Human-Computer Interaction Group

Leibniz Universität Hannover

Programmieren 1 – WS 2019/20

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 09.01. um 23:59 Uhr über https://assignments.hci.uni-hannover.de/WiSe2019/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 wolf_goat_cabbage.c. In dieser Aufgabe sollen verschiedene Funktionen auf einer einfach verketteten Liste mit Elementen vom Typ String implementiert werden. Diese Funktionen werden in Aufgabe 2 benötigt. Die Struktur für die Listenknoten ist vorgegeben. Die Benutzung der Listen und Listenfunktionen der Programmieren I Library ist für diese Aufgabe nicht erlaubt. 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 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.

b) 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 das 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.

c) Implementieren Sie mindestens drei Beispielaufrufe in der Funktion `length_list_test`. Verwendetn Sie `test_equal_i(actual, expected);`

d) Implementieren Sie die Funktion `int index_list(Node* list, String s)`. Diese soll den Index von `s` in `list` zurückgeben. Wenn `s` nicht in `list` vorkommt, soll -1 zurückgegeben werden. Implementieren Sie außerdem mindestens drei Beispielaufrufe in der zugehörigen Testfunktion.

e) Implementieren Sie die Funktion `Node* remove_list(Node* list, int index)`. Diese soll den Knoten an Position index löschen (und den Speicher freigeben) und die resultierende Liste zurückgeben. Wenn der Index ungültig ist, soll die Liste nicht verändert werden. Implementieren Sie außerdem mindestens drei Beispielaufrufe in der zugehörigen Testfunktion.

Aufgabe 2: Der Wolf, die Ziege und der Kohlkopf

Das Template für diese Aufgabe ist ebenfalls wolf_goat_cabbage.c. Entfernen Sie die Kommentare vor `make_puzzle` und `play_puzzle` (am Ende der `main`-Funktion). In dieser Aufgabe soll ein Spiel implementiert werden, in dem ein Bauer einen Wolf, eine Ziege und einen Kohlkopf in einem Boot über einen Fluss transportieren möchte. Zunächst sind Bauer, Wolf, Ziege, Kohlkopf und Boot am linken Ufer des Flusses. Leider hat das Boot nur einen freien Platz (neben dem Bauern, der das Boot rudert). Der Bauer darf aber den Wolf und die Ziege nicht alleine lassen, weil sonst der Wolf die Ziege frisst. Er darf auch die Ziege mit dem Kohlkopf nicht alleine lassen, weil sonst die Ziege den Kohlkopf frisst. Ist der Bauer in Reichweite, besteht keine Gefahr. Das Spiel ist dann erfolgreich gelöst, wenn Wolf, Ziege und Kohlkopf sicher am rechten Ufer angelangt sind.

a) Implementieren Sie die Funktion `print_puzzle`, die den aktuellen Spielzustand ausgibt. Der Anfangszustand soll z.B. mit folgender Zeile dargestellt werden:

```
[Wolf Ziege Kohl][]     []
```

Dabei sind linkes Ufer, Boot und rechtes Ufer durch Listen repräsentiert. Im Ausgangszustand sind alle Objekte am linken Ufer, das Boot liegt am linken Ufer und ist leer und das rechte Ufer ist ebenfalls leer. Wenn das Boot leer nach rechts fährt, ändert sich die Ausgabe in:

```
[Wolf Ziege Kohl]     [][]
```

b) Implementieren Sie die Funktion `finish_puzzle`, die allen dynamisch allokierten Speicher freigibt und das Programm beendet. Zum Beenden kann `exit(0);` verwendet werden.

c) Implementieren Sie die Funktion `evaluate_puzzle`, die aktuelle Situation analysiert und ausgibt. Die Funktion soll das Programm mit einer Meldung beenden, wenn die Aufgabe gelöst wurde bzw. die Situation kritisch ist.

d) Implementieren Sie die Funktion `play_puzzle`, die den Anfangszustand des Spiels ausgibt, Eingaben des Spielers entgegen nimmt und die Listen entsprechend manipuliert. Ist beispielsweise das Boot leer und auf der linken Seite, dann soll nach Eingabe von `Wolf` der Wolf vom linken Ufer genommen und in das Boot geladen werden. Die Eingabe `l` bewegt das Boot nach links, die Eingabe `r` nach rechts. Die Eingabe von `q` (für quit) soll das Spiel beenden. Nach jeder Eingabe soll diese Funktion die neue Situation evaluieren und ausgeben. Es folgt ein (nicht erfolgreicher) Beispielablauf:

[Wolf Ziege Kohl][]	[]	→ Anfangszustand, alle am linken Ufer
Wolf		← Eingabe: Spieler lädt den Wolf ins Boot
[Ziege Kohl][Wolf]	[]	→ Wolf im Boot
r		← Eingabe: r für nach rechts übersetzen
[Ziege Kohl]	[Wolf][]	→ Boot mit Wolf ist am rechten Ufer
Die Ziege frisst den Kohl.		→ Ziege ist am linken Ufer allein mit dem Kohl und frisst ihn

Beispielablauf: Hinüberbringen der Ziege:

[Wolf Ziege Kohl][]	[]	→ Anfangszustand, alle am linken Ufer
Ziege		← Eingabe: Spieler lädt die Ziege ins Boot
[Wolf Kohl][Ziege]	[]	→ Ziege im Boot
r		← Eingabe: r für nach rechts übersetzen
[Wolf Kohl]	[Ziege][]	→ Boot mit Ziege ist am rechten Ufer
Ziege		← Eingabe: Spieler lädt die Ziege aus dem Boot
[Wolf Kohl]	[][Ziege]	→ Ziege am rechten Ufer, Boot leer

Hinweise: Verwenden Sie `String s_input(100)`, um einen dynamisch allokierten String von der Standardeingabe einzulesen. Verwenden Sie `bool s_equals(String s, String t)`, um zu prüfen, ob zwei Strings gleich sind. Nutzen Sie die in Aufgabe 1 definierten Listenoperationen. Geben Sie dynamisch allokierten Speicher wieder frei. Sie dürfen, falls notwendig, beliebige Hilfsfunktionen implementieren.

