Human-Computer Interaction Group

Leibniz Universität Hannover

Programmieren 1 – WS 2018/19

Prof. Dr. Michael Rohs, Tim Dünte, M.Sc.

Übungsblatt 10

Alle Übungen (bis auf die erste) müssen in Zweiergruppen bearbeitet werden. Beide Gruppenmitglieder müssen die Lösung der Zweiergruppe einzeln abgeben. Die Namen beider Gruppenmitglieder müssen sowohl in der PDF Abgabe als auch als Kommentar in jeglichen Quelltextabgaben genannt werden. Wenn Sie für Übungsblatt 1 noch keinen Gruppenpartner haben, geben Sie alleine ab und nutzen Sie das erste Tutorium dazu, mit Hilfe des Tutors einen Partner zu finden. Plagiate führen zum Ausschluss von der Veranstaltung.

Abgabe bis Donnerstag den 10.01. um 23:59 Uhr über https://assignments.hci.uni-hannover.de/WiSe2018/Programmieren1. Die Abgabe muss aus einer einzelnen zip-Datei bestehen, die den Quellcode, ein pdf bei Freitextaufgaben und alle weiteren nötigen Dateien (z.B. Eingabedaten oder Makefiles) enthält. Lösen Sie Umlaute in Dateinamen bitte auf.

Die Dokumentation der Prog1lib finden Sie unter: https://hci.uni-hannover.de/files/prog1lib/index.html

Aufgabe 1: Operationen auf Listen

Das Template für diese Aufgabe ist list.c. In dieser Aufgabe sollen verschiedene Funktionen auf einer einfach verketteten Liste mit dynamisch allokierten Elementen vom Typ String implementiert werden. Die Struktur `Node` für die Listenknoten ist vorgegeben. Die Nutzung von `xmalloc` statt `malloc` und `xcalloc` statt `calloc` ist verpflichtend, damit das System die korrekte Freigabe des Speichers überprüfen kann. Achten Sie darauf, dass keine Speicherlecks auftreten.

a) Beschreiben Sie die Funktionsweise der Funktion `to_list` sowie der Hilfsfunktionen `skip_whitespace` und `skip_token`. Beantworten Sie in Ihrer Beschreibung insbesondere die folgenden Teilfragen: Warum genügt es, dass die skip-Funktionen nur einen einzelnen Parameter (den String s) übergeben bekommen und warum ist ein Index als zusätzlicher Parameter nicht notwendig? Warum wird von den skip-Funktionen ein Zeiger zurückgegeben und kein Index? Wozu verwendet `to_list` den Zeiger `last`? Warum wird der von `s_sub` zurückgegebene Teilstring am Ende der Schleife wieder freigegeben? Wozu dient die Prüfung von `first` auf `NULL`? Worauf zeigen `a` und `b` jeweils am Ende der Schleife?

b) Implementieren Sie die Funktion `free_list`, die eine List mitsamt Inhalt freigibt. Beachten Sie, dass die Liste Besitzer („owner") der Listenelemente (`value`) ist und alle Listenelemente dynamisch allokiert sind. Diese müssen also auch freigegeben werden.

c) Implementieren Sie die Funktion `bool test_equal_lists(int line, Node* list1, Node* list2)`, die überprüft, ob die beiden Listen inhaltlich gleich sind, ob sie also die gleichen Elemente haben. Die Funktion soll genau dann `true` zurückgeben, wenn dies der Fall ist. Außerdem soll sie das Ergebnis in folgender Form ausgeben:

- `Line 78: The lists are equal.`
- `Line 82: The values at node 1 differ: second <-> hello.`
- `Line 86: list1 is shorter than list2.`
- `Line 90: list1 is longer than list2.`

Eine Testfunktion mit Beispielaufrufen existiert bereits (`test_equal_lists_test`). Diese muss nicht verändert werden. Die Funktion illustriert auch, wie Listen mit `new_node` generiert werden.

