Dr. habil. Bernd Schürmann
Dr. Annette Bieniusa
pinc-support@cs.uni-kl.de

TU Kaiserslautern
Fachbereich Informatik

Lösungshinweise zum Übungsblatt 11: Programmieren in C (WS 2018/19)

1. Dieses Übungsblatt ist ein **Pflichtübungsblatt**, d.h. Sie müssen bei der Bearbeitung dieses Blattes **mind. 50% der Gesamtpunkte** erreichen.
2. Die Abgabe erfolgt über das Exclaim-System gemeinsam mit Ihrem Teampartner.
3. Zur Beantwortung von Fragen und Hilfestellung bei der Bearbeitung kommen Sie bitte in die wöchentliche Fragestunde, Mittwoch 15:30, in Terminalraum 32-410.
4. Bitte laden Sie einzelne Dateien hoch, wie in der Aufgabenstellung angegeben, keine Archive, keine kompilierten Dateien.
5. Programme, die nicht kompilieren, werden nicht korrigiert und mit **0 Punkten** bewertet.
6. Bitte beachten Sie unsere Regelung bei Plagiaten:
   - Wenn Sie sekundäre Quellen, wie Bücher oder das Internet verwenden, müssen Sie immer die Quelle angeben. Das einfache Kopieren aus anderen Quellen ist für die Übungen nicht gestattet. Wenn wir in einer Übungsabgabe kopierten Code ohne Quellenangabe finden, wird die gesamte Abgabe mit 0 Punkten bewertet.
   - Sie können Übungsaufgaben gerne mit den Mitgliedern anderer Teams diskutieren. Sie sollten jedoch Ihren Code vor der Abgabefrist nicht an andere Teams weitergeben bzw. zeigen.
   - Wenn Code von anderen Teams kopiert wurde, werden die Abgaben **von beiden Teams** mit 0 Punkten bewertet.
   - Wir behalten uns vor Punkte auch nachträglich abzuziehen, wenn ein Verstoß erst später bemerkt wird.

Verkette Liste (15 Punkte)

Laden Sie sich die Datei `linkedlist.c` herunter, welche die Implementierung von einfach verketteten Listen aus der Vorlesung enthält. In dieser Aufgabe sollen Sie diese Liste um weitere Funktionen erweitern.

a) Schreiben Sie eine Funktion `bool list_is_sorted(linked_list_t *ll)`, welche prüft ob die in der Liste `ll` gespeicherten Werte aufsteigend sortiert sind.

```
bool list_is_sorted(linked_list_t *ll)
{
  node_t *a = ll->first;
  if (a == NULL)
  {
    return true;
  }
  node_t *b = a->next;
  while (b != NULL)
  {
    if (a->value > b->value)
    {
      return false;
    }
    a = b;
    b = b->next;
  }
  return true;
}
```

b) Schreiben Sie eine Funktion `bool list_has_duplicates(linked_list_t *ll)`, welche prüft, ob in der Liste `ll` Werte existieren, die mehrmals vorkommen.

```c
bool contains(node_t *node, int value)
{
  while (node != NULL)
  {
    if (node->value == value)
    {
      return true;
    }
    node = node->next;
  }
  return false;
}

bool list_has_duplicates(linked_list_t *ll)
{
  for (node_t *node = ll->first; node; node = node->next)
  {
    if (contains(node->next, node->value))
    {
      return true;
    }
  }
  return false;
}
```

