Verkettete Listen

Allgemeine Anmerkungen zur Aufgabenstellung:

- Beantworten Sie die Kontrollfragen in `TUWEL` bevor Sie mit der Übung beginnen.
- Legen Sie für `Teil A` der Angabe ein neues `Code::Blocks`-Projekt an und erweitern Sie dieses schrittweise für die darauffolgenden Punkte.
- Stellen Sie während der Übung laufend sicher, dass Ihr Programm kompilierbar ist und richtig funktioniert.
- Die Abgabe erfolgt durch Hochladen Ihres vollständigen `Code::Blocks`-Projektordners als zip-Datei (`Matrikel-Nr_UE2.zip`) in `TUWEL`. Anschließend melden Sie sich für das Abgabegespräch in `TUWEL` an. Dieses erfolgt über die Telekonferenz-App `Zoom` (www.zoom.us). Stellen Sie sicher, dass diese App rechtzeitig vor Beginn des Abgabegespräches auf Ihrem PC installiert ist und dass Sie Ihr aktuelles Übungsprogramm in `Code::Blocks` für die Abgabe geöffnet haben. Testen Sie bitte auch die korrekte Funktionalität Ihrer Videokamera, da Sie zu Beginn Ihres Abgabegespräches Ihren Studentenausweis herzeigen müssen und auch während des Gesprächs eine aktive Videoverbindung erforderlich ist.
- Beachten Sie, dass Upload und Terminauswahl für das Abgabegespräch erst möglich sind, wenn Sie 80% der Kontrollfragen richtig beantwortet haben. Bis **09.11.2021 23:59** müssen diese Fragen positiv beantwortet, Ihre Ausarbeitung hochgeladen und ein Termin für das Abgabegespräch ausgewählt werden.
- Inhaltliche Fragen stellen Sie bitte in der Fragestunde oder im zugehörigen Forum in `TUWEL`.
- Für eine positive Beurteilung Ihrer Abgabe **muss** diese kompilierbar sein, Ihr Programm fehlerfrei terminieren und die in der Angabe angeführten Funktionen mit den vorgegebenen Funktionsköpfen enthalten. Des Weiteren müssen die Funktionen hinreichend getestet sein. Sie müssen in der Lage sein, Ihr Programm beim Abgabegespräch zu erklären!

Folgende Hilfsmittel stehen Ihnen in `TUWEL` zur Verfügung:

- Alle bisherigen Referenzbeispiele sowie Vorlesungsfolien
- Interaktive Jupyter-Notebooks auf `prog1.iue.tuwien.ac.at`
- Das Buch zur LVA "Programmieren in C" (Robert Klima, Siegfried Selberherr)
- Kurzanleitung für die Entwicklungsumgebung `Code::Blocks`

Hinweis: Es wird erwartet, dass alle Abgaben zu den Übungen **eigenständig** erarbeitet werden und wir weisen Sie darauf hin, dass alle Abgaben in einem standardisierten Verfahren auf Plagiate überprüft werden. Bei Plagiatsvorfällen entfällt das Abgabegespräch und es werden keine Punkte für die entsprechende Abgabe vergeben.

Stoffgebiet
Zusätzlich zum gesamten Lehrinhalt der Lehrveranstaltung Programmieren 1 benötigen Sie für diese Übung folgenden Stoff:

- Speicher reservieren und freigeben (Kapitel 20)
- Datenstrukturen (Einfach und doppelt verkettete Liste) (Kapitel 19)
- Zeiger auf Funktionen (Kapitel 14.7)

Einleitung
In dieser Übung werden Sie Ihr erworbenes Wissen zu dem Themengebiet dynamischer Speicher aus der 1. Übungseinheit und der Vorlesung verwenden, um eine einfache Wiedergabeliste (Playlist) zu erstellen. Eine Playlist kann zum Beispiel folgendermaßen aussehen:

```
  Pos   Interpret       Titel
----------------------------------------
    1   The Dubliners   The Wild Rover
    2   Pavarotti       O Sole Mio
    3   Percy Sledge    My Special Prayer
```

DUMA Bibliothek
Die unsachgemäße Verwendung von dynamischen Speicherbereichen kann zu schwer auffindbaren Programmfehlern zur Laufzeit führen. Um die Fehlersuche zu vereinfachen und die korrekte dynamische Speicherverwaltung sicherzustellen, verwenden Sie die `DUMA` Bibliothek. Mit Hilfe dieser Bibliothek bekommen Sie eine Rückmeldung über die Verwendung von dynamisch alloziertem Speicher. Diese Bibliothek ersetzt die Systemfunktionen `malloc()`, `calloc()`, `realloc()` und `free()` durch eigene Varianten, die bei falscher Verwendung unmittelbar eine Fehlermeldung zurück liefern. Für die Übung haben wir `DUMA` so verändert, dass über eine Umgebungsvariable (`DUMA_MALLOC_ERROR`, siehe unten) die Fehlerwahrscheinlichkeit für die Speicherallozierung eingestellt werden kann.