Hinweis: Die String-Elemente können mit `s_equals` oder `strcmp` verglichen werden.

d) Fügen Sie drei weitere Testbeispiele zur Funktion `to_list_test` hinzu. Diese soll Beispielaufrufe der Funktion `to_list` enthalten. Verwenden Sie `test_equal_lists` und achten Sie darauf, dass die angelegten Testlisten mit `free_list` wieder freigegeben werden, so dass keine Speicherlecks entstehen.

e) Implementieren Sie die Funktion `int char_count_in_list(Node* list, char c)`. Diese soll ermitteln, wie oft das Zeichen `c` in der Liste auftaucht.

f) Implementieren Sie die Funktion `Node* remove_list(Node* list, int index)`. Diese soll den Knoten an Position `index` löschen (und dessen Speicher freigeben) und die resultierende Liste zurückgeben. Wenn der Index ungültig ist, soll die Liste nicht verändert werden. Fügen Sie außerdem drei weitere Beispielaufrufe in der zugehörigen Testfunktion hinzu. Achten Sie darauf, dass keine Speicherlecks entstehen.

g) Implementieren Sie die Funktion `Node* insert_in_list(Node *list, Node* list_to_insert, int position)`. Diese soll die `list_to_insert` an Indexposition `position` in `list` einfügen. Für eine Position kleiner oder gleich Null soll die Liste vorne eingefügt werden. Wenn die Position größer oder gleich der Länge von `list` ist, soll die Liste hinten angehängt werden. Ansonsten soll sie an der entsprechenden Position dazwischen eingefügt werden. Es empfiehlt sich, diese drei Fälle in der Implementierung separat zu behandeln. Beachten Sie auch die Fälle, dass `list` leer ist bzw. dass `list_to_insert` leer ist. Eine Testfunktion mit Beispielaufrufen ist bereits gegeben.

Aufgabe 2: Mensa

Das Template für diese Aufgabe ist mensa.c. In dieser Aufgabe soll eine interaktive Mensasimulation geschrieben werden. Die Mensa bietet auf der Tageskarte fünf verschiedene Gerichte an (`const String menu[]`). Bevor die ersten Studierenden kommen, kocht die Küche bereits fünf zufällige Gerichte von der Tageskarte. Jedes Mal, wenn ein Gericht an einen Studierenden ausgegeben wurde, kocht die Küche ein weiteres zufälliges Gericht. Die fertigen Gerichte sind in der String-Liste `food` gespeichert. Es stehen zunächst 3 Studierende in der Schlange und warten auf Essen. Die Studierenden sind in der String-Liste `students` durch ihre Essenswünsche repräsentiert. Der Benutzer der Simulation spielt die Rolle einer bzw. eines Mensa-Bediensteten und hat das Ziel, durch Ausgabe der gewünschten Gerichte die Reputation der Mensa zu maximieren. Wenn ein Essenswunsch erfüllt wurde, steigt die Reputation um eins. Jedes Mal, wenn die Reputation der Mensa steigt, kommt ein weiterer Studierender dazu. Wenn ein falsches Gericht ausgegeben wurde, sinkt die Reputation um eins und das Essen wird zurückgenommen. Wenn ein Essenswunsch nicht erfüllt werden kann, weil das entsprechende Gericht gerade nicht fertig ist, sinkt die Reputation der Mensa um 2. Die/der Studierende verlässt daraufhin die Mensa ohne gegessen zu haben. Hier ein möglicher Simulationsablauf:

```
fertige Essen: [Salat, Vegi, Vegi, Spaghetti, Eintopf]
nächster Essenswunsch: Spaghetti (3 hungrige Studierende warten)
Reputation der Mensa: 0
> 3 (← Mitarbeiter/in nimmt fertiges Essen an Position 3, Spaghetti entspricht dem Wunsch)
Vielen Dank! Ich liebe die Mensa!
fertige Essen: [Salat, Vegi, Vegi, Eintopf, Eintopf]
nächster Essenswunsch: Spaghetti (3 hungrige Studierende warten)
Reputation der Mensa: 1
> -1 (← Mitarbeiter/in antwortet „haben wir gerade nicht")
Spaghetti ist nicht da? Schade.
fertige Essen: [Salat, Vegi, Vegi, Eintopf, Eintopf]
nächster Essenswunsch: Salat (2 hungrige Studierende warten)
Reputation der Mensa: -1
> 0 (← Mitarbeiter/in nimmt fertiges Essen an Position 0, Salat entspricht dem Wunsch)
Vielen Dank! Ich liebe die Mensa!
fertige Essen: [Vegi, Vegi, Eintopf, Eintopf, Eintopf]
nächster Essenswunsch: Eintopf (2 hungrige Studierende warten)
Reputation der Mensa: 0
> 1 (← Mitarbeiter/in versucht fälschlicherweise Vegi statt Eintopf auszugeben)
Vegi möchte ich nicht! Ich möchte Eintopf!
fertige Essen: [Vegi, Vegi, Eintopf, Eintopf, Eintopf]
nächster Essenswunsch: Currywurst (1 hungrige Studierende warten)
Reputation der Mensa: -1
> 1 (← Mitarbeiter/in versucht fälschlicherweise Vegi statt Currywurst auszugeben)
Vegi möchte ich nicht! Ich möchte Currywurst!
Fertig für heute. Die Mensa schließt.
Finale Reputation der Mensa: -2
```

a) Implementieren Sie die Funktion `length_list`, die die Anzahl der Elemente der Liste zurückgibt.

