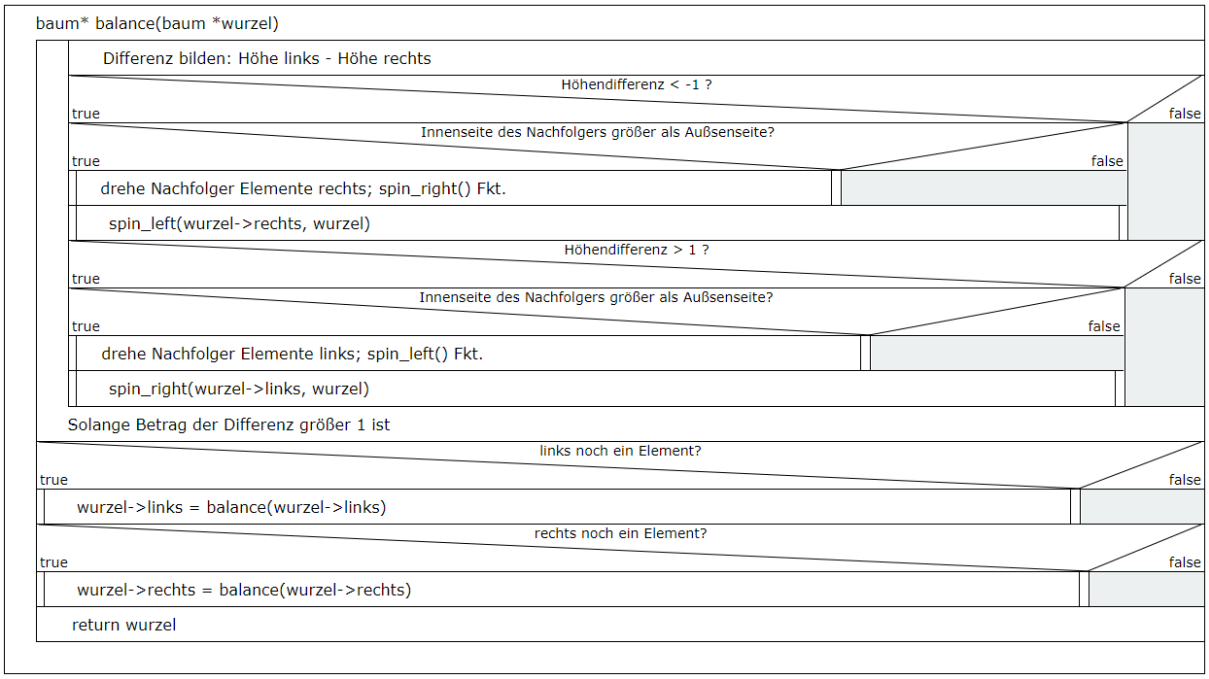

Abbildung 9

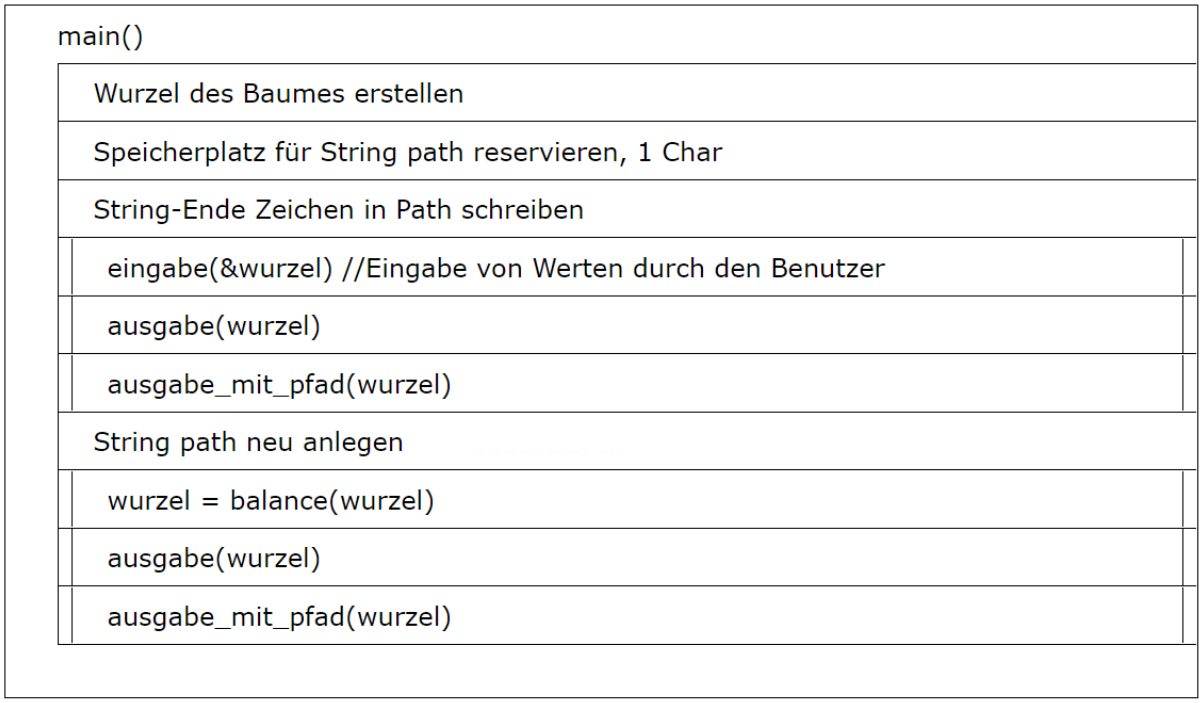

Abbildung 10

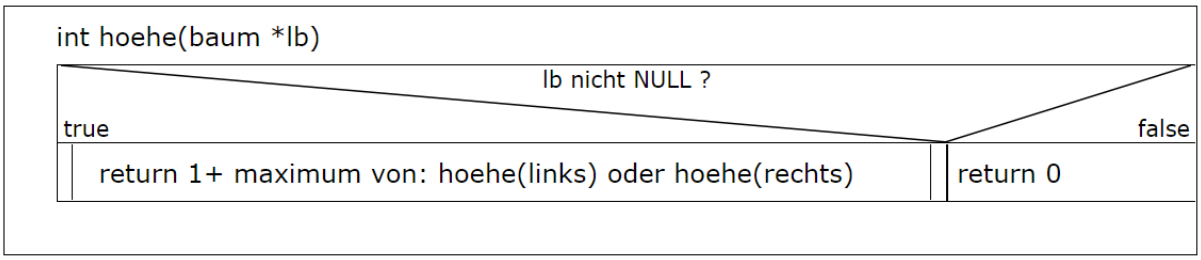

Abbildung 11

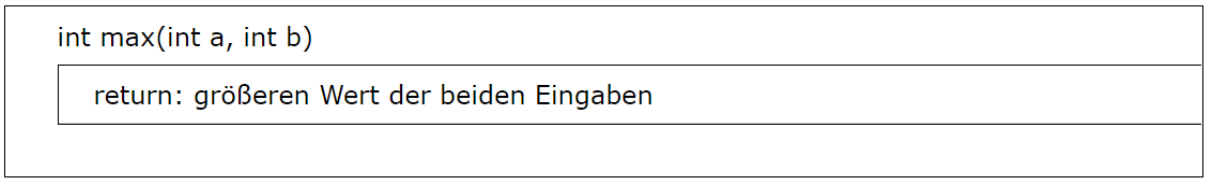

