Human-Computer Interaction Group

Leibniz Universität Hannover

Assignment 9

Programmieren 1 – WiSe 25/26

Prof. Dr. Michael Rohs, Falk Stock, M.Sc.

Alle Assignments (bis auf das erste) müssen in Zweiergruppen bearbeitet werden. Ein Gruppenmitglied kann dabei die Lösung der Zweiergruppe gebündelt abgeben. Einzige Ausnahme ist dabei das erste Assignment, wo jedes Gruppenmitglied einzeln abgeben muss. 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 Donnerstag, den 08.01. um 23:59 Uhr über https://assignments.hci.uni-hannover.de/WiSe2025/Prog1. Die Abgabe muss aus einer einzelnen ZIP-Datei bestehen, die den Quellcode, eine PDF für Freitextaufgaben und alle weiteren nötigen Dateien (z.B. Eingabedaten oder Makefiles) enthält. Lösen Sie Umlaute in Dateinamen auf.

Zum Bestehen der Studienleistung müssen Sie mindestens zwei von vier Punkten pro Assignment erzielen. Möchten Sie zusätzlich den Klausurbonus erreichen, müssen Sie über alle Assignments hinweg 75% der Punkte erreichen.

Hinweis: prog1lib

Die Dokumentation der prog1lib finden Sie unter der Adresse:
https://postfix.hci.uni-hannover.de/files/prog1lib/

Aufgabe 1: Matrizen (1 Punkt)

Das Template für diese Aufgabe ist **matrix.c**. In dieser Aufgabe geht es um dynamisch allokierte Matrizen. Eine Matrix wird durch einen Pointer auf eine struct Matrix repräsentiert. Diese Struktur enthält die Angabe der Anzahl von Zeilen und Spalten, sowie einen Pointer auf ein Array, das Pointer auf die Zeilen der Matrix enthält. Jede Zeile der Matrix ist ein weiteres Array mit Elementen vom Typ `double`. Verwenden Sie `xmalloc()` und `free()` zur dynamischen Anforderung bzw. Freigabe von Speicher. Der dynamisch angeforderte Speicher soll vor dem Ende des Programms auch wieder freigegeben werden.

a) Implementieren Sie die Funktion **make_matrix()**, die dynamisch eine Matrix mit der entsprechenden Anzahl Zeilen und Spalten erzeugt, die Elemente mit `0` initialisiert und einen Pointer auf die erzeugte Matrix zurückgibt.

b) Implementieren Sie die Funktion **copy_matrix()**. Diese Funktion bekommt einen Pointer auf ein eindimensionales Array mit `n_rows * n_cols` `double`-Werten übergeben. Es soll nun eine Matrix dynamisch erzeugt werden und die übergebenen `double`-Werte in diese Matrix kopiert werden. Die Werte sind in der Eingabe zeilenweise angeordnet.

c) Implementieren Sie die Funktion **print_matrix()**, die eine Matrix sinnvoll formatiert ausgibt.

d) Implementieren Sie die Funktion **add_matrices()**, die zwei Matrizen addiert. Die Eingabematrizen dürfen dabei nicht verändert werden, sondern das Ergebnis soll als neue Matrix dynamisch erzeugt

und zurückgegeben werden. Die Funktion soll auch überprüfen, ob die Dimensionen der Argumente kompatibel sind. Geben Sie im Fehlerfall `NULL` zurück.

e) Implementieren Sie die Funktion **free_matrix()**, die eine dynamisch allokierte Matrix freigibt. Wenn allokierter Speicher nicht wieder freigegeben wird, erscheint beim Beenden des Programms eine ähnliche Fehlermeldung wie die folgende:

```
4 bytes allocated in make_matrix (matrix.c at line 123) not freed
```

Aufgabe 2: Dynamisch wachsender Stack (1 Punkt)

In dieser Aufgabe geht es darum, einen dynamisch wachsenden Stack zu implementieren. Der Stack soll die bekannten Funktionen `push` und `pop` bieten und Integer-Werte abspeichern. Im Gegensatz zur vorherigen Woche soll dieser Stack eine beliebig große Menge an Werten speichern können. Der Speicher für die Integer-Werte des Stack soll also **dynamisch** allokiert werden.

