Human-Computer Interaction Group

Leibniz Universität Hannover

Programmieren 1 – WS 2018/19

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

Übungsblatt 12

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 24.01. 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: Spielecharaktere und Objektlisten

Die für diese Aufgabe relevanten Dateien sind object.h, character.h, fighter.{h|c}, wizard.{h|c}, object_list.{h|c} und characters_test.c. Änderungen sind nur in fighter.c, wizard.c und characters_test.c nötig. Die Struktur `Character` soll einen Charaktertyp in einem Computerspiel repräsentieren, für den eine Angriffsstärke (attack) und eine Abwehrstärke (defense) berechnet werden kann. Die Strukturen `Fighter` und `Wizard` repräsentieren Spielecharaktere des jeweiligen Typs.

a) Machen Sie sich mit dem vorhandenen Quelltext vertraut. Die Definition von `Class` in object.h wurde in der Vorlesung besprochen. Die Struktur `CharacterClass` erweitert `Class` um Zeiger auf Funktionen vom Typ `AttackFun` und `DefenseFun`. Funktionen dieses Typs sollen der Berechnung von Angriffs- und Abwehrstärke eines Charakters dienen. Implementieren Sie `attack_fighter` und `defense_fighter` sowie `attack_wizard` und `defense_wizard` wie folgt:

$$attack_{Fighter} = 0.9 * sword * strength$$
$$defense_{Fighter} = 0.8$$
$$attack_{Wizard} = 1.0$$
$$defense_{Wizard} = 0.5 * potion * will^2$$

Überprüfen Sie Ihre Implementierung mit den (zu ergänzenden) Beispielaufrufen in characters_test.c.

b) Erstellen Sie in characters_test.c eine Objektliste mit den Objekten `f1`, `f2`, `w1` und `w2`. Diese Liste stellt eine Gruppe bestehend aus diesen vier Charakteren dar. Berechnen Sie mit einer `for`-Schleife über die Liste Gesamtangriffs- und Gesamtabwehrstärke der Gruppe. Geben Sie das Ergebnis aus.

c) Berechnen Sie nun in characters_test.c die Gesamtangriffsstärke der Objekte in der Liste mit Hilfe der `reduce_list`-Funktion und Ihrer Implementation von `add_attack` und geben Sie das Ergebnis aus.

d) Filtern Sie die Liste, so dass eine Ergebnisliste nur der `Wizards` gebildet wird. Verwenden Sie dazu die Funktion `filter_list` und definieren Sie eine Testfunktion `is_wizard`. Geben Sie die Ergebnisliste aus.

Aufgabe 2: Binärbäume

Die für diese Aufgabe relevanten Dateien sind integer_tree.{h|c}, integer_list.{h|c} und integer_tree_test.c. In dieser Aufgabe werden Operationen für einen Binärbaum implementiert. Der Binärbaum speichert ganze Zahlen.

a) In der `main`-Funktion von integer_tree_test.c findet sich das int-Array `values`. Fügen Sie die Elemente des Arrays geordnet in einen Binärbaum ein. Geben Sie die Elemente des Baums aus, so dass sie in aufsteigend sortierter Reihenfolge erscheinen.

b) Implementieren Sie die Funktion `free_tree` in integer_tree.c, die den Speicher des Baumes freigibt. Beachten Sie, dass der Baum nur ganze Zahlen speichert.

c) Die `print_tree`-Funktion in integer_tree.c gibt Bäume so aus, dass der Wert eines Knotens in der Mitte und die linken und rechten Unterbäume links und rechts davon erscheinen. Um die Struktur sichtbar zu machen wird jeder Knoten geklammert. Die Form ist also: (<left>, <value>, <right>). Der leere Baum wird als Unterstrich dargestellt. Beispiel:

```
(((_, 11, _), 1, (_, 12, _)), 0, (_, 2, (_, 7, _)))
```

Erweitern Sie die `print_tree`-Funktion so, dass Blätter in abgekürzter Form dargestellt werden. Für das Beispiel:

```
((11, 1, 12), 0, (_, 2, 7))
```

d) Implementieren Sie die Funktion get_odd_even_rec die einen Baum und zwei Zeiger auf Ergebnislisten übergeben bekommt. Die Ergebnisliste `odd` soll später alle ungeraden Elemente enthalten. Die Ergebnisliste `even` nur die geraden Elemente. Beantworten Sie zusätzlich die folgende Frage als Kommentar im Quellcode: Warum wird als Typ für die Ergebnisliste ein Zeiger auf einen Zeiger auf eine Node (bspw.: `Node ** odd`) verwendet und nicht ein Zeiger auf eine Node (`Node * odd`)?

e) Implementieren Sie nun das Filtern von geraden oder ungeraden Zahlen als Filterfunktion. Verwenden Sie die Funktion `filter_tree` und implementieren Sie eine Filterfunktionen `odd_or_even`. Die Funktion soll entweder gerade oder ungerade Elemente filtern. Nutzen Sie den Parameter x um dies zu ermöglichen.

