Human-Computer Interaction Group

Leibniz Universität Hannover

# Programmieren 1 – WS 2017/18

**Prof. Dr. Michael Rohs, Oliver Beren Kaul, M.Sc., 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. Plagiate führen zum Ausschluss von der Veranstaltung.

Abgabe bis Freitag den 12.1. um 23:59 Uhr über https://assignments.hci.uni-hannover.de/WiSe2017/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.

## 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: 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_candle()` und `new_star()`, die die entsprechenden Strukturen initialisieren und einen Pointer zurückgeben. Suchen Sie sich eine Farbe Ihrer Wahl aus für die `color` Attribute 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. Kerzen sind zudem nach der Erstellung nicht angezündet (`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<br>\|<br>Null | Tree<br>\|<br>Branch | Tree<br>\|<br>Branch<br>/ \\<br>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) Implementieren Sie die Funktion `get_count_of_branches_for_depth(Tree* tree, int depth)`, die die Knoten in einem Baum zählt, die in einer bestimmten Tiefe sind. Für das obige Beispiel eines Baumes mit Tiefe 2 gibt es 1 Knoten in der Tiefe 0, und 2 Knoten in der Tiefe 1. Ist der Baum leer oder ist der Wert von `depth` größer als die Baumtiefe soll 0 zurückgegeben werden. Die Testfälle in `test_depht()` sollten alle erfüllt werden.

e) Implementieren Sie die Funktion `light_candles(Tree* tree, bool on)`, die alle Kerzen auf dem Baum anzündet oder löscht, je nach übergebenen Wahrheitswert. Sie können sich nun den erzeugten Baum auf der Konsole ausgeben lassen und die Kerzen bei Eingabe von „c" anzünden oder löschen. Alle „i" sollten auf „I" wechseln.

f) (Optional) Verschönern Sie die Ausgabe des Baumes.

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`