Abbildung 12

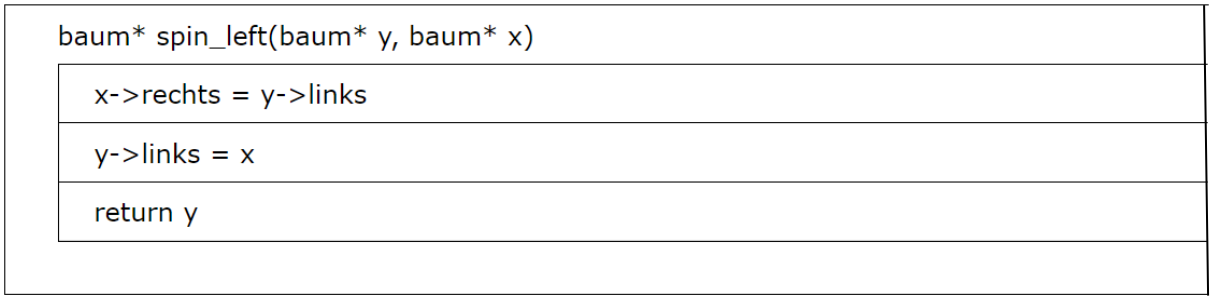

Abbildung 13

```
baum* spin_right(baum* y, baum* x)
    x->links = y->rechts
    y->rechts = x
    return y
```