c) Schreiben Sie eine Funktion `void list_add_before(linked_list_t *ll, int x, int y)`, welche einen neuen Eintrag mit Wert `x` direkt vor dem ersten Eintrag mit Wert `y` in die Liste einfügt. Wenn `y` nicht in der Liste enthalten ist soll `x` am Ende der Liste eingefügt werden.

```c
void list_add_before(linked_list_t *ll, int x, int y)
{
    node_t *new_node = malloc(sizeof(node_t));
    if (new_node == NULL)
  {
    printf("Couldn't allocate new node");
    exit(-1);
  }
  // Initialisiere den Wert des neuen Knotens
  new_node->value = x;

    // Einfuegen am Anfang der Liste
    if (ll->first == NULL || ll->first->value == y)
    {
  new_node->next = ll->first;
  ll->first = new_node;


    } else {
  node_t *nodePtr = ll->first;
  while (nodePtr->next != NULL && nodePtr->next->value != y)
  {
    nodePtr = nodePtr->next;
  }
    new_node->next = nodePtr->next;
    nodePtr->next = new_node;
  }

}
```

d) **[freiwillig, Achtung: schwierig]** Schreiben Sie eine Funktion `int list_remove(linked_list_t *ll, int value)`, welche alle Vorkommen von `value` aus der Liste `ll` entfernt und zurückgibt, wie viele Elemente entfernt wurden.
Die Liste soll dabei von der Funktion nur einmal durchlaufen werden.

```c
int list_remove(linked_list_t *ll, int value)
{
  int count = 0;
  node_t **nodePtr = &ll->first;
  while (*nodePtr != NULL)
  {
    node_t *node = *nodePtr;
```

```
    if (node->value == value)
    {
      *nodePtr = node->next;
      free(node);
      count++;
    }
    else
    {
      nodePtr = &node->next;
    }
  }
  return count;
}
```

Texteditor - Teil 1 (20 Punkte)

Diese Aufgabe bildet den Auftakt zur schrittweisen Entwicklung eines größeres Programms, das in den nächsten Übungsblättern fortgesetzt wird. Wir werden dazu einen einfachen konsolenbasierten Texteditor implementieren.

Schritt 1: Modulstruktur anlegen Es sollen folgende Module für Systemfunktionalität, die Ein-/Ausgabe, die Initialisierung und die Fehlerbehandlung angelegt werden:

`main.c`	Hauptprogramm
`main.h`	Globale Typen, Variablen und Konstanten
`editor_functions.c, editor_functions.h`	Funktionen des Editor
`sub_functions.c, sub_function.h`	Hilfsfunktionen
`in_out.c, in_out.h`	Ein- und Ausgabefunktionen
`error_handling.c, error_handling.h`	Ausgabe von Fehlermeldungen
`command.c, command.h`	Einlesen der nächsten Editorkommandos

Schritt 2: Einlesen der Benutzerkommandos Unser einfacher Editor soll über Kommandos der Standardeingabe gesteuert werden (vgl. Unix-Editor vi). Schreiben Sie hierzu eine Funktion, die das nächste Eingabekommando liest und überprüft und die Kommandodaten als Ergebnis zurückliefert.

Folgende Kommandos sind für unseren Editor vorgesehen und müssen eingelesen und erkannt werden.

`d<row>`	Lösche Zeile `<row>`
`i<row>/<text>`	Füge eine Zeile `<row>` mit dem Text `<text>` ein
`p` bzw. `p<row>`	Drucke den gesamten Text bzw. die Zeile `<row>` auf dem Bildschirm aus
`q`	Editor beenden (quit)
`r`	Daten zurücksetzen (reset)
`s/<text>`	Sucht eine Zeile, die `<text>` enthält und gibt diese aus

Implementieren Sie im Modul `command` eine Funktion

```
void next_command (char *command_char, int *line_no, char text[]);
```

die eine Kommandozeile mittels `fgets` einliest und den Kommandobuchstaben, die Zeilenangabe sowie den Text über die Parameter zurückliefert. Ist in einem Kommando keine Zeilenangabe bzw. Text vorhanden, wird Null bzw. eine leere Zeichenkette zurückgegeben. Sie können davon ausgehen, dass nur korrekte Eingaben vorliegen, d.h. Sie müssen noch nicht überprüfen, ob die Kommandostruktur korrekt ist (das kommt auf einem späteren Übungsblatt).

