Fakultät IV - Elektrotechnik und Informatik
Einführung in die Programmierung WS 2017/18
Feldmann / Foucard / Lichtblau / Streibelt / Krüger / Enghardt / Palmer

Aufgabenblatt 6

letzte Aktualisierung: 15. Dezember, 11:06 Uhr

(ee580d4d98569a737df7accf03505a675e524c9e)

Ausgabe: Freitag, 15.12.2017
Abgabe: spätestens Mittwoch, 10.1.2018, 18:00

Thema: *Binäre Suchbäume*

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.
- Alle Abgaben müssen an der dafür vorgesehenen Stelle die Namen aller Gruppenmitglieder bzw. bei Einzelabgaben des Autors 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>@tubit/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!
- Benutze für alle Abgaben die in der jeweiligen Abgabe 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 Programmieraufgaben C-Quellcode ab und achte darauf, dass die Datei mit .c endet
- Bei Textaufgaben sind, wenn nicht anders angegeben, .txt Dateien zugelassen

1. Aufgabe: Binärer Suchbaum (1 Punkt)

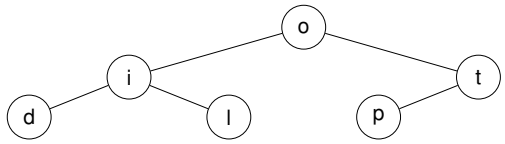

Abbildung 1: binärer Suchbaum

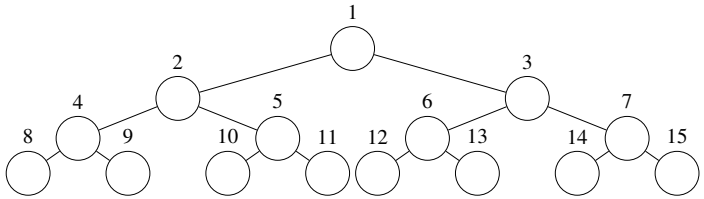

Abbildung 2: Nummerierung der Knoten im Baum

1.1. Löschen aus binären Suchbäumen Es sollen die folgenden Elemente nacheinander aus dem binären Suchbaum in Abbildung 1 (gemäß des Verfahrens aus der Vorlesung) gelöscht werden: (o), (i), (t), (p), (d)

Die Abgabe erfolgt über die Fragen 1-5 des Formulars `introprog_baum_operationen_vorgabe.txt`, welches ihr im SVN findet. Dabei wird der Zustand des Baums wie folgt beschrieben:

- 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 2).
- Um zu beschreiben, dass der Knoten mit dem Index 5 den Wert y enthält, verwenden wir die Notation "5:y".
- Die Angabe mehrerer Elemente erfolgt jeweils auf einer eigenen Zeile.
- Der Zustand des Baums aus Abbildung 1 wird somit wie folgt beschrieben:

```
1:o
2:i
3:t
4:d
5:l
6:p
```

- Beachte, dass die Nummerierung zwar von links nach recht vorgeht, aber auch nicht vorhandene Knoten mitgezählt werden (siehe Abbildung 3).

1.2. Einfügen in Bäume Welcher Baum ergibt sich, nachdem Du die folgenden Elemente in der gegebenen Reihenfolge in den Baum in Abbildung 1 einfügst?

(k), (v), (m), (u)

Trage den entsprechenden Zustand (nachdem alle Werte eingefügt wurden) gemäß der Beschreibung aus Aufgabe 1.1 als Antwort auf Frage 6 im Formular `introprog_baum_operationen_vorgabe.txt` ein und committe deine Antworten im SVN als `introprog_baum_operationen.txt`.

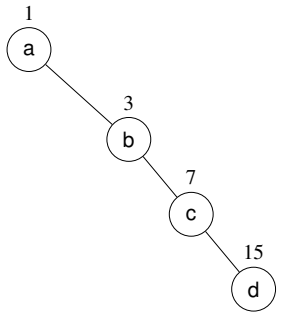

Abbildung 3: Weiteres Beispiel für die Nummeriung von Knoten

2. Aufgabe: Telefonbuch implementieren (2 Punkte)

In dieser Aufgabe soll ein Telefonbuch als binärer Suchbaum eingelesen, durchsucht und ausgegeben werden. Die Telefonnummern sind dabei natürliche Zahlen mit maximal neun Ziffern (also 1 und 999999999, aber nicht -2, 3, 5 oder 1234567890) und sollen so angeordnet werden, dass kleinere Zahlen immer links und größere Zahlen immer rechts platziert werden (siehe Abbildung 4). Es dürfen keine Zahlen mehrfach vorkommen.

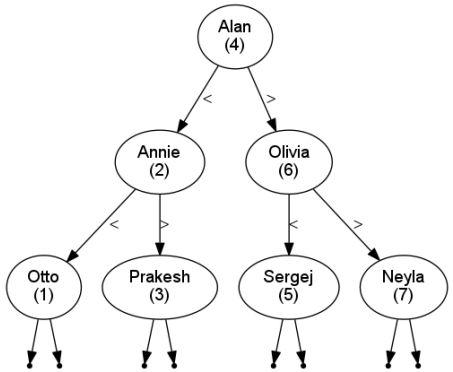

Abbildung 4: Beispiel-Telefonbuch als binärer Suchbaum

Du kannst Dich auf die Implementierung der folgenden Funktionen beschränken:

- `bst_insert_node(bstree* bst, unsigned long phone, char *name)`
  - Platziere einen neuen Knoten in den Baum `bst`.
  - Halte die Ordnung wie oben beschrieben ein.
  - Verändere, wenn nötig, den Pointer auf den Wurzelknoten.
  - Gebe eine Fehlernachricht aus und gehe mit **`return`** aus der Funktion, wenn versucht wird, eine Telefonnummer, die schon im Baum existiert, erneut einzugeben. `bst->root`.
- `bst_node* find_node(bstree* bst, unsigned long phone)`