Abbildung 14

7. Programmtests

```
Eingabe: 1
Eingabe: 2
Eingabe: 3
Eingabe: 4
Eingabe: 5
Eingabe: 6
Eingabe: 7
Eingabe: 8
Eingabe: 9
Eingabe: 0
vor balancing:
Ausgabe Baum: 1   2   3   4   5   6   7   8   9
1 (w)   2 (R)   3 (RR)   4 (RRR)   5 (RRRR)   6 (RRRRR)   7 (RRRRRR)   8 (RRRRRRR)   9 (RRRRRRRR)
nach balancing:
Ausgabe Baum: 1   2   3   4   5   6   7   8   9
1 (LLL)   2 (LL)   3 (L)   4 (LR)   5 (w)   6 (RL)   7 (R)   8 (RR)   9 (RRR)

------------------
(program exited with code: 0)

Drücken Sie eine beliebige Taste . . .
```

Abbildung 15, aufsteigende Eingabe (einfach verzeigerte Liste)

```
Eingabe: 9
Eingabe: 8
Eingabe: 7
Eingabe: 6
Eingabe: 5
Eingabe: 4
Eingabe: 3
Eingabe: 2
Eingabe: 1
Eingabe: 0
vor balancing:
Ausgabe Baum: 1   2   3   4   5   6   7   8   9
1 (LLLLLLLL)   2 (LLLLLLL)   3 (LLLLLL)   4 (LLLLL)   5 (LLLL)   6 (LLL)   7 (LL)   8 (L)   9 (w)
nach balancing:
Ausgabe Baum: 1   2   3   4   5   6   7   8   9
1 (LLL)   2 (LL)   3 (L)   4 (LR)   5 (w)   6 (RL)   7 (R)   8 (RR)   9 (RRR)

------------------
(program exited with code: 0)

Drücken Sie eine beliebige Taste . . .
```

Abbildung 16, absteigende Eingabe (einfach verzeigerte Liste)

```
Eingabe: 10
Eingabe: 30
Eingabe: 50
Eingabe: 70
Eingabe: 90
Eingabe: 80
Eingabe: 60
Eingabe: 40
Eingabe: 20
Eingabe: 0
vor balancing:
Ausgabe Baum: 10   20   30   40   50   60   70   80   90
10 (w)   20 (RL)   30 (R)   40 (RRL)   50 (RR)   60 (RRRL)   70 (RRR)   80 (RRRRL)   90 (RRRR)
nach balancing:
Ausgabe Baum: 10   20   30   40   50   60   70   80   90
10 (LL)   20 (LLR)   30 (L)   40 (LR)   50 (w)   60 (RL)   70 (R)   80 (RRL)   90 (RR)

-----------------
(program exited with code: 0)

Drücken Sie eine beliebige Taste . . .
```

Abbildung 17, gemischte Reihenfolge

```
Eingabe: 10
Eingabe: 5
Eingabe: 15
Eingabe: 20
Eingabe: 14
Eingabe: 9
Eingabe: 1
Eingabe: 0
vor balancing:
Ausgabe Baum: 1   5   9   10   14   15   20
1 (LL)   5 (L)   9 (LR)   10 (w)   14 (RL)   15 (R)   20 (RR)
nach balancing:
Ausgabe Baum: 1   5   9   10   14   15   20
1 (LL)   5 (L)   9 (LR)   10 (w)   14 (RL)   15 (R)   20 (RR)

------------------
(program exited with code: 0)

Drücken Sie eine beliebige Taste . . .
```