Um die `DUMA`-Bibliothek zu installieren und in Ihrem Projekt zu verwenden, folgen Sie den Anweisungen in der auf `TUWEL` zur Verfügung gestellten `DUMA`-Anleitung.

Um ein gescheitertes Anfordern von Speicherbereichen zu simulieren, starten Sie unter Linux Ihr Programm in der Konsole mit dem Befehl

```
DUMA_MALLOC_ERROR=50 ./<Programmname>
```

Unter Windows verwenden Sie `cmd` und starten Ihr Programm wie folgt

```
set DUMA_MALLOC_ERROR=50
<Programmname>.exe
```

Die Umgebungsvariable `DUMA_MALLOC_ERROR` gibt dabei die Fehlerwahrscheinlichkeit in Prozent an. In diesem Beispiel würde also im Mittel jede zweite Speicheranforderung fehlschlagen.

Hinweis: Beachten Sie, dass während des Abgabegesprächs auch die korrekte Verwendung der dynamischen Speicherverwaltung überprüft wird. Achten Sie daher u.a. auf Speicherlecks und korrekte Freigaben. Ihr Programm muss bei einer gescheiterten Speicher-Allozierung eine definierte Fehlerbehandlung durchführen und darf unter **keinen Umständen** abstürzen!

Teil A

- ✘ Fügen Sie die `DUMA` Bibliothek zu Ihrem Projekt hinzu bevor Sie mit der eigentlichen Implementierung beginnen. Die genauen Anweisungen zur Installation und Verwendung dieser Bibliothek finden Sie in der zugehörigen `DUMA`-Anleitung auf `TUWEL`.

- ✘ Erstellen Sie eine *doppelt verkettete Liste*, welche Songtitel und Interpreten abspeichern kann. Dazu erstellen Sie den Verbunddatentyp:

```
typedef struct Element_s {
char songtitle[256], interpreter[256];
struct Element_s *next, *prev;
} Element_t;
```

- ✘ Implementieren Sie die Funktion

```
void read_from_keyboard(Element_t *item);
```

  welche `songtitle` und `interpreter` von der Tastatur einliest und dem Element `item` zuweist. Überpruefen Sie auch, ob es sich bei `item` um einen gültigen Zeiger handelt.

- ✘ Implementieren Sie die Funktion

```
Element_t *allocate_element();
```

  welche Speicherplatz für ein neues Element reserviert und die Funktion `read_from_keyboard` benutzt, um dafür Werte einzulesen. Geben Sie einen Zeiger auf dieses neue Element zurück und stellen Sie sicher, dass die Zeiger des Elements richtig initialisiert werden.

- ✘ Um Elemente in die Liste einzufügen, implementieren Sie als nächstes die Funktion

```
Element_t *insert_last(Element_t *list);
```

  welche ein neues Listenelement mit Hilfe der Funktion `allocate_element` erzeugt und an das Ende der Liste `list` anfügt. Sollte `list == 0` sein, legen Sie eine neue Liste an. Der Rückgabewert ist ein Zeiger auf den Listenanfang.

- ✘ Implementieren Sie weiters die Funktion

```
void free_list(Element_t **list);
```

  welche die übergebene Liste `*list` wieder korrekt freigibt, sowie den Zeiger auf die Liste auf Null setzt.

- ✘ Um die Liste anzuzeigen, implementieren Sie die Funktionen

```
void print_single_element(Element_t *element);
```

  und

```
void print_entire_list(Element_t *list);
```

  die ein einzelnes Element bzw. die ganze Liste ausgeben. Etwa in der Form

```
1. Titel1 Interpret1
2. Titel2 Interpret2
...
```

✘ Erstellen Sie ein Hauptprogramm, in dem Sie zuerst eine leere Liste anlegen und danach ein Menü der folgenden Form implementieren:

```
0. Programm beenden
1. Neuen Eintrag einlesen und am Ende einfügen
2. Liste ausgeben
3. Liste löschen
```

Teil B

✘ Implementieren Sie die Funktion

```
long elements_count(Element_t *list);
```

welche die Anzahl der Elemente in der Liste `list` zurückgibt.

✘ Implementieren Sie die Funktion

```
Element_t *element_at(Element_t *list, long index);
```

welche einen Zeiger auf das Element an der Stelle `index` der Liste `list` zurückgibt. `index` = 0 ist dabei das 1. Element!

