SWE2 Labor
Binäre Suchbäume (Binary-Search-Tree „BST“)
WS 2018/19

Aufgabe 1 (Einfügen und Suchen)
Binäre Suchbäume sind eine weitere wichtige Datenstruktur in der Computerwissenschaft. Im Gegensatz zu verketteten Listen, welche wir explizit sortieren mussten, ist die Ordnung eine inhärente Eigenschaft von binären Bäumen. Alle Werte der linken Kinder eines Knotens sind strikt kleiner als im Knoten selbst, und strikt größer für alle rechten. Das Konzept ist direkt verwandt mit der binären Suche (binary search). Außerdem sind sie ebenfalls eine dynamische Datenstruktur. Sie ermöglichen es, neue Elemente an beliebiger Stelle einzufügen oder zu entfernen, ohne das vorhandene Elemente verschoben werden müssen. Darin liegt auch ihr Vorteil gegenüber dynamischen Arrays. Anwendung finden Bäume als Teil größerer Datenstrukturen wie etwa Datenbanken oder ganze Dateisysteme.

Schreiben Sie eine eigene Implementierung eines BST, welcher aus nur einer Datenstruktur für einen Knoten und zugehöriger Funktionen besteht. Verwenden Sie folgende Protoypen in ihrer binary_tree.h Header-Datei:

```
struct node {
     /* 'val' can be any type, even another pointer! */
    int val;
    struct node* left;
    struct node* right;
};

struct node*    new_node(       int item);
struct node*    search(         struct node *root, int val);
void            print_inorder(  struct node *root);
void            print_preorder( struct node *root);
void            print_postorder(struct node *root);
struct node*    insert(         struct node* root, int val);
```

Diese Funktionen sind in der Datei binary_tree.c zu implementieren.
Durch den sich selbst wiederholenden Aufbau von BSTs bietet es sich an, rekursive Implementierungen für einige Funktionen zu wählen.
Implementieren sie `insert()`, `print_inorder()` und `search()` jeweils rekursiv.

überlegen Sie genau, was die einzelnen Funktionen ausführen müssen, um das Testprogramm auf der letzten Seite lauffähig zu machen (kommentieren sie `delete_node()` zum testen von Aufgabe 1 aus)

Aufgabe 2 (Löschen von Knoten)
Für das Löschen von Knoten müssen wir etwas mehr Fallunterscheidungen treffen. Implementieren Sie die Funktion

```
struct node*    successor(      struct node* node);
```

welche das in der Reihenfolge nächstgrössere Element im Baum zurückgibt. Diese dient wiederum als interne Hilfsfunktion für die Funktion

```
struct node*    delete_node(    struct node* root, int key);
```

delete_node() soll den Knoten mit dem entsprechend key-Wert löschen und freigeben und die Verknüpfungen korrekt neu setzen. Beide Funktionen solle ebenfalls rekursiv implementiert werden.

Erweitern Sie also ihre Header-Datei um die entsprechenden Prototypen und kommentieren Sie den entsprechenden Zeilen zum Testen der Funktionen in der main() wieder ein.

Aufgabe 3 (Mehrfach vorkommende Werte) [optional]

Erweitern Sie ihr Programm so, dass auch mehrfach vorkommende Werte in dem Baum gespeichert Werden können. Anstatt der naiven Lösung, bei der mehrfach vorkommende einfach in entweder den linken oder rechten Teilbaum einsortiert werden, sollen Sie allerdings eine verkettete Liste pro Knoten verwenden.

Die Verwendung einer verketteten Liste erlaubt es uns, unsere bestehende Implementierung mit nur wenigen Veränderungen beizubehalten. Sie dürfen Ihre Implementierung der verketteten Liste aus dem letzten Übungsblatt anpassen und verwenden.

Jeder Knoten im Baum soll eine eigene verkettete Liste aufbauen, wobei die Elemente der Liste jeweils alle den gleichen Wert wie der jeweilige Knoten des Baums haben.
Für einfache Integer-Werte ist dies natürlich etwas komplexer als es sein muss (man könnte auch einen einfachen Zähler verwenden). Sollen jedoch ganze Datenstrukturen anhand eines Schlüssels einsortiert werden, ist dies ein sehr guter Ansatz.

Für diesen integrierten Anwendungsfall können Sie auf die Nutzung der Header Datenstruktur verzichten. Neue Elemente werden beim einsortieren in den Baum einfach immer am Anfang der einfach verketteten Liste eingefügt.

Tipp: Genau genommen kann man sogar die Node-Datenstruktur als Listenelement missbrauchen, da sie den gleichen Inhalt haben (left/right bzw. next/prev sind lediglich beliebige Namen).

```
#include <stdio.h>
#include <stdlib.h>
#include "binary_tree.h"

int main()
{
     unsigned int i = 0;
     /* test values */
     int values[] = {50, 30, 20, 40, 70, 60, 80, 90};
     /* we need this to explicitly start the tree */
     struct node* root = NULL;
     struct node* res  = NULL;

    /* example tree
              50
           /     \
         30       70
        /  \     /  \
      20   40  60   80 */

    root = insert(root, values[0]);

    for(i=1; i<(sizeof(values)/sizeof(values[0])); i++) {
              insert(root, values[i]);
     }
     /* print the tree in order */
    print_inorder(root);

    /* search an item in the tree */
    res = search(root, 60);
    printf("Found %i in tree!\n", res->val);
    puts("Deleting 70...");
    delete_node(root, 70);
    print_inorder(root);
    return 0;
}
```