Abbildung 18, Eingabe eines ausbalancierten Baumes)

```c
/*****************************************************************/
/* Hausarbeit Elektrotechnik, Labor Informatik II Hr. Kandziora */
/* Matrikelnummer: 9026481                                       */
/* Ausbalancierender Binärer Baum                                */
/*****************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h> //für Funktion: strcat(), in ausgabe_mit_pfad()

struct Baum //Struktur für eine Element im Binären Baum
{
    int wert;
    struct Baum *links, *rechts;
};
typedef struct Baum baum; //einfacherer Aufruf: struct Baum vs. baum

void eingabe(baum **wurzel); //Vorlage
void ausgabe(baum *zgr); //Vorlage
/*Ausgabe des baumes mit Pfad zu jedem Element*/
void ausgabe_mit_pfad(baum *zgr, char *path);
/*Funktion zum balacieren des Baumes. Rechtsdrehung zweier Knoten*/
baum* spin_right(baum* y, baum* x);
/*Funktion zum balacieren des Baumes. Linksdrehung zweier Knoten */
baum* spin_left(baum* y, baum* x);
/*Funktion welche das ausbalcieren vornimmt */
baum* balance(baum *wurzel);
/* Funktion um die Höhe eines Elementes zu bestimmen */
int hoehe(baum *lb);
int max(int a, int b); //Rückgabe des größeren Wertes

int main()
{
    struct Baum *wurzel = NULL; //Baum erstellen
    char *path; //String anlegen für Fkt. ausgabe_mit_pfad()
    path = (char*)malloc(sizeof(char));
    path[0] = '\0';
    eingabe(&wurzel); //Eingabe von Werten des Benutzers in den Baum.
    printf("vor balancing:\nAusgabe Baum: ");
    ausgabe(wurzel); //Ausgabe der Elemente des Baumes
    printf("\n");
    ausgabe_mit_pfad(wurzel, path); //Ausgabe der Elemente mit Pfad
    wurzel = balance(wurzel); //Ausbalancieren des Baumes
    printf("\nnach balancing:\nAusgabe Baum: ");
    path = (char*)malloc(sizeof(char)); //reinitialisierung vom path
//string da dieser bei der letzten ausgabe mit pfad gelöscht worden ist
    path[0] = '\0';
    ausgabe(wurzel);//Ausgabe der Elemente des ausbalancierten Baumes
    printf("\n");
    ausgabe_mit_pfad(wurzel, path); //Ausgabe der Elemente mit Pfad
    return 0;
}

void eingabe(baum **wurzel) //Eingabefunktion original Vorlage
{
    int wert;
    baum *zgr, *lz;

    do
    {
        printf("Eingabe: ");
        scanf("%d", &wert);
        if (wert != 0)
        {
            zgr = (baum*)malloc(sizeof(baum));
```

```
65              zgr->wert = wert;
66              zgr->links = zgr->rechts = NULL;
67              if (*wurzel == NULL) *wurzel = zgr;
68              else
69              {
70                  lz = *wurzel;
71                  while (lz != NULL)
72                  {
73                      if(wert < lz->wert)
74                      {
75                          if(lz->links == NULL)
76                          {
77                              lz->links = zgr;
78                              lz = NULL;
79                          }
80                          else lz = lz ->links;
81                      }
82                      else
83                      {
84                          if(lz->rechts == NULL)
85                          {
86                              lz->rechts = zgr;
87                              lz = NULL;
88                          }
89                          else lz = lz-> rechts;
90                      }
91                  }
92              }
93          }
94      } while (wert != 0);
95  }
96
97  void ausgabe_mit_pfad(baum *zgr, char *path){
98      if(zgr->links != NULL){
99  /*Der Pfad zum Element wird in einem String gespeichert. Dafür wird für
100  * jeden rekursiven Funktionsaufruf ein neuer String erstellt mit +1
101  * Platz für ein neuen Wegbeschreibenden Character (L/R) */