Beispiele für Kommandos: Die Kommandos sollen folgendermaßen aussehen (Reihenfolge ist für die Beispiele hier nicht wichtig):

```
d34
i1/Hallo Welt!
p
```

```
p1
q
r
s/Hallo
```

```c
#include <stdio.h>
#include "main.h"

void next_command (char * command_char, int * line_no, char text[]) {
        char command_string[2*LINE_LENGTH];
        int i, j;

        fgets(command_string, 2*LINE_LENGTH, stdin);
        *line_no = 0;
        text[0] = '\0';

        // Einlesen des Kommandos
        *command_char = command_string[0];

        // Einlesen der Zeilennummer
        i = 1;
        while(('0'<=command_string[i]) && (command_string[i]<='9')) {
                *line_no = (*line_no)*10 + (command_string[i]-'0');
                i++;
        }

        if (command_string[i]=='\0') {
                text[0] = '\0';
                return;
        };

        // Einlesen von Text
        i++; j = 0;
        while((command_string[i]!='\0') && (command_string[i]!='\n')) {
                text[j] = command_string[i];
                i++; j++;
        }
        text[j] = '\0';
}
```

Schritt 3: Anlegen und Initialisieren der globalen Datenstruktur in `main.h` Es soll ein globales zweidimensionales Array von Zeichen angelegt werden (`text_field`). Die Anzahl von Zeilen `LINE_COUNT` und Zeichen pro Zeile `LINE_LENGTH` sind als Präprozessor-Konstanten mit Wert 1000 bzw. 256 zu definieren. Weiterhin benötigen Sie eine Variable `max_line_no`, die die Anzahl beschriebener Zeilen speichert; diese hat initial den Wert 0. Die Verwendung von `max_line_no` erlaubt es uns, bei Ausgabe auf dem Bildschirm (Kommando `p`) den Text nur bis zur letzten beschriebenen Zeilen auszugeben.

Schritt 4: Hilfsfunktionen Implementieren Sie folgende Hilfsfunktionen zum Einfügen, Löschen und Kopieren von Zeilen (Module `sub_functions`):

`bool search_in_line (int line_no, char pattern[]);`	Testet, ob der String `pattern` in Zeile `line_no` vorkommt.
`void copy_line (int to, int from);`	Kopiert den Text der Zeile `from` in die Zeile `to`.
`void add_line (int line_no);`	Fügt eine leere Zeile hinzu. Hierzu müssen zunächst alle Zeilen von `line_no` bis zum Textende (`max_line_no`) um jeweils eine Zeile nach unten kopiert werden. [1]
`void delete_line (int line_no);`	Hierzu müssen die Zeilen `line_no+1` bis zum Textende (`max_line_no`) um jeweils eine Zeile nach oben kopiert werden. Zusätzlich muss eine leere Zeile am Textende eingefügt werden.

Bei den beiden letztgenannten Funktionen muss die globale Variable `max_line_no` aktualisiert werden. Testen Sie diese Hilfsfunktionen geeignet.

```c
#include <stdio.h>
#include <string.h>
#include <stdbool.h>
#include "main.h"
#include "sub_functions.h"

void copy_line (int to, int from) {
        int i;
        for (i=0; text_field[from-1][i]!='\0'; i++)
                text_field[to-1][i] = text_field[from-1][i];
        text_field[to-1][i] = '\0';
}

void add_line (int line_no) {
        int i;
        for (i=max_line_no; i>=line_no-1;i--)
                copy_line(i+1, i);
        text_field[line_no-1][0] = '\0';
        max_line_no = (max_line_no<line_no-1) ? line_no-1 : max_line_no+1;
}

void delete_line (int line_no) {
        int i;
        for (i=line_no+2; i<=max_line_no;i++)
                copy_line(i-1, i);

        max_line_no--;
}

bool contains(char *search, int searchSize, char *input, int inputSize)
{
    for (int i=0; i<inputSize-searchSize; i++) {
        bool found = true;
        for (int j=0; j<searchSize; j++)
        {
            if (input[i+j] != search[j])
            {
                found = false;
                break;
            }
        }
        if (found)
        {
            return true;
        }
    }
    return false;
}

bool search_in_line (int line_no, char* pattern) {
    return contains(text_field[line_no-1], strlen(text_field[line_no-1]),
    pattern, strlen(pattern));
}
```