Die Template-Datei für diese Aufgabe ist **dynamic_stack.c**. Bearbeiten Sie die mit **todo** markierten Stellen.

a) Vervollständigen Sie die Definition des `structs` **DynamicStack**. Überlegen Sie sich, welche Werte Sie zum Verwalten des Stacks und dessen Speicher benötigen.

b) Implementieren Sie die Funktion **stack_new()**, die einen neuen `DynamicStack` erzeugt. Beachten Sie, dass diese Funktion bereits einen Pointer zurückgibt. Sie müssen hier also den Speicher für das `struct` bereits dynamisch allokieren. Anfangs sollen noch keine Werte in dem `DynamicStack` liegen.

c) Implementieren Sie die Funktion **stack_push()**, die den gegebenen Wert oben auf den Stack legt. Reicht der reservierte Speicher nicht aus, so erweitern Sie den Speicher des `DynamicStack`, um genügend Platz für den neuen Wert zu schaffen.

d) Implementieren Sie die Funktion **stack_pop()**, die das oberste Element vom Stack nimmt und zurückgibt. Geben Sie hier den Speicher nicht frei, sondern halten Sie diesen für neue Werte reserviert. Ist der Stack leer, so soll diese Funktion `0` zurückgeben.

e) Implementieren Sie die Funktion **stack_free()**, die den dynamischen Stack löscht und allen verwendeten Speicher freigibt.

Aufgabe 3: Operationen auf Listen (1 Punkt)

Die Template-Datei 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 4 benötigt. Die Struktur für die Listenknoten ist vorgegeben.

a) Implementieren Sie die Funktion **free_list()**, die eine Liste 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 **test_equal_lists()**, die überprüft, ob zwei Listen inhaltlich gleich sind, also ob sie 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. Verwenden Sie zum Vergleichen von Zeichenketten `s_equals(s, t)` oder `strcmp(s, t) == 0`.

c) Implementieren Sie mindestens drei Beispielaufrufe in der Funktion **length_list_test()**. Verwenden 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. Sichern Sie die Funktion mit einer Precondition ab, falls der `String s == NULL` ist. Implementieren Sie außerdem mindestens drei Beispielaufrufe in der zugehörigen Testfunktion **index_list_test()**.

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. Sichern Sie die Funktion mit entsprechenden Preconditions ab. Nutzen Sie das Makro `ensure_code`, um eine Postcondition zu formulieren (z.B. `neue Listenlänge == alte Listenlänge -1`). Implementieren Sie außerdem mindestens drei Beispielaufrufe in der zugehörigen Testfunktion **remove_list_test()**.

Aufgabe 4: Der Wolf, die Ziege und der Kohlkopf (1 Punkt)

Das Template für diese Aufgabe ist ebenfalls **wolf_goat_cabbage.c**. Entfernen Sie die Kommentarzeichen 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 muss. Zunächst sind Bauer, Wolf, Ziege, Kohlkopf und Boot am linken Ufer des Flusses. Leider hat das Boot (wenn der Bauer im Boot ist) nur einen freien Platz. 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 am gleichen Ufer, besteht keine Gefahr. Das Spiel ist dann erfolgreich gelöst, wenn Wolf, Ziege und Kohlkopf sicher am rechten Ufer angelangt sind.

Die Funktion `print_puzzle()` gibt den aktuellen Spielzustand aus. Der Anfangszustand wird z.B. mit folgender Zeile dargestellt:

```
[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]        [][]
```

a) Implementieren Sie die Funktion **evaluate_puzzle()**, die die aktuelle Situation analysiert und ausgibt, ob die Aufgabe gelöst wurde bzw. kritisch ist. Die Funktion darf das Programm auch abbrechen (`finish_puzzle(Puzzle* p)`), falls das Spiel verloren wurde.

b) Implementieren Sie die Funktion **play_puzzle()**, die das Spiel ausgibt, Eingaben des Spielers entgegen nimmt und die Listen entsprechend manipuliert. Ist beispielsweise das Boot auf der linken Seite und die Ziege im Boot, dann soll nach Eingabe von `goat` bzw. `g` die Ziege aus dem Boot genommen und auf die linke Seite gesetzt werden. Die Eingabe `l` bewegt das Boot nach links, die Eingabe `r` nach rechts. Die Eingabe von `q` soll das Spiel beenden. Nach jeder Eingabe soll `evaluate_situation()` aufgerufen werden.

c) Stellen Sie sicher, dass dynamisch allokierter Speicher über `free()` wieder freigegeben wird.

Ein möglicher Spielablauf könnte wie folgt aussehen:

```
[wolf, goat, cabbage][]      []
> g
[wolf, cabbage][goat]      []
> r
[wolf, cabbage]      [goat][]
> goat
[wolf, cabbage]      [][goat]
> l
[wolf, cabbage][]      [goat]
> wo
[cabbage][wolf]      [goat]
> r
[cabbage]      [wolf][goat]
> w
[cabbage]      [][goat, wolf]
> l
[cabbage][]      [goat, wolf]
wolf eats goat
```

Hinweise: Verwenden Sie `String s_input(100)`, um einen dynamisch allokierten String von der Standardeingabe einzulesen. Verwenden Sie `strcmp(s, t) == 0`, um zu prüfen, ob zwei Zeichenketten gleich sind. Nutzen Sie die in Aufgabe 3 definierten Listenoperationen. Sie dürfen, falls notwendig, beliebige Hilfsfunktionen implementieren.