✘ Erweitern Sie Ihr Hauptprogramm um die Menüpunkte:

```
4. Anzahl der Elemente ausgeben
5. Element an der Stelle x ausgeben
```

Teil C

✘ Implementieren Sie die Funktion

```
Element_t *insert_at(Element_t *list, long index);
```

die ein neues Element nach dem Element an der Stelle `index` in die Liste `list` einfügt und einen Zeiger auf den Listenanfang zurückgibt. Zum Erzeugen des neuen Elements benutzen Sie die Funktion `allocate_element`. Falls `index` einen ungültigen Wert hat, geben Sie eine entsprechende Fehlermeldung aus und lassen die Liste unverändert.

✘ Erweitern Sie Ihr Hauptprogramm um den Menüpunkt:

```
6. Neuen Eintrag einlesen und an der Stelle x einfügen
```

Teil D

✘ Implementieren Sie die Funktion

```
Element_t *delete_at(Element_t *list, long index);
```

die das Element an der Position `index` der Liste `list` löscht und einen Zeiger zum Listenanfang zurückgibt. Falls `index` einen ungültigen Wert hat, geben Sie eine entsprechende Fehlermeldung aus und lassen die Liste unverändert.

✘ Erweitern Sie Ihr Hauptprogramm um den Menüpunkt:

```
7. Eintrag an der Stelle x löschen
```

Teil E

✘ Implementieren Sie die Funktion

```
void print_list_reverse(Element_t *list);
```

welche die Liste `list` in der umgekehrten Reihenfolge auf dem Bildschirm ausgibt ohne die Originalliste zu verändern.

✘ Implementieren Sie eine Funktion

```
void print_list_random(Element_t *list);
```

welche die Liste `list` in einer zufälligen Reihenfolge auf dem Bildschirm ausgibt ohne dabei die Originalliste zu verändern.

✘ Erweitern Sie Ihr Hauptprogramm um die Menüpunkte:

```
8. Liste in umgekehrter Reihenfolge ausgeben
9. Liste in zufälliger Reihenfolge ausgeben
```

Teil F

✘ Definieren Sie einen Datentyp für Funktionen (Funktionszeiger), welche zwei Listenelemente miteinander vergleicht.

```
typedef long (*element_compare) (Element_t *, Element_t *);
```

Eine solche Funktion liefert einen Wert zurück, der angibt, welches der beiden Elemente anhand eines bestimmten Attributes (bspw. `songtitle`) "größer" ist.

✘ Implementieren Sie die Funktionen

```
long compare_songtitles_larger(Element_t *A, Element_t *B);

long compare_songtitles_smaller(Element_t *A, Element_t *B);
```

welche zurückgeben sollen, ob `songtitle` von `A` lexikographisch größer/kleiner ist als jener von `B`.

✘ Implementieren Sie die Funktion

```
Element_t *sort_list(Element_t *list, element_compare fkt);
```

die die Liste anhand der übergebenen Funktion `fkt` sortiert (bspw. mit *Bubblesort*). Der Rückgabewert soll ein Zeiger auf den neuen Listenkopf sein.

Hinweis: Aus Performancegründen sollen beim Sortieren nicht die gespeicherten Elementdaten kopiert, sondern lediglich die Zeiger auf nachfolgende oder vorherige Listenelemente (`next` und `prev`) umgelenkt werden. Stellen Sie sicher, dass Ihnen keine Listenelemente verloren gehen, achten Sie auf die Reihenfolge der Zeigerupdates und einen konsistenten Anfang der Liste.

✘ Erweitern Sie Ihr Hauptprogramm um den Menüpunkt:

```
10. Liste aufsteigend nach songtitle sortieren
11. Liste absteigend nach songtitle sortieren
```

Teil G

✘ Implementieren Sie die Funktion

```
Element_t *unique_list(Element_t *list);
```

Diese Funktion soll *alle* Duplikate aus `list` entfernen. Zwei Elemente sollen dabei als gleich angesehen werden, wenn sowohl `songtitle` als auch `interpreter` identisch sind. Der Rückgabewert der Funktion ist der (eventuell veränderte) Listenanfang.

✘ Erweitern Sie Ihr Hauptprogramm um den Menüpunkt:

```
12. Duplikate aus Liste entfernen
```

Teil H

✘ Implementieren Sie die Funktion

```
Element_t *rotate_list(Element_t *list, long shift);
```

welche die Liste um `shift` rotiert. Beispielsweise soll die Liste

```
A<->B<->C<->D<->E
```

nach dem Rotieren um `shift=2` wie folgt aussehen:

```
D<->E<->A<->B<->C
```

Der Rückgabewert der Funktion ist der neue Listenanfang.

✘ Erweitern Sie Ihr Hauptprogramm um die Menüpunkte:

```
13. Liste um x rotieren
```