  - Finde den Knoten im Baum, der die angegebene Telefonnummer besitzt, und gebe ihn zurück.
  - Gebe `NULL` zurück, wenn es keinen entsprechenden Knoten gibt.
- `bst_in_order_walk_node(bst_node* node)`
  - Gebe den Unterbaum von `node` in in-order Reihenfolge aus (`bst_in_order_walk(bstree* bst)` aufgerufen)
  - Benutze dafür die Funktion `print_node`. Sie benötigt als einzigen Parameter einen Pointer auf den gegenwärtigen Knoten (also im Format `bst_node*`).
- `bst_free_subtree(bst_node* node)`
  - Gebe den Speicher eines Teilbaums des binären Suchbaums frei (wird von `bst_free_tree(bstree* bst)` aufgerufen)
  - Achte dabei darauf, dass du den Baum "post-order" traversierst

Um die Aufgabe übersichtlicher zu gestalten, ist der Code auf vier Dateien verteilt:

introprog_input_telefonbuch.c enthält Eingabefunktionen und kleine Hilfsfunktionen. Hier solltest Du nichts verändern.

introprog_main_telefonbuch.c kommt dazu, um das Interface von der eigentlichen Logik zu trennen. Hier wird die `main` Funktion implementiert und auch hier solltest Du nichts verändern.

introprog_telefonbuch.c In dieser Datei sollst Du die oben genannten Funktionen implementieren. Neu ist dabei, dass sich die `main` Funktion und die **`struct`** in anderen Dateien befinden.

introprog_telefonbuch.h ist die Header Datei, die alle Dateien miteinander verknüpft. In dieser Aufgabe sind an dieser Stelle auch die **`struct`** und **`typedef`**, sowie die `includes` definiert.

Bei der Kompilierung muss neben wie gehabt `introprog_input_telefonbuch.c` auch zusätzlich die Datei `introprog_main_telefonbuch.c` angegeben werden. Das Programm benötigt als Parameter die Angabe des Telefonbuchs. Es sind dafür in den Vorgaben vorhanden:

telefonbuch_leer.txt Ein leeres Telefonbuch

telefonbuch_einfach.txt Ein kleines Telefonbuch-Beispiel

telefonbuch_gross.txt Ein großes Telefonbuch-Beispiel

Listing 1: Programmbeispiel

```
1  > gcc -std=c99 -Wall introprog_telefonbuch.c introprog_input_telefonbuch.c \
2                       introprog_main_telefonbuch.c -o introprog_telefonbuch
3  > ./introprog_telefonbuch telefonbuch_gross.txt
4   Fernsprech-Datensatz-System
5   ===========================
6    Füge in das Telefonbuch ein:  + <Nummer> <Name>
7    Gebe das Telefonbuch aus:     p
8    Finde den Namen:              ? <Nummer>
9    Debugausgabe des Baumes:      d
10   Beende das System:            q
```

Um das Debugging zu vereinfachen, haben wir eine Debugging-Funktionalität eingebaut. Mittels 'd' lässt sich beim Aufruf des Programms der binäre Suchbaum in eine PNG Datei ausgeben. Dafür wird die Software Graphviz benötigt. Auf den IRB Rechnern sollte die schon installiert sein, auf Heimrechnern muss die ggfs. nachinstalliert werden.

Benutze die folgende Codevorgabe:

Listing 2: Vorgabe `introprog_telefonbuch_vorgabe.c`

```c
/* === INTROPROG ABGABE ===
 * Blatt 6, Aufgabe 2
 * Tutorium: t00
 * Gruppe: g00
 * Gruppenmitglieder:
 *  - Erika Mustermann
 *  - Rainer Testfall
 * ========================
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "introprog_telefonbuch.h"

// Fügt einen Knoten mit der Telefonnummer phone und dem Namen name in den
// Binären Suchbaum bst ein.  Für den Suchbaum soll die Eigenschaft gelten,
   ↪ dass
// alle linken Kinder einen Wert kleiner gleich (<=) und alle rechten Kinder
// einen Wert größer (>) haben.
void bst_insert_node(bstree* bst, unsigned long phone, char *name) {
}

// Diese Funktion liefert einen Zeiger auf einen Knoten im Baum
// mit dem Wert phone zurück, falls dieser existiert. Ansonsten wird
// NULL zurückgegeben.
bst_node* find_node(bstree* bst, unsigned long phone) {
}

// Gibt den Unterbaum von node in "in-order" Reihenfolge aus
void bst_in_order_walk_node(bst_node* node) {
}

// Gibt den gesamten Baum bst in "in-order" Reihenfolge aus. Die Ausgabe
// dieser Funktion muss aufsteigend soriert sein.
void bst_in_order_walk(bstree* bst) {
    if (bst != NULL) {
        bst_in_order_walk_node(bst->root);
    }
}

// Löscht den Teilbaum unterhalb des Knotens node rekursiv durch
// "post-order" Traversierung, d.h. zurerst wird der linke und dann
// der rechte Teilbaum gelöscht. Anschließend wird der übergebene Knoten
// gelöscht.
void bst_free_subtree(bst_node* node) {
}
```



```c

// Löscht den gesamten Baum bst und gibt den entsprechenden Speicher frei.
void bst_free_tree(bstree* bst) {
    if(bst != NULL && bst->root != NULL) {
        bst_free_subtree(bst->root);
        bst->root = NULL;
    }
}
```