Human-Computer Interaction Group

Leibniz Universität Hannover

Programmieren 1 – WS 2020/21

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

Übungsblatt 8

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 10.12. um 23:59 Uhr über https://assignments.hci.uni-hannover.de/WiSe2020/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: Memory

In dieser Aufgabe geht es darum, das Spiel „Memory" zu implementieren, so dass man als Einzelspieler auf der Konsole spielen kann. Es geht darum, Paare gleicher, verdeckt liegender Karten durch Aufdecken von jeweils zwei Karten zu finden. Eine detailliertere Spielbeschreibung findet sich z.B. unter https://de.wikipedia.org/wiki/Memory_(Spiel). In der zu implementierenden Variante soll nur ein einzelner Spieler spielen können. Die Versuche und Punkte sollen vom Computer gezählt werden. Die zugehörige Template-Datei ist `memory_game.c`. Hier eine Beispielausgabe:

```
  1 2 3
1 # # #
2 # # #                    ← Anfangszustand mit sechs verdeckten Karten
0 points, 0 turns
> 11                       ← Eingabe: Spieler möchte Zeile 1, Spalte 1 aufdecken

  1 2 3
1 B # #                    ← Karte in Zeile 1, Spalte 1 aufgedeckt
2 # # #
0 points, 1 turns
> 12                       ← Eingabe: Spieler möchte Zeile 1, Spalte 2 aufdecken

  1 2 3
1 B C #                    ← zwei aufgedeckte Karten
2 # # #
> 0 points, 2 turns
press <return> to continue ← warten auf Eingabe, da B≠C wird wieder verdeckt
```

a) Die Funktion `shuffle` mischt die Karten (Einträge des Arrays a der Länge n) zufällig. Erläutern Sie die Arbeitsweise von `shuffle`. Angenommen die Funktion `uniform` liefere gleichverteilt zufällige ganze Zahlen im Intervall [0, i]. Sind dann alle möglichen Permutationen des Arrays gleich wahrscheinlich?

b) Implementieren Sie die Funktion `init_cards`. Nehmen Sie an, dass das übergebene Array bereits existiert und dass lediglich die Zeichen im Array `cards` so gesetzt werden müssen, dass jedes verwendete Symbol genau zweimal vorkommt und dass die Anordnung zufällig ist. Nicht verwendete Plätze auf dem Spielfeld sollen mit Leerzeichen aufgefüllt werden. Als Symbole sollen die ASCII-Zeichen ab `'A'` verwendet werden. Aufgedeckte Karten werden über ihren den ASCII-Wert (z.B. `'A'` = 65), verdeckte Karten über den negativen ASCII-Wert des Symbols (z.B. `-'A'` = -65) repräsentiert. Füllen Sie das Array beginnend mit Index 0 mit `-'A'`, `-'A'`, `-'B'`, usw. Nutzen Sie danach die gegebene `shuffle` Funktion.

c) Implementieren Sie die Funktion `print_board`, die den aktuellen Zustand des Spielfelds ausgibt. Die Ausgaben sollen aussehen, wie in der obigen Beispielausgabe und wie in den Kommentaren im Quelltext beschrieben. Die Zeichen sind im `cards`-Array zeilenweise von oben nach unten abgelegt.

d) Implementieren Sie die Funktion `int array_index(Board *b, int r, int c)`, die den zu den Koordinaten (row, column) gehörigen Array-Index in `cards` berechnet. Die Zeichen sind im `cards`-Array zeilenweise von oben nach unten abgelegt. Die Koordinaten (0,0) entsprechen also Index 0, die Koordinaten (0,1) dem Index 1 und die Koordinaten (1,0) dem Index `b->cols`. Das Programm muss mit `exit(1);` beendet werden, wenn ungültige Spalten- oder Zeilenwerte übergeben wurden.

e) Implementieren Sie unter Verwendung von `array_index` die Funktionen `get`, `set` und `turn` zum Lesen, Setzen bzw. Umdrehen einer Karte an der Position (row, column).

f) Implementieren Sie die Funktion `int clamp(int x, int low, int high)`, die den Wert von x auf das Intervall [low, high] beschränkt.

g) Erklären Sie den Aufbau der Funktion `do_move`. Ist die Funktion robust gegenüber Fehleingaben oder kann das Programm durch bestimmte Eingaben in einen inkonsistenten Zustand gebracht werden? Begründen Sie Ihre Antwort.

Stellen Sie sicher, dass Ihre Lösungen die in der Testfunktion `tests` aufgeführten Testfälle erfüllen.

Aufgabe 2: Sortieren durch zufälliges Vertauschen

Implementieren Sie aufbauend auf dem Template `random_sort.c` einen Sortieralgorithmus, der durch zufälliges Tauschen von zwei Elementen eines Arrays dieses sortiert. Es sollen Autos sortiert werden, die von einer Konstruktorfunktion erzeugt werden.

a) Schreiben Sie eine Funktion `int compare(Car car1, Car car2)`, die zwei Autos vergleicht und 1 zurückgibt, wenn car1 jünger ist als car2 und -1, wenn car1 älter ist als car2. Wenn beide Autos gleich alt sind, dann soll die Funktion auch die Marke überprüfen und diese lexikographisch sortieren. Sind zwei Autos gleich alt und haben die gleiche Marke, so soll 0 zurückgegeben werden. Nutzen Sie für den Vergleich von zwei

Zeichenketten die Funktion `int strcmp(char* str1, char* str2)`. Diese liefert als Rückgabe einen ganzzahligen Wert der kleiner als 0 ist, gleich 0 ist, oder größer als 0 ist, abhängig davon ob `str1` lexikographisch kleiner, gleich oder größer ist als `str2`.

b) Schreiben Sie eine Testfunktion `void compare_test(void)`, mit mindestens 5 Testfällen, die die korrekte Funktion Ihrer `compare` Funktion überprüft.

c) Implementieren Sie eine Funktion `bool sorted(Car* a, int length)`, die testet, ob das Array sortiert ist. Nutzen Sie die bereits implