

# Hardwarenahe Programmierung / Angewandte Informatik

## Musterlösung zu den Übungsaufgaben – 23. Januar 2017

### Aufgabe 1: Ternärer Baum

Der in der Vorlesung vorgestellte *binäre Baum* ist nur ein Spezialfall; im allgemeinen können Bäume auch mehr als zwei Verzweigungen pro Knotenpunkt haben. Dies ist nützlich bei der Konstruktion *balancierter Bäume*, also solcher, die auch im *Worst Case* nicht zu einer linearen Liste entarten, sondern stets eine – möglichst flache – Baumstruktur behalten.

Wir betrachten einen Baum mit bis zu drei Verzweigungen pro Knotenpunkt, einen sog. *ternären Baum*. Jeder Knoten enthält dann nicht nur einen, sondern *zwei* Werte als Inhalt:

```
typedef struct node
{
    int content_left, content_right;
    struct node *left, *middle, *right;
} node;
```

Wir konstruieren nun einen Baum nach folgenden Regeln:

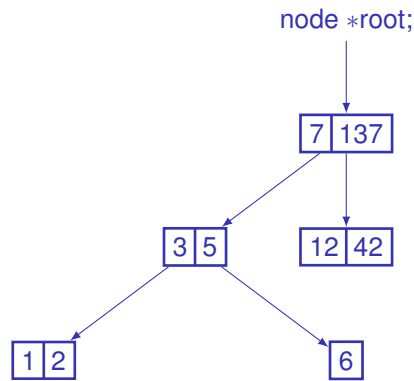
- Innerhalb eines Knotens sind die Werte sortiert: `content_left` muß stets kleiner sein als `content_right`.
- Der Zeiger `left` zeigt auf Knoten, deren enthaltene Werte durchweg kleiner sind als `content_left`.
- Der Zeiger `right` zeigt auf Knoten, deren enthaltene Werte durchweg größer sind als `content_right`.
- Der Zeiger `middle` zeigt auf Knoten, deren enthaltene Werte durchweg größer sind als `content_left`, aber kleiner als `content_right`.
- Ein Knoten muß nicht immer mit zwei Werten voll besetzt sein; er darf auch *nur einen* gültigen Wert enthalten.  
Der Einfachheit halber lassen wir in diesem Beispiel nur positive Zahlen als Werte zu. Wenn ein Knoten nur einen Wert enthält, setzen wir `content_right = -1`, und der Zeiger `middle` wird nicht verwendet.
- Wenn wir neue Werte in den Baum einfügen, werden *zuerst* die nicht voll besetzten Knoten aufgefüllt und *danach erst* neue Knoten angelegt und Zeiger gesetzt.
- Beim Auffüllen eines Knotens darf nötigenfalls `content_left` nach `content_right` verschoben werden. Ansonsten werden einmal angelegte Knoten nicht mehr verändert.

(In der Praxis dürfen Knoten gemäß speziellen Regeln nachträglich verändert werden, um Entartungen gar nicht erst entstehen zu lassen – siehe z. B. <https://de.wikipedia.org/wiki/2-3-4-Baum>.)

- (a) Zeichnen Sie ein Schaubild, das veranschaulicht, wie die Zahlen 7, 137, 3, 5, 6, 42, 1, 2 und 12 nacheinander und in dieser Reihenfolge in den oben beschriebenen Baum eingefügt werden – analog zu den Vortragsfolien ([hp-20170123.pdf](#)), Seite 21.
- (b) Dasselbe, aber in der Reihenfolge 2, 7, 42, 12, 1, 137, 5, 6, 3.
- (c) Beschreiben Sie in Worten und/oder als C-Quelltext-Fragment, wie eine Funktion aussehen müßte, um den auf diese Weise entstandenen Baum sortiert auszugeben.

### Lösung

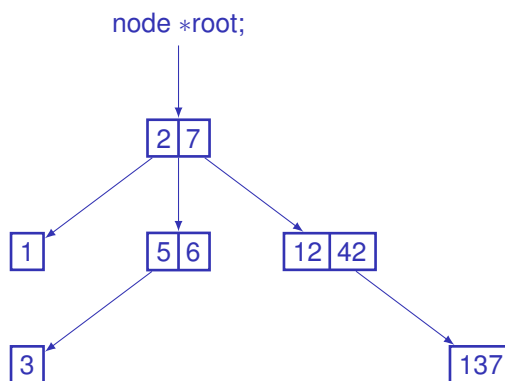
- (a) **Zeichnen Sie ein Schaubild, das veranschaulicht, wie die Zahlen 7, 137, 3, 5, 6, 42, 1, 2 und 12 nacheinander und in dieser Reihenfolge in den oben beschriebenen Baum eingefügt werden – analog zu den Vortragsfolien ([hp-20170123.pdf](#)), Seite 21.**



Bemerkungen:

- Zeiger mit dem Wert **NULL** sind nicht dargestellt: **right**-Zeiger von 7/137, **middle**-Zeiger von 3/5, sämtliche Zeiger von 1/2, 12/42 und 6.
- Beim Einfügen der 12 wird die sich bereits vorher in diesem **node** befindliche 42 zu **content\_right**, und die 12 wird das neue **content\_left**.
- Dieser Baum hat sehr einfache Regeln und ist daher *nicht* balanciert. Insbesondere unsere Regel, daß einmal angelegte Knoten nicht mehr verändert werden dürfen, steht dem im Wege. Ein einfaches Beispiel für einen *balancierten* ternären Baum ist der 2-3-Baum – siehe z. B. [https://en.wikipedia.org/wiki/2-3\\_tree](https://en.wikipedia.org/wiki/2-3_tree).