Schritt 5: Editor-Funktionen Implementieren Sie folgende Funktionen im entsprechenden Modul `editor_functions`:

`void insert (int line_no, char text[]);`	Fügt den Text `text` in die Zeile `line_no` ein. Verwenden Sie hierzu die o.g. Hilfsfunktionen.
`void delete (int line_no);`	Löscht die Zeile `line_no`. Verwenden Sie hierzu die oben genannten Hilfsfunktionen aus dem Modul `sub_functions`.
`int search (char text []);`	Sucht die erste Zeile, die den Text `text` enthält und gibt diese aus.

Schritt 6: Ein-/Ausgabe und Fehlerbehandlung Das Modul `in_out` enthält zwei Ausgabefunktionen:

`void print_text (void);`	Gibt den gesamten Text auf dem Bildschirm aus.
`void print_line (int line_no);`	Gibt die Zeile `line_no` auf dem Bildschirm aus.

Bei der Ausgabe soll vor jede Zeile immer die Zeilennummer geschrieben werden (Zeilennummer mit 3 Positionen).

Beispiel:

```
  1  aaa
  2  bbb
  3  ccc
```

Das Fehlerbehandlungsmodul `error_handling` beinhaltet (zunächst) eine Funktion zur Fehlerausgabe, falls ein falsches Kommando eingegeben wird:

`void error_message (int error_no);`	Gibt eine geeignete Fehlermeldung aus.

Je nach Fehlernummer soll dazu eine folgende Fehlermeldung ausgegeben werden:

```
0  "Falsches Kommando"
1  "Zeilenangabe fehlt"
2  "Text fehlt"
```

Schritt 7: Globale Steuerung Schreiben Sie die Hauptfunktion `main`, die zunächst `text_field` initialisiert und dann die Benutzerinteraktion durchführt. Im Kern besteht die Mainfunktion aus eine Endlosschleife, die jeweils ein Kommando über `next_command` einliest und abhängig vom zurück gegebenen Kommandobuchstaben die entsprechende Editorfunktion aus dem Modul `editor_functions` aufruft. Beim Kommando 'q' wird das Programm beendet. Bei einem nicht definierten Editorbuchstaben (= falsche Eingabe) wird eine Fehlermeldung über `error_message` (s.o.). ausgegeben. Die in `main.h` global deklarierten Variablen `text_field` und `max_line_no` sind im Modul `main` zu definieren und zu initialisieren.

Schritt 8: Programmtest Testen Sie Ihr Programm ausführlich!

Hinweise:

- Bitte laden Sie die Datei `command.c` und `sub_function.c` gesondert als Abgabe für Aufgabe 2 und 3 in Exclaim hoch, damit diese einzeln getestet werden können. Diese dürfen die Implementierung der oben genannten Funktionen (sowie geeignete Hilfsfunktionen) beinhalten (aber **keine** Main-Funktion). Bitte achten Sie darauf, die Dateien und Funktionen richtig zu benennen. Das Testsystem in Exclaim bindet Ihre Implementierung dann in das jeweilige Testprogramm ein.
- Bei Aufgabe 4 müssen Sie alle Dateien aus Schritt 1 hochladen (d.h. Ihre gesamte Lösung!).
- Die Vorgaben der Modulstruktur und der Funktionsschnittstellen sind einzuhalten, was bei größeren Softwareprojekten in Betrieben typischerweise auch der Fall ist.
- Dokumentieren Sie Ihr Programm ausführlich! Auf den nächsten Übungsblättern wird die Arbeit an dem Texteditor weitergeführt.