f) Implementieren Sie die Funktion `is_sorted`. Diese gibt genau dann wahr zurück, wenn der Baum sortiert ist. Ein Baum ist sortiert, wenn er leer ist, nur aus dem Wurzelknoten besteht oder wenn für alle Nachfolgeknoten links eines Elternknoten gilt, dass Sie kleiner sind als der Elternknoten und für alle Nachfolgeknoten rechts eines Elternknoten gilt, dass Sie größer sind als der Elternknoten. Schauen Sie sich die Testfälle an. Sie können für Ihre Implementation beliebige Hilfsfunktionen erstellen und auch den Option-Type `Option_int` nutzen. Dieser ist entweder nicht initialisiert, wenn `valid` den Wert `false` enthält oder initialisiert, dann enthält `valid` den Wert `true` und in `value` ist ein gültiger Wert gespeichert.

Tipp: Es kann hilfreich sein, wenn Sie sich einen Baum aufmalen und prüfen, welche Grenzen für jeden Knoten gelten. Bedenken Sie, dass die Grenzen für Wurzelknoten nicht initialisiert sind, da er kein Nachfolgerknoten ist.

Aufgabe 3: Table of Contents

In dieser Aufgabe erstellen Sie eine Datenstruktur, die ein Dokument wie eine Bachelorarbeit speichern kann. Zur Vereinfachung speichert die Struktur nur die Kapitelnamen, sowie die Anzahl der Seiten für jedes Kapitel. Schauen Sie sich TOC.c und TOC.h an. Nutzen Sie für das Allokieren von Speicher ausschließlich xmalloc bzw. xcalloc.

a) Implementieren Sie die Konstruktorfunktion `new_TNode`, die den Titel eines Kapitels, die Anzahl der Seiten dieses Kapitels sowie einen Zeiger auf eine Liste von Unterkapiteln übergeben bekommt. Als Rückgabe liefert Sie einen Pointer auf eine neue TNode. Implementieren Sie zusätzlich die Funktion `new_node`, die eine neue Node erstellt. `new_node` bekommt einen Verweis auf eine TNode übergeben, sowie einen Pointer auf eine nachfolgende Node.

b) Implementieren Sie die Funktionen `free_TNode` und `free_Nodes`. Die erste Funktion gibt den von einem Kapitel allokierten Speicher frei (einschließlich der Unterkapitel). Die zweite Funktion gibt den Speicher der übergebenen Node sowie aller nachfolgenden Nodes und deren Kapitel frei. Am Ende soll der komplette vom Baum allokierte Speicher freigegeben werden.

c) Berechnen Sie die Anzahl der Seiten, die ein Kapitel benötigt. Implementieren Sie dafür die Funktion `calculate_pages`, die eine TNode übergeben bekommt und die Summe der Seiten zurückgibt. Gehen Sie davon aus, dass jedes Kapitel (auch Unterkapitel) auf einer neuen Seite beginnt. Die Anzahl der Seiten für ein Kapitel ist die Summe aus den Seiten des Kapitels plus die Anzahl der Seiten der Unterkapitel. Bsp. Das Kapitel Introduction hat eine eigene Seite plus die Seiten aus Motivation, Research Questions und Goals. Geben Sie die Anzahl der Seiten der Bachelorarbeit auf der Konsole aus.

d) Implementieren Sie die Funktion `print_TOC`, die das Inhaltsverzeichnis gut formatiert auf der Konsole ausgibt. Dazu gehören, die Hierarchie, der Titel, sowie die Seitenzahl. Vgl. Sie folgendes Bild:

```
The thesis has: 131 pages.
My Bachelorthesis
1 Introduction ..........................  0
1.1 Motivation ..........................  1
1.2 Research Questions ..................  3
1.3 Goals ................................  4
2 Basics ................................  7
2.1 Mathematical Basics ..................  8
2.2 Related Work ......................... 11
2.3 Interaction Techniques ............... 21
3 Concept ............................... 26
3.1 Old System ............................ 27
3.1.1 Structure ........................ 31
3.1.2 Functions ........................ 33
3.2 New System ............................ 35
3.2.1 Functions ........................ 41
3.2.2 Structure ........................ 43
3.3 Changes ............................... 45
3.3.1 Hardware Changes ................. 48
3.3.2 Software Changes ................. 50
4 Prototype ............................. 52
4.1 Hardware ............................... 53
4.2 Software ............................... 65
5 Studies ................................ 73
5.1 Study One ............................. 74
5.1.1 Participants ..................... 75
5.1.2 Setup ........................... 76
5.1.3 Procedure ........................ 79
5.2 Study Two ............................. 83
5.2.1 Setup ........................... 84
5.2.2 Procedure ........................ 86
5.2.3 Participants ..................... 88
6 Evaluation ............................. 90
6.1 Methods ............................... 91
6.2 Results ............................... 95
6.3 User Group ............................ 97
7 Conclusion ........................... 101
8 Literature ........................... 103
9 Apendix ............................. 106
```

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`