b) Implementieren Sie die Funktion `get_list`, die das Listenelement an der Indexposition zurückgibt. Das erste Listenelement befindet sich an Indexposition 0.

c) Implementieren Sie die Funktion `free_list`, die eine List mitsamt Inhalt freigibt. Beachten Sie, dass die Liste Besitzer („owner") der Listenelemente (`value`) ist und alle Listenelemente dynamisch allokiert sind. Diese müssen also auch freigegeben werden.

d) Implementieren Sie die Funktion `append_list`, die ein neues Listenelement hinten an die Liste anfügt.

e) Implementieren Sie die Funktion `Node* remove_list(Node* list, int index)`. Diese soll den Knoten an Position `index` löschen (und dessen Speicher freigeben) und die resultierende Liste zurückgeben. Wenn der Index ungültig ist, soll die Liste nicht verändert werden. Achten Sie darauf, dass keine Speicherlecks entstehen.

f) Implementieren Sie die Funktion `print_situation`, die den aktuellen Zustand der Simulation wie im obigen Beispiel ausgibt. Beispielausgabe:

```
fertige Essen: [Spaghetti, Vegi, Salat, Salat, Vegi]
nächster Essenswunsch: Currywurst (3 hungrige Studierende warten)
Reputation der Mensa: -1
```

g) Implementieren Sie die Funktion `finish`, die eine abschließende Meldung ausgibt, allen dynamisch allokierten Speicher freigibt und das Programm beendet. Achten Sie darauf, dass keine Speicherlecks auftreten. Beispielausgabe:

```
Fertig für heute. Die Mensa schließt.
Finale Reputation der Mensa: 3
```

h) Implementieren Sie die Funktion `run_mensa`, die es erlaubt, interaktiv gewünschte Essen auszugeben. Es gibt nur eine Essensausgabe vor der die Studierenden in einer Schlange warten und nacheinander bedient werden. Dazu wird der Index des gewünschten Gerichts in der Liste der fertiggestellten Essen eingegeben. Falls ein Essenswunsch nicht erfüllt werden kann, soll -1 eingegeben werden. Die Eingabe von -2 beendet das Programm. Implementieren Sie die Funktion entsprechend dem oben gezeigten Beispiel.

Hinweise:

- Lesen Sie eine ganze Zahl von der Standardeingabe mit `int i_input(void);`

Erzeugen Sie eine Zufallszahl mit `int i_rnd(int n); // 0, 1, …, n-1`

Aufgabe 3: Weihnachtsbaum

Das Template für diese Aufgabe ist xmas_tree.c. In dieser Aufgabe soll ein Weihnachtsbaum erstellt werden. Dieser besteht aus Zweigen die in einer Binärbaumstruktur verkettet sind. Sie sollen den Baum erstellen und schmücken. Die Strukturen für den Baum sind bereits vorgegeben. Sowie eine `print` Funktion, die den Baum ausgibt. TIPP: Sie können sich beliebige Hilfsfunktionen erstellen, die auch rekursiv definiert sein dürfen.

Die Nutzung von `xmalloc` statt `malloc` und `xcalloc` statt `calloc` ist verpflichtend, um die Überprüfung auf korrekte Freigabe des Speichers zu ermöglichen.

a) Implementieren Sie die Funktionen `new_ball()`, `new_LED_candle()` und `new_star()`, die die entsprechenden Strukturen initialisieren und einen Pointer zurückgeben. Suchen Sie sich eine Farbe Ihrer Wahl aus für das `color` Attribut aus. Für den `radius` von Kugeln und für die Anzahl der Zacken eines Sterns (`corner_count`), verwenden Sie Zufallszahlen im Bereich von 0 bis 8. LED-Kerzen sind zudem nach der Erstellung nicht an (`on = false`) und Kugeln sind immer glänzend (`shiny = true`).

b) Implementieren sie die Funktion `new_branch(Branch* left, Branch* right)`, die zwei Branches übergeben bekommt und einen neuen Branch erstellt, der diese beiden als Kinder `left` and `right` hat. Die Variable `dry` soll mit `false` initialisiert werden. Das Element, welches an einem `Branch` hängt soll zufällig entweder einen Stern, eine Kerze oder eine Kugel enthalten. Implementieren Sie die Funktion `new_xmas_tree(int height)`, die einen neuen Baum erstellt mit einer bestimmten Tiefe erstellt.

