Human-Computer Interaction Group

Leibniz Universität Hannover

Programmieren 1 – WS 2022/23

Prof. Dr. Michael Rohs, Jan Feuchter, 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 Donnerstag den 22.12. um 23:59 Uhr über https://assignments.hci.uni-hannover.de/WiSe2022/Prog1. Die Abgabe muss aus einer einzelnen Zip-Datei bestehen, die den Quellcode, ein PDF für 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://postfix.hci.uni-hannover.de/files/prog1lib/

Aufgabe 1: Operationen auf Listen

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 2 benötigt. Die Struktur für die Listenknoten ist vorgegeben. Die Benutzung der Listen und Listenfunktionen der prog1lib 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. 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`. 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. Sichern Sie die Funktion mit einer Precondition ab, falls der String `s == NULL` ist. 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. 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.

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

b) Implementieren Sie die Funktion `play_puzzle`, die des Spiel ausgibt, Eingaben des Spielers entgegennimmt und die Listen entsprechend manipuliert. Ist beispielsweise das Boot leer und auf der linken Seite, dann soll nach Tastatureingabe von `Wolf<return>` der Wolf vom linken Ufer genommen und in das Boot geladen werden. Die Eingaben `l<return>` und `r<return>` sollen das Boot nach links bzw. rechts bewegen. Die Eingabe von `q<return>` (für quit) soll das Spiel beenden. Nach jeder Eingabe soll die neue Situation evaluiert und ausgegeben werden. 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 am linken Ufer allein mit 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 `s_equals(s, t)` oder `strcmp(s, t) == 0`, um zu prüfen, ob zwei Zeichenketten 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: Dateisystem

In dieser Aufgabe sollen Sie die Struktur eines Dateisystems implementieren. Ein Dateisystem ist ein Baum, dessen Knoten entweder Ordner (NodeType NT_DIR) oder Dateien (NodeType NT_FILE) sind. Jeder Knoten besitzt einen Namen mit maximal 63 Zeichen. Ordner enthalten eine Liste mit Einträgen. Dateien besitzen einen Zeiger auf ihren Inhalt und speichern ihre Länge in Bytes. Die Verzeichniseinträge eines Ordners sind stets aufsteigend lexikographisch sortiert. Als Wurzelknoten für ein Dateisystem dient ein Ordner mit leerem Namen. Andere Namen dürfen nicht leer sein. Das Slash (/) dient als Trennsymbol für Namen auf einem Pfad. Mehrere aufeinander folgende Slashes sollen als ein Slash betrachtet werden (z.B. /// wird zu /).

Die Template-Datei für diese Aufgabe ist `filesystem.c`.

a) Machen Sie sich mit der gegebenen Struktur `Node` vertraut und implementieren Sie die Konstrukturfunktionen `new_file` und `new_directory`. Nutzen Sie die Funktion `new_node`.

b) Implementieren Sie die Funktion `insert_into_directory`, die einen Knoten in einen Ordner einfügt. Erzeugen Sie hier keine Kopie des Knotens, sondern fügen Sie den Knoten in das Dateisystem ein. Beachten Sie, dass die Position des neuen Knotens in dem Array so gewählt werden muss, dass das Array alphabetisch sortiert bleibt. Nutzen Sie dafür die `strcmp` Funktion. (Zum sortierte Einfügen, siehe Folie: Inserting an Element in Order)

c) Implementieren Sie die Funktion `free_node`, die den allokierten Speicher eines Knotens und gegebenenfalls aller enthaltener Knoten freigibt.

d) Vervollständigen Sie die Implementierung der Funktion `find_node`, die mit einer Pfadangabe den entsprechenden Knoten zurückgibt. Dabei sollen sowohl Dateien (z.B. "/system/4_processes.txt") und Ordner (z.B. "/home/user" und "/home/user/") zurückgegeben werden können. Existiert der Pfad nicht, soll NULL zurückgegeben werden. Sie können davon ausgehen, dass sie nur korrekt formatierte Eingaben behandeln müssen. Erzeugen Sie auch hier keine Kopie des Knotens, sondern geben Sie einen Zeiger auf den Knoten zurück. Implementieren Sie `find_node` rekursiv. Nutzen Sie die Funktionen `strcmp`, `find_entry` und `find_entry_n`.

e) Entfernen Sie den Kommentar vor `report_memory_leaks(true)` am Beginn der main Funktion. Stellen Sie sicher, dass der komplette Speicher wieder korrekt freigegeben wird.