Hinweis: Es gibt mehrere Varianten solcher Flussüberquerungsrätsel. Siehe: https://de.wikipedia.org/wiki/Flussüberquerungsrätsel

Aufgabe 3: Wunschliste

Das Template für diese Aufgabe ist `wish_list.c`. In dieser Aufgabe sollen Sie dem Weihnachtsmann helfen, indem Sie ihm die Entscheidung abnehmen, welches Kind welches Geschenk bekommt. Jedes Kind soll drei seiner Wünsche erfüllt bekommen. Kinder, die weniger als drei Wünsche haben, sollen zusätzliche Geschenke erhalten. Die Auswahl der zusätzlichen Geschenke soll anhand der Wunschhäufigkeit eines Geschenkes erfolgen. Die Kinder, die nur zwei Wünsche haben sollen zusätzlich das am häufigsten gewünschte Geschenk erhalten. Die Kinder, die nur einen Wunsch haben, sollen zusätzlich das häufigste und das zweithäufigste gewünschte Geschenk erhalten.

Kinder, die sich mehr als 3 Wünsche gewünscht haben, sollen auch nur 3 Geschenke erhalten. Die Auswahl dieser Geschenke soll so geschehen, dass die Kinder die 3 Geschenke aus ihrer Wunschliste erhalten, die insgesamt am seltensten gewünscht wurden.

Es gibt eine Textdatei `wishes.txt`. Diese enthält die Weihnachtswünsche verschiedener Kinder. Das Einlesen der Datei ist bereits implementiert.

a) Beschreiben Sie in einem Kommentar den Zusammenhang der verschiedenen Listenstrukturen. Ist es sinnvoll bei dieser Aufgabe auf Listen zu setzen oder sollten lieber Arrays verwendet werden? Begründen Sie. Beschreiben Sie in Kommentaren von maximal 3 Sätzen, was die folgenden Funktionen machen:

- `wish_list_length`
- `read_wishes`
- `read_wish_list`
- `swap`

b) Implementieren Sie die Funktion `Node* contains(Node* n, char* value)` rekursiv. Diese soll einen Zeiger auf eine `Node` übergeben bekommen und diese und alle folgenden Nodes darauf untersuchen, ob eine `Node` einen Wunsch enthält der gleich `value` ist. Nutzen Sie für den Vergleich von zwei Zeichenketten die C-Funktion `strcmp(const char* str1, const char* str2)`. Diese liefert 0 zurück, wenn `str1` und `str2` gleich sind. Wenn die Funktion eine entsprechende `Node` gefunden hat, soll sie einen Zeiger auf diese `Node` zurückgeben. Sollte keine `Node` gefunden werden, so soll `NULL` zurückgegeben werden.

c) Implementieren Sie die Funktion `Wish_List* create_wish_statistics(Lnode* lnode)`. Diese Funktion soll eine Statistik über alle Wünsche der Kinder erstellen. Dafür bekommt sie eine Liste übergeben, die wieder die Wunschlisten der Kinder enthält. Die Funktion soll einen Zeiger auf eine Wish_List zurückgeben. Die Struktur eignet sich dank ihrer Parameter auch für das Speichern der Häufigkeiten der einzelnen Funktionen. Nutzen Sie die gegebenen Funktionen aus dem Template wie `contains` und `new_node`. Vergleichen Sie die Häufigkeiten mit den im Quelltext angegebenen Häufigkeiten. Die Sortierung wird jedoch noch nicht stimmen.

d) Implementieren Sie die Funktion `bool is_sorted(Wish_List* list)` iterativ. Die Funktion soll `true` zurückgeben, wenn die Liste absteigend sortiert ist. Danach sollte die Wunschstatistik sortiert ausgegeben werden.

e) Implementieren Sie den Rest der Funktion `void three_wishes_per_child(Lnode* lnode, Wish_List* stats)`. Diese soll die in der Einleitung beschriebene Funktionalität implementieren, indem sie die zu langen Wunschlisten kürzt und die zu kurzen Wunschlisten auffüllt. Zu kurze Wunschlisten sollen mit den am Häufigsten gewünschten Geschenken aufgefüllt werden, die sich das Kind nicht wünscht. Aus zu langen Listen sollen die am häufigsten gewünschten Geschenke entfernt werden. Betrachten Sie dazu die **sortierte** Statistik, die ausgegeben wird . Überlegen Sie sich, wie diese Ihnen bei der Aufgabe hilft, bevor Sie anfangen zu implementieren. Nutzen Sie die Funktionen `contains`, `new_node` und `remove_node`.

f) Implementieren Sie die Funktionen `free_node`, `free_wish_list` und `free_lnode`. Diese sollen den Speicher der jeweiligen Strukturen freigeben. Am Ende sollen alle allokierten Speicherbereiche wieder freigegeben sein. Implementieren Sie `free_node` und `free_lnode` rekursiv.

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`