Tiefe 0	Tiefe 1	Tiefe 2
Tree | Null	Tree | Branch	Tree | Branch / \\ Branch Branch

c) Implementieren Sie die Funktion `free_tree(Tree* tree)`, die den durch einen Baum allokierten Speicher wieder freigibt. Nach dem Aufruf der Funktion sollte kein Speicher mehr allokiert sein.

d) Schreiben Sie eine Funktion, die eine Statistik für den erzeugten Baum ausgibt:

```
Tree statistic:
Stars: 17
Balls: 5
Candles: 9
```

Implementieren Sie sich dazu eine Hilfsfunktion, die einen Pointer auf einen Baum und ein Element vom Typ Tag übergeben bekommt. Die Funktion soll den Baum durchlaufen und für den gegeben Wert von Tag (`BALL`, `STAR` oder `LED_CANDLE`) die Anzahl an Elementen zählen und zurückgeben.

e) Implementieren Sie die Funktion `bool dry_out(Tree* tree)` rekursiv. Wenn das nicht möglich ist, erstellen Sie rekursive Hilfsmethoden. Die Funktion soll das Austrocknen eines Weihnachtsbaumes über einen Tag simulieren können. Jeder Branch hat ein `bool dry`. Dieses soll gesetzt werden, wenn der Zweig ausgetrocknet ist. Es ist möglich das ein Zweig ausgetrocknet ist, die nachfolgenden Zweige (Kinder) jedoch nicht. Dabei soll die Funktion „from bottom to top" arbeiten. Als erstes sollen die untersten Knoten (Blätter oder Knoten ohne Nachfolger) verarbeitet werden. Erst wenn dies geschehen ist, darf der Elternknoten verarbeitet werden. Die Funktion soll den booleschen Wert `dry` eines Zweiges unter folgenden Bedingungen auf `true` setzen:

   a. Wenn der Zweig keine Kinder mehr hat (`left == NULL` und `right == NULL`) soll der Zweig mit 25% Wahrscheinlichkeit an diesem Tag austrocknen.

   b. Wenn der Zweig zwei Kinder hat, so soll er mit 50% Wahrscheinlichkeit austrocken, wenn ein Kind bereits ausgetrocknet ist und mit 75% Wahrscheinlichkeit, wenn beide Kinder ausgetrocknet sind. Sind beide Kinder noch nicht ausgetrocknet so soll der Zweig nur mit 10% Wahrscheinlichkeit austrocknen.

   Wenn der Weihnachtsbaum komplett vertrocknet ist, so soll er entsorgt werden. Die Funktion `dry_out` soll in diesem Fall `true` zurückgeben. In allen anderen Fällen `false`. Ein leerer Baum oder Zweig soll immer als ausgetrocknet gelten. Wie viele Tage vergehen im Durchschnitt für einen gegebenen Baum mit einer Tiefe von 5-10 bis er ausgetrocknet ist? Nutzen Sie die Funktion `dry_out` und testen Sie für jede Höhe 100 zufällig erstellte Bäume. Geben Sie die Ergebnisse auf der Konsole aus.

f) Nehmen Sie an, Sie möchten den Baum neu schmücken. Also entsprechend alle Elemente abhängen und neue anbringen. Für welche Strukturen müssten Sie den Speicher freigeben und für welche nicht? Beschreiben Sie Ihr Vorgehen als Kommentar in die Template Datei. Ihre Beschreibung sollte so aussagekräftig sein, dass ein Kommilitone aus dieser eine Funktion implementieren kann.

g) (Optionale Teilaufgabe) Verschönern Sie die Ausgabe des Baumes. Geben Sie auch Bäumer anderer Tiefe korrekt aus. Geben Sie Bäume aus, die nicht symmetrisch sind.

Hinweise zum Editieren, Compilieren und Ausführen:

- mit Texteditor `file.c` editieren und speichern
- `make file` ← ausführbares Programm erstellen
- `./file` ← Programm starten (evtl. ohne `./`)
- Die letzten beiden Schritte lassen sich auf der Kommandozeile kombinieren zu: `make file && ./file`