```c
/*
 ================================================================================
 Aufgabe     : Datenstrukturen - Woche 4
 Autor       : Erik Kaufmann
 Matrikel    : 1390365
 Version     : 1.0
 ================================================================================
 */
#include <stdbool.h>
#include <stdlib.h>
#include "dhbwstudentlist.h"
#include "dhbwstudent.h"

 // DIESE METHODEN NICHT AENDERN

StudentLP StudentLPAlloc(Student_p newStudent)
{
    StudentLP new = malloc(sizeof(StudentL));

    new->student = newStudent;
    new->next = NULL;

    return new;
}

void StudentLInsertFirst(StudentLP* anchor_adr, Student_p newStudent)
{
    StudentLP oldFirst = *anchor_adr;
    StudentLP newFirst = StudentLPAlloc(newStudent);
    *anchor_adr = newFirst;
    newFirst->next = oldFirst;
}

void StudentLInsertLast(StudentLP* anchor_adr, Student_p newStudent)
{
    StudentLP current = *anchor_adr;
    StudentLP previous = NULL;

    StudentLP newLP = StudentLPAlloc(newStudent);

    while (current != NULL)
    {
        previous = current;
        current = current->next;
    }

    if (previous != NULL)
    {
```

```
49          previous->next = newLP;
50      }
51      else
52      {
53          *anchor_adr = newLP;
54      }
55  }
56
57  int StudentLSize(StudentLP* anchor_adr)
58  {
59      StudentLP current = *anchor_adr;
60
61      int size = 0;
62
63      while (current != NULL)
64      {
65          size++;
66          current = current->next;
67      }
68
69      return size;
70  }
71
72  void StudentLPFree(StudentLP info)
73  {
74      if (info == NULL)
75          return;
76
77      StudentFree(info->student);
78      free(info);
79      return;
80  }
81
82  void StudentLFree(StudentLP* anchor_adr)
83  {
84      StudentLP current = *anchor_adr;
85
86      while (current)
87      {
88          StudentLP newCurrent = current->next;
89          StudentLPFree(current);
90          current = newCurrent;
91      }
92
93      *anchor_adr = NULL;
94      return;
95  }
96
97  StudentLP deepLPCopy(StudentLP info)
98  {
99      if (info == NULL)
100         return NULL;
101     StudentLP copy = StudentLPAlloc(deepCopy(info->student));
```

```
102     copy->next = NULL;
103     return copy;
104 }
105
106 StudentLP* deepLCopy(StudentLP* anchor_adr)
107 {
108     if (anchor_adr == NULL)
109         return NULL;
110
111     StudentLP* copy = malloc(sizeof(StudentLP));
112     *copy = NULL;
113
114     StudentLP current = *anchor_adr;
115
116     while (current)
117     {
118         StudentLInsertLast(copy, deepCopy(current->student));
119         current = current->next;
120     }
121
122     return copy;
123 }
124
125 StudentLP* StudentsFromFile(char* filename)
126 {
127     FILE* in = fopen(filename, "r");
128
129     char string[BUF_SIZE];
130     StudentLP* all_students = malloc(sizeof(StudentLP));
131     *all_students = NULL;
132     StudentLP reverse_students_anchor = NULL;
133     StudentLP* reverse_students = &reverse_students_anchor;
134
135     while (fgets(string, BUF_SIZE, in))
136     {
137         // remove newline (works for both windows and unix)
138         string[strcspn(string, "\r\n")] = 0;
139
140         // printf("%s\n", string);
141         StudentLInsertFirst(reverse_students, StudentAlloc(string));
142     }
143
144     // create normal ordered list
145     {
146         StudentLP current = *reverse_students;
147         while (current != NULL)
148         {
149             StudentLInsertFirst(all_students, deepCopy(current-
                  >student));
150             current = current->next;
151         }
152     }
153
```

```
154     StudentLFree(reverse_students);
155
156     return all_students;
157 }
158
159 // Bis hier nicht ndern
160
161 // Ab hier Aufgaben
162
163 bool StudentLImplemented()
164 {
165     // TODO: hier auf true aendern, damit Ihre Implementierung
          getestet wird
166     return true;
167 }
168
169 bool StudentLContainsStudent(StudentLP* anchor_adr, Student_p
      student)
170 {
171     // start from the first link
172     StudentLP current = *anchor_adr;
173
174     // if list is empty
175     if (current == NULL)
176         return NULL;
177
178     // iterate over linked list
179     while (current)
180     {
181         Student_p currentStud = current->student;
182
183         if (strcmp(currentStud->lastname, student->lastname) == 0 &&
              currentStud->matrnr == student->matrnr)
184         {
185             // lastname is equal
186             // matrnr is equal
187             // return current;
188             return true;
189         }
190
191         // NULL if reached the tail
192         if (current->next == NULL)
193             return NULL;
194         else
195             // get next
196             current = current->next;
197     }
198
199     return false;
200 }
201
202 StudentLP StudentLExtractStudent(StudentLP* anchor_adr, Student_p
      student)
```

```
203 {
204     // start from the first link
205     StudentLP current = *anchor_adr;
206     StudentLP before = NULL;
207 
208     // if list is empty
209     if (current == NULL)
210         return NULL;
211 
212     // iterate over linked list
213     while (current)
214     {
215         if (strcmp(current->student->lastname, student->lastname) ==
            0
216             && current->student->matrnr == student->matrnr)
217         {
218             // lastname is equal
219             // matrnr is equal
220             // extract
221             StudentLP temp = current;
222 
223             if (before)
224             {
225                 before->next = current->next;
226             }
227             else if (current->next)
228             {
229                 current = current->next;
