```c
/*
 ================================================================================
 Aufgabe     : Datenstrukturen
 Autor       : Erik Kaufmann
 Matrikel    : 1390365
 Version     : 1.0
 ================================================================================
 */
#include <stdbool.h>
#include <stdlib.h>
#include "dhbwstudent.h"
#include "dhbwstudenttree.h"

 // DIESE METHODEN NICHT AENDERN

StudentTP StudentTPAlloc(Student_p newStudent)
{
    StudentTP new = malloc(sizeof(StudentT));

    new->student = newStudent;
    new->lchild = NULL;
    new->rchild = NULL;

    return new;
}

void StudentTPFree(StudentTP tree)
{
    if (tree == NULL)
        return;

    StudentFree(tree->student);
    free(tree);
    return;
}

void StudentTFree(StudentTP* root_adr)
{

    StudentTP current = *root_adr;

    if (current == NULL)
    {
        return;
    }
    StudentTFree(&(current->lchild));
    StudentTFree(&(current->rchild));

    StudentTPFree(current);
```

```
50      return;
51  }
52
53  StudentTP deepTPCopy(StudentTP info)
54  {
55      if (info == NULL)
56          return NULL;
57      StudentTP copy = StudentTPAlloc(deepCopy(info->student));
58      copy->lchild = NULL;
59      copy->rchild = NULL;
60      return copy;
61  }
62
63  // Bis hier nicht ndern
64
65  // Ab hier Aufgaben
66
67  bool StudentTImplemented()
68  {
69      // TODO: hier auf true aendern, damit Ihre Implementierung
          getestet wird
70      return true;
71  }
72
73  bool StudentTContainsStudent(StudentTP* root_adr, Student_p student)
74  {
75      // Rekursiv nach dem Element suchen
76      // Hilfsmethode schreiben
77
78      // set root adr
79      StudentTP root = *root_adr;
80      bool foundStudent = false;
81
82      if (root->student->matrnr == student->matrnr) // gleich
83      {
84          //printf("<Duplicate student %s %d >\n", student->lastname,
              student->matrnr);
85          return true;
86      }
87      else if (root->student->matrnr > student->matrnr) // linker baum
88      {
89          root = root->lchild;
90      }
91      else // rechter Baum
92      {
93          root = root->rchild;
94      }
95
96      if (root != NULL)
97      {
98          foundStudent = StudentTContainsStudent(&root, student);
99      }
100
```

```
101     return foundStudent;
102 }
103
104 StudentTP StudentTFindByMatr(StudentTP* root_adr, int matrnr)
105 {
106     return NULL;
107 }
108
109 StudentTP StudentTFindByName(StudentTP* root_adr, char* lastname)
110 {
111     return NULL;
112 }
113
114 StudentLP* StudentTToSortedList(StudentTP* root_adr)
115 {
116     return NULL;
117 }
118
119 bool insertSorted(StudentT* studentNode, Student_p newStudent)
120 {
121     bool ret = false;
122
123     if (studentNode == NULL) // insertFirst
124     {
125         return true; // parentNode will be new StudentNode
126     }
127
128     else if ((studentNode)->student->matrnr > newStudent->matrnr) //?
        left tree
129     {
130         ret = insertSorted((studentNode)->lchild, newStudent);
131         if (ret)
132         {
133             if (StudentTContainsStudent(&studentNode, newStudent))
134             {
135                 return false; // student already exists
136             }
137             StudentTP newChild = StudentTPAlloc(newStudent);
138             (studentNode)->lchild = newChild;
139         }
140     }
141     else
142     {
143         ret = insertSorted((studentNode)->rchild, newStudent);
144         if (ret)
145         {
146             if (StudentTContainsStudent(&studentNode, newStudent))
147             {
148                 return false; // student exists
149             }
150             StudentTP newChild = StudentTPAlloc(newStudent);
151             (studentNode)->rchild = newChild;
152         }
```

```
153     }
154 }
155
156 void StudentTInsertSorted(StudentTP* root_adr, Student_p newStudent)
157 {
158     StudentTP current = *root_adr;
159     if (current == NULL) // insertFirst
160     {
161         *root_adr = StudentTPAlloc(newStudent);
162     }
163     else
164     {
165         insertSorted(current, newStudent);
166     }
167 }
168
169 int getTreeSize(StudentT* studentNode)
170 {
171     int counter = 0;
172
173     if (studentNode == NULL)
174     {
175         return 0;
176     }
177     else
178     {
179         counter = getTreeSize(studentNode->lchild) + getTreeSize
              (studentNode->rchild);
180     }
181
182     return counter + 1;
183 }
184
185 int StudentTSize(StudentTP* root_adr)
186 {
187     int counter = 0;
188
189     StudentTP current = *root_adr;
190
191     if (*root_adr == NULL)
192         return 0;
193     else
194     {
195         counter = getTreeSize(current);
196     }
197     // Return length of tree
198     // amount of all elements under the tree
199
200     return counter;
201 }
202
203 int getDepth(StudentT* studentNode)
204 {
```

```
205     if (studentNode == NULL)
206         return 0;
207
208     int tempDepthLeft = getDepth(studentNode->lchild);
209     int tempDepthRight = getDepth(studentNode->rchild);
210
211     /* use the larger one */
212     if (tempDepthLeft > tempDepthRight)
213         return (tempDepthLeft + 1);
214     else
215         return (tempDepthRight + 1);
216 }
217
218 int StudentTDepth(StudentTP* root_adr)
219 {
220     int maxDepth = 0;
221
222     if (root_adr == NULL)
223         return 0;
224     else
225     {
226         maxDepth = getDepth((*root_adr));
227     }
228
229     return maxDepth;
230 }
231
```