Brandenburgische
Technische Universität
Cottbus - Senftenberg

Brandenburgische Technische Universität Cottbus – Senftenberg
Institut für Informatik, Informations- und Medientechnik
PD Dr.-Ing. habil. Thomas Hinze

Wintersemester 2015 / 2016

Abgabe der Lösungen: bis Mittwoch, 03.02.2016

Einführung in die Programmierung
Übungsblatt 6

Hinweise

- Erstellen Sie bitte eine schriftliche Lösung der nachfolgenden Aufgaben und erzeugen Sie daraus eine pdf-Datei. Sie können auch Ihre (lesbare!) handschriftliche Lösung als pdf einscannen. Achten Sie bitte darauf, dass am Beginn der ersten Seite des pdfs Ihr *Name*, Ihre *Matrikelnummer* und der Name Ihres *Tutors* (Laborübungsgruppe) vermerkt ist. Jede Aufgabe, die durch **(Laborübung)** gekennzeichnet ist, beinhaltet als Lösung auch einen C-Quelltext. Bitte packen Sie Ihr pdf gemeinsam mit allen C-Quelltextdateien in eine zip-Datei, die Sie termingemäß im moodle hochladen.
- Sie bekommen eine Bewertung Ihrer Lösung mit der Information, ob sie ausreichend zum Bestehen des Übungsblattes ist. Im Falle einer nichtausreichenden Lösung haben Sie Gelegenheit, eine verbesserte Version zeitnah, aber spätestens bis 10.02.16 im moodle nachzureichen. Plagiate werden nicht als ausreichende Lösungen anerkannt.

Aufgabe 1

(Laborübung) In natürlichsprachigen Texten variiert die Häufigkeit, mit der die einzelnen Buchstaben vorkommen. Beispielsweise enthalten deutsche Texte üblicherweise deutlich mehr Buchstaben *e* als *x*. Die prozentuale Verteilung der Buchstabenhäufigkeiten lässt sich als „Fingerabdruck“ der entsprechenden natürlichen Sprache auffassen. Für zahlreiche Sprachen hat man diese Häufigkeitsverteilung erstellt, indem eine Vielzahl von Texten aus dem Literaturschaffen, aber auch Alltagsjournalismus, Fachbücher und Schriftensammlungen akribisch ausgewertet wurden. In der *englischen* und in der *deutschen* Sprache ergibt sich daraus jeweils folgende prozentuale Verteilung, wobei die deutschen Umlaute *ä* durch *ae*, *ö* durch *oe*, *ü* durch *ue* sowie das Sonderzeichen *ß* durch *ss* ersetzt wurden. Groß- und Kleinschreibung wird nicht unterschieden:

	a	b	c	d	e	f	g	h	i	j	k	l	m
englisch	8.167	1.492	2.782	4.253	12.702	2.228	2.015	6.094	6.966	0.153	0.772	4.025	2.406
deutsch	6.51	1.89	3.06	5.08	17.40	1.66	3.01	4.76	7.55	0.27	1.21	3.44	2.53

	n	o	p	q	r	s	t	u	v	w	x	y	z
englisch	6.749	7.507	1.929	0.095	5.987	6.327	9.056	2.758	1.628	0.114	0.387	0.308	0.136
deutsch	9.78	2.51	0.79	0.02	7.00	7.58	6.15	4.35	0.67	1.89	0.03	0.04	1.13

Es zeigt sich, dass die Buchstabenhäufigkeiten im Deutschen etwas stärker streuen als im Englischen.

Schreiben Sie ein C-Programm, das eine Textdatei einliest und die Häufigkeiten der darin enthaltenen Buchstaben ermittelt. Es empfiehlt sich, den Inhalt der Textdatei in eine einzige Zeichenkette abzulegen. Die maximal zulässige Länge der Zeichenkette kann durch eine Konstante `MAX_DATEILAENGE` vorab festgelegt werden, zum Beispiel auf 100 000 Zeichen. Ist die eingelesene Datei länger, sollen nur die anfänglichen Zeichen berücksichtigt werden, bis die definierte Maximallänge erreicht ist. Liegt der auszuwertende Dateiinhalt in einer Zeichenkette vor, werden alle enthaltenen Großbuchstaben in Kleinbuchstaben umgewandelt.