102         char *path_cpy = (char*)malloc(sizeof(char)*(strlen(path)+1));
103         strcpy(path_cpy, path);
104 /*Anhängen von "L" an die Kopie des strings */
105         strcat(path_cpy, "L");
106 /* Rekursiver Aufruf der Funktion mit aktualisiertem, kopiertem Pfad*/
107         ausgabe_mit_pfad(zgr->links, path_cpy);
108     }
109     if(path[0] == '\0'){ //Wurzel finden und mit (w) markieren.
110         printf("%d (w)   ", zgr->wert);
111     }
112     else{ //Ausgabe von Wert und Pfad des Elements
113         printf("%d (%s)   ", zgr->wert, path);
114     }
115     if(zgr->rechts != NULL){//Analog zum obigen vorgehen, nun für Rechts
116         char *path_cpy = (char*)malloc(sizeof(char)*(strlen(path)+1));
117         strcpy(path_cpy, path);
118         strcat(path_cpy, "R");
119         ausgabe_mit_pfad(zgr->rechts, path_cpy);
120     }
121     free(path); //Strings am Ende wieder freigeben
122 }
123
124 baum* spin_right(baum* y, baum* x){ //Rechtsdrehung von y und x
125 /* Vertauschen der Zeiger um Rangfolge im Baum zu ändern */
126     x->links = y->rechts;
127     y->rechts = x; //Zuvor war x über y. Jetzt ist y über x.
128     return y;
```

```
129 }
130
131 baum* spin_left(baum* y, baum* x){ //Linksdrehung von y und x
132     x->rechts = y->links; //Funktionsweise analog zur Rechtsdrehung
133     y->links = x;
134     return y;
135 }
136
137 baum* balance(baum *wurzel){
138     int hoehe_links;
139     int hoehe_rechts;
140     int aktuelle_diff;//Speicher der Differenz von Höhe Links und Rechts
141     do{
142         hoehe_links = hoehe(wurzel->links);
143         hoehe_rechts = hoehe(wurzel->rechts);
144         aktuelle_diff = hoehe_links - hoehe_rechts;
145 /*Wenn aktuelle diff kleiner als -1 ist, ist der Baum rechts höher */
146         if(aktuelle_diff < -1){
147 /*Ein Baum der rechts höher ist, wird durch linksdrehung balanciert
148  *ggf. ist vorher jedoch eine Linksdrehung notwendig (Links-Rechts)*/
149             if(hoehe(wurzel->rechts->links) >
150             hoehe(wurzel->rechts->rechts)){
151                 wurzel->rechts =
152                 spin_right(wurzel->rechts->links, wurzel->rechts);
153             }
154             wurzel = spin_left(wurzel->rechts, wurzel);
155         }
156 /* Analoges Vorgehen wie zuvor, jetzt für den Fall das Links höher ist*/
157         else if(aktuelle_diff > 1){
158             if(hoehe(wurzel->links->rechts) >
159             hoehe(wurzel->links->links)){
160                 wurzel->links =
161                 spin_left(wurzel->links->rechts, wurzel->links);
162             }
163             wurzel = spin_right(wurzel->links, wurzel);
164         }
165 /*Solange wiederholen wie die aktuelle Wurzel nicht balaciert ist*/
166     }while(aktuelle_diff > 1 || aktuelle_diff < -1);
167 /*Wenn die Wurzel ausbalaciert ist werden rekursiv die Teilbäume links-
168  * und rechts ausbalciert. Die neue Wurzel wird dabei gespeichert */
169     if(wurzel->links != NULL) wurzel->links = balance(wurzel->links);
170     if(wurzel->rechts != NULL) wurzel->rechts = balance(wurzel->rechts);
171     return wurzel;
172 }
173
174 int hoehe(baum *lb){
175 /* Höhe bestimmen durch suchen des längsten Wegs zum "Boden", rekursiv*/
176     if (lb != NULL) return 1 + max(hoehe(lb->links), hoehe(lb->rechts));
177     else return 0;
178 }
179
180 int max(int a, int b){ //Rückgabe des größeren Elementes
181     return a > b ? a : b;
182 }
183
184 void ausgabe(baum *zgr) //Ausgabefunktion Original Vorlage
185 {
186     if(zgr->links != NULL) ausgabe(zgr->links);
187     printf("%d   ", zgr->wert);
188     if(zgr->rechts !=NULL) ausgabe(zgr->rechts);
189 }
190
```