230                 *anchor_adr = current;
231             }
232             else
233             {
234                 *anchor_adr = NULL;
235             }
236 
237             return temp;
238         }
239 
240         // NULL if reached the tail
241         if (current->next == NULL)
242         {
243             return NULL;
244         }
245         else
246         {
247             // get next
248             before = current;
249             current = current->next;
250         }
251     }
252 }
253 
254 StudentLP StudentLFindStudent(StudentLP* anchor_adr, Student_p
```

```
    student)
{
    // start from the first link
    StudentLP current = *anchor_adr;

    // if list is empty
    if (current == NULL)
        return NULL;

    // iterate over linked list
    while (current)
    {
        Student_p currentStud = current->student;

        if (strcmp(currentStud->lastname, student->lastname) == 0 &&
           currentStud->matrnr == student->matrnr)
        {
            // lastname is equal
            // matrnr is equal
            // return current;
            return current;
        }

        // NULL if reached the tail
        if (current->next == NULL)
            return NULL;
        else
            // get next
            current = current->next;
    }
}

StudentLP StudentLFindByMatr(StudentLP* anchor_adr, int matrnr)
{
    // start from the first link
    StudentLP current = *anchor_adr;

    // if list is empty
    if (current == NULL)
        return NULL;

    // iterate over linked list
    while (current)
    {
        Student_p currentStud = current->student;

        if (currentStud->matrnr == matrnr)
            // matrnr is equal
            return current;

        // NULL if reached the tail
        if (current->next == NULL)
            return NULL;
```

```
306         else
307             // get next
308             current = current->next;
309     }
310 }
311
312 StudentLP StudentLFindByName(StudentLP* anchor_adr, char* lastname)
313 {
314     // start from the first link
315     StudentLP current = *anchor_adr;
316
317     // if list is empty
318     if (current == NULL)
319         return NULL;
320
321     // iterate over linked list
322     while (current)
323     {
324         Student_p currentStud = current->student;
325
326         if (strcmp(currentStud->lastname, lastname) == 0)
327             // matrnr is equal
328             return current;
329
330         // NULL if reached the tail
331         if (current->next == NULL)
332             return NULL;
333         else
334             // get next
335             current = current->next;
336     }
337 }
338
339 void StudentLInsertSorted(StudentLP* anchor_adr, Student_p
      newStudent)
340 {
341     // start from the first link
342     StudentLP current = *anchor_adr;
343     StudentLP before = NULL;
344     StudentLP newStudentLP = NULL;
345
346     if (newStudent->matrnr == 0)
347     {
348         // skip corrupt data
349         // there should be no entry with matrNr == 0
350         return;
351     }
352
353     if (*anchor_adr == NULL)
354     {
355         // use existing method to insertFirst
356         StudentLInsertFirst(anchor_adr, newStudent);
357         return;
```

```
358     }
359
360     // iterate over linked list
361     // as long as current not null or nullptr
362     while (current)
363     {
364         if (current->next == NULL) // If current is tail
365         {
366             if (newStudent->matrnr < current->student->matrnr)
367             {
368                 newStudentLP = StudentLPAlloc(newStudent);
369
370                 if (before == NULL)
371                 {
372                     // Set newStudent as new head of list
373                     StudentLP oldHead = *anchor_adr;
374                     *anchor_adr = newStudentLP;
375                     newStudentLP->next = oldHead;
376                 }
377                 else
378                 {
379                     // place in between
380                     before->next = newStudentLP;
381                 }
382             }
383             else
384             {
385                 if (StudentLContainsStudent(anchor_adr, newStudent))
386                 {
387                     //avoid redundancy
388                     return;
389                 }
390                 else
391                 {
392                     StudentLInsertLast(anchor_adr, newStudent);
393                 }
394             }
395
396             return;
397         }
398         else // not tail
399         {
400             newStudentLP = StudentLPAlloc(newStudent);
401
402             if (current->student->matrnr < newStudent->matrnr &&
                  current->next->student->matrnr > newStudent->matrnr)
403             {
404                 newStudentLP->next = current->next;
405                 current->next = newStudentLP;
406                 return;
407             }
408
409             if (newStudent->matrnr < current->student->matrnr)
```

```
410                 {
411                     if (before == NULL) // current is head
412                     {
413                         StudentLInsertFirst(anchor_adr, newStudent);
414                     }
415                     else
416                     {
417                         before->next = newStudentLP;
418                         newStudentLP->next = current;
419                     }
420 
421                     return;
422                 }
423                 else
424                 {
425                     // set before and get next
426                     before = current;
427                     current = current->next;
428                 }
429             }
430         }
431 }
432 
```