(b) **Dasselbe, aber in der Reihenfolge 2, 7, 42, 12, 1, 137, 5, 6, 3.**



Bemerkungen:

- Wieder sind Zeiger mit dem Wert **NULL** nicht dargestellt: **middle**- und **right**-Zeiger von 5/6, **left**- und **middle**-Zeiger von 12/42, sämtliche Zeiger von 1, 3 und 137.
- Beim Einfügen der 12 wird wieder die sich bereits vorher in diesem **node** befindliche 42 zu **content\_right**, und die 12 wird das neue **content\_left**.

(c) **Beschreiben Sie in Worten und/oder als C-Quelltext-Fragment, wie eine Funktion aussehen müßte, um den auf diese Weise entstandenen Baum sortiert auszugeben.**

Die entscheidende Idee ist **Rekursion**.

Eine Funktion, die den gesamten Baum ausgibt, müßte einmalig für den Zeiger **root** aufgerufen werden und folgendes tun:

1. falls der übergebene Zeiger den Wert **NULL** hat, nichts ausgeben, sondern die Funktion direkt beenden,
2. sich selbst für den **left**-Zeiger aufrufen,
3. den Wert von **content\_left** ausgeben,
4. sich selbst für den **middle**-Zeiger aufrufen,
5. sofern vorhanden (also ungleich **-1**), den Wert von **content\_right** ausgeben,
6. sich selbst für den **right**-Zeiger aufrufen.

Als C-Fragment:

```
void output_tree (node *root)
{
    if (root)
    {
        output_tree (root->left);
        printf ("%d\n", root->content_left);
        output_tree (root->middle);
        if (root->content_right >= 0)
            printf ("%d\n", root->content_right);
        output_tree (root->right);
    }
}
```

Die Datei [loesung-1c.c](#) erweitert dieses Fragment zu einem vollständigen C-Programm zum Erstellen und sortierten Ausgeben eines ternären Baums mit den Zahlenwerten von Aufgabenteil (a).

## Aufgabe 2: Aufräumen

Im Zuge der Übungsaufgaben von letzter Woche ([hp-uebung-20170116.pdf](#) – *Stack-Operations, Iterativer Floodfill, Doppelt verkettete Liste*) wurden verschiedene Funktionen zur Manipulation von Arrays und Listen erstellt.

Überarbeiten Sie diese Funktionen derart, daß das Programm auch bei unsinnigen Eingabewerten nicht abstürzt, sondern eine Fehlermeldung ausgibt und sich kontrolliert beendet oder kontrolliert weiterläuft.

Hinweis: Es ist in C leider nicht möglich, bei einem Zeiger zwischen einem sinnvollen und einem zufälligen Wert zu unterscheiden. Wir müssen uns daher bei Zeigern damit begnügen, zwischen sinnvollen Zeigern auf Daten und dem Wert **NULL** zu unterscheiden.

## Lösung

Wir nehmen die Überarbeitung an den jeweils fortgeschrittensten Versionen der Programme vor: [loesung-2-1d.c](#) ist eine Überarbeitung von [loesung-1d.c](#), und [loesung-2-3b.c](#) ist eine Überarbeitung von [loesung-3b.c](#).

Wichtige Punkte:

- **Vor jedem Zugriff auf ein Array muß sichergestellt werden, daß sich der Index im gültigen Bereich befindet.** In [loesung-2-1d.c](#) ist dies in den Funktionen [push\(\)](#), [pop\(\)](#), [insert\(\)](#) und [insert\\_sorted\(\)](#) notwendig. Die Funktionen [show\(\)](#) und [search\(\)](#) sind so geschrieben, daß dieses Problem nicht auftreten kann.
- In [loesung-2-1d.c](#) könnte man strenggenommen noch prüfen, ob die Variable [stack\\_pointer](#) möglicherweise durch Manipulation von außen einen negativen und daher sinnlosen Wert bekommen hat.  
Alternativ könnte man aus [stack\\_pointer](#) eine **unsigned**-Variable machen, die keine negativen Werte annehmen kann sondern ggf. überläuft und einen großen positiven Wert annimmt, der dann durch die Bedingung [stack\\_pointer < STACK\\_SIZE](#) abgefangen wird.
- **Vor jedem Zugriff auf einen Zeiger muß sichergestellt sein, daß der Zeiger nicht den Wert NULL hat.** In [loesung-2-3b.c](#) ist dies in den Funktionen [insert\\_into\\_list\(\)](#) und [delete\\_from\\_list\(\)](#) notwendig. Die Funktionen [check\\_list\(\)](#), [output\\_list\(\)](#) prüfen dies bereits automatisch.  
Man beachte, daß der Zeigerzeiger [node \\*\\*first](#) in [delete\\_from\\_list\(\)](#) nicht **NULL** sein darf, während dies für den Zeiger [\\*first](#), auf den er zeigt, kein Problem darstellt.
- Während des „Aufräumens“ in [loesung-2-3b.c](#) ist aufgefallen, daß die Initialisierung der Zeiger [next](#) und [prev](#) in [node \\*element5](#) fehlte. Diese Art von Fehler – Zeiger mit *zufälligen* Werten – läßt sich durch Prüfung der Zeigerwerte *nicht* abfangen.
- Die Hauptprogramme enthalten Testfälle für die Fehlerbehandlung. Weil das Programm nach jedem Fehler abbricht, kann man diese nicht gleichzeitig, sondern nur einzeln testen. Aus diesem Grunde sind alle Testfälle bis auf einen auskommentiert.