Dazu steht die Funktion `int tolower(int c)` zur Verfügung, die in der Standardbibliothek `ctype.h` definiert ist. Anschließend werden die Buchstabenhäufigkeiten ausgezählt und dazu in einem gesonderten Feld abgelegt. Die Kleinbuchstaben decken als ASCII-Werte den Bereich von 97 (`'a'`) bis 122 (`'z'`) ab. Zusätzlich wird die Gesamtzahl Buchstaben bestimmt, so dass die prozentuale (relative) Häufigkeit jedes Buchstaben von `a` bis `z` berechnet werden kann. Seien `h['a']` bis `h['z']` die aus der eingelesenen Textdatei gewonnenen prozentualen Buchstabenhäufigkeiten. Ihr Programm soll diese Häufigkeitsverteilung ausgeben und abschließend die aufsummierte quadratische Abweichung sowohl zur Buchstabenverteilung in der englischen Sprache als auch zur Buchstabenverteilung in der deutschen Sprache unter Nutzung der obigen Tabellen bestimmen:

$$\texttt{abweichung\_englisch} = \sum_{\texttt{i='a'}}^{\texttt{'z'}} (\texttt{h[i]} - \texttt{englisch[i]}) \cdot (\texttt{h[i]} - \texttt{englisch[i]})$$

$$\texttt{abweichung\_deutsch} = \sum_{\texttt{i='a'}}^{\texttt{'z'}} (\texttt{h[i]} - \texttt{deutsch[i]}) \cdot (\texttt{h[i]} - \texttt{deutsch[i]})$$

Die Sprache mit der kleinsten aufsummierten quadratischen Abweichung ist dann die vom Programm prognostizierte Sprache des Textes. Testen Sie Ihr Programm anhand der Textdateien in `textsammlung.zip` aus. Dort enthalten ist die bissige Informatiker-Satire *Bastard Operator* sowie das Märchen *Hänsel und Gretel* jeweils in einer inhaltsgleichen deutschen und englischen Fassung. Zusätzlich steht die Datei `lorem-ipsum.txt` bereit, deren Inhalt weder deutsch noch englisch ist (sondern Pseudo-Latein). Protokollieren Sie bitte für alle Textdateien die aufsummierten quadratischen Abweichungen der Buchstabenhäufigkeiten zum Deutschen und zum Englischen und überzeugen Sie sich, dass die vom Programm vorgenommene Sprachenzuordnung (bis auf lorem ipsum) mit der tatsächlichen Sprache des jeweiligen Textes übereinstimmt.

Aufgabe 2

Lineare Listen bilden eine leicht programmierbare *dynamische Datenstruktur*, die sich zur Datenhaltung in zahlreichen Anwendungsszenarien vorteilhaft nutzen lässt. Insbesondere kleine Datenbanken, bei denen häufig Datensätze hinzugefügt oder gelöscht werden, profitieren von der Flexibilität linearer Listen. Im Gegensatz zum Feld, dessen Größe (Anzahl Feldelemente) sich nach dem Anlegen nicht mehr verändern kann, lassen sich lineare Listen während der Programmlaufzeit verkürzen oder verlängern. Ihr Speicherplatzbedarf passt sich somit gleitend den Erfordernissen der Datenbanknutzung an. Jeder Datensatz einer linearen Liste stellt ein *Listenelement* dar. Neben den Nutzerdaten enthält jedes Listenelement in einer *einfach verketteten* linearen Liste zusätzlich einen Verweis (Zeiger) auf das unmittelbar nachfolgende Listenelement. Beim letzten Element der Liste wird stattdessen jedoch der Vermerk `NULL` (Nullzeiger) eingetragen, wodurch das Ende der Liste markiert ist. Für die praktische Arbeit mit einer linearen Liste legt man einen globalen Verweis (Anker) auf das führende Listenelement an. Das Durchlaufen einer einfach verketteten linearen Liste erfolgt elementweise vom Anker aus. Während der Laufzeit des listenverwaltenden Programms können neue Listenelemente angelegt sowie bestehende Listenelemente gelöscht werden. Der Speicherbedarf der Liste variiert folglich bei der Programmabarbeitung und ist mithin dynamisch.

(Laborübung) Mithilfe einer einfach verketteten linearen Liste sollen die Ergebnisse einer Meisterschaft oder Spielsaison einer Mannschafts-Ballsportart (z.B. Fußballmeisterschaft) mit mehreren Spielen erfasst und ausgewertet werden. Ein Datensatz (Listenelement) besitzt dabei folgende Struktur:

```
struct TSpiel
{
  char mannschaft1[MAX_STRINGLAENGE]; //Name der ersten Mannschaft
  char mannschaft2[MAX_STRINGLAENGE]; //Name der zweiten Mannschaft
  int tore1;                          //Tore der ersten Mannschaft
  int tore2;                          //Tore der zweiten Mannschaft
  struct TSpiel *next;
};
```

Im Begleitmaterial finden Sie den Lückenquelltext `meisterschaft.c`, der einige nur als Dummies eingetragene Funktionen enthält. Ihre Aufgabe besteht darin, die Implementierungen dieser Funktionen zu vervollständigen (jeweils unter der Kommentarzeile `//Bitte hier Quelltext eintragen`):

- Die Funktion `fuegeSpielHinzu(char *m1, char *m2, int tore1, int tore2)` fügt ein neues Spiel mit den übergebenen Parametern am *Anfang* der Liste ein. `m1` ist der Name der Mannschaft1, `m2` der Name der Mannschaft2, `tore1` die Anzahl der von Mannschaft1 im Spiel erzielten Tore und `tore2` entsprechend von Mannschaft2. Bei ungültigen übergebenen Werten (negative Toranzahlen oder leere Zeichenketten) bricht die Funktionsabarbeitung ab und gibt den Fehlerwert 1 zurück, ebenso, wenn nicht genügend Speicherplatz zum Anlegen des neuen Listenelementes vorhanden ist. Bei erfolgreicher Abarbeitung liefert die Funktion den Rückgabewert 0.
- Die Funktion `anzahlSpiele()` liefert die Anzahl der in der gesamten Liste erfassten Spiele zurück. Bei leerer Liste wird die Zahl 0 zurückgegeben.
- Die Funktionen `gibMannschaft1(int index, char **m)` und `gibMannschaft2(int index, char **m)` stellen in `*m` den Namen der Mannschaft1 bzw. Mannschaft2 des Listenelements an der Indexposition `index` bereit. Bei erfolgreicher Ausführung wird zudem der Wert 0 zurückgegeben. Im Fehlerfall (ungültige Indexposition) erfolgt die Rückgabe des Wertes 1.
- Die Funktionen `gibTore1(int index, int *t)` und `gibTore2(int index, int *t)` liefern zum Spiel (Listenelement) an der Indexposition `index` die jeweils hinterlegte Toranzahl in `*t`.
- Die Funktion `gesamtzahlTore(void)` ermittelt die Gesamtzahl Tore über alle erfassten Spiele.
- Die Funktion `zeigeSpiellisteAn(void)` gibt die gesamte Spielliste als Tabelle am Bildschirm aus.
- Die Funktion `setzeMeisterschaft(void)` löscht zunächst alle Spiele, die ggf. bereits in der Liste enthalten sind. Anschließend werden durch hintereinander vorgenommene Aufrufe der Funktion `fuegeSpielHinzu` Spielergebnisse erfasst, die dann als fertig befüllte Liste für weitere Bearbeitungen oder Auswertungen zur Verfügung stehen. Tragen Sie dazu die Ergebnisse einer Meisterschaft oder Spielsaison (mindestens 12 Spiele) einer Mannschafts-Ballsportart Ihrer Wahl ein (z.B. Fußball, Handball, Volleyball, Basketball, Football, Eishockey, Wasserball, ...). Bei internationalen Meisterschaften können Sie für die Mannschaftsnamen Länderabkürzungen verwenden wie z.B. `DE` für Deutschland oder `BR` für Brasilien.

Es bieten sich die Daten der *Fußball-Weltmeisterschaft 2014* an (`https://de.wikipedia.org/wiki/Fu%C3%9Fball-Weltmeisterschaft_2014`) mit insgesamt 32 beteiligten Ländern und 64 Spielen ebenso wie die

Fußball-Weltmeisterschaft der Frauen 2015 (`https://de.wikipedia.org/wiki/Fu%C3%9Fball-Weltmeisterschaft_der_Frauen_2015`) mit 24 Ländermannschaften und 52 Spielen.

Die `main`-Funktion mit einer einfachen Menüführung sowie weitere nützliche Funktionen sind in der Quelltextvorgabe bereits fertig implementiert. Nachdem Sie die oben genannten Funktionen vervollständigt haben, können Sie die Mannschaften absteigend nach der Anzahl erzielter Tore anzeigen lassen (vorimplementierte Funktion `zeigeMannschaftenMaxTore`) sowie die Spielliste in eine Textdatei exportieren (vorimplementierte Funktion `speichereListe`). Darüber hinaus können weitere Spiele hinzugefügt, Spiele gelöscht und die Spielliste gemeinsam mit der Gesamtzahl Spiele und der Gesamtzahl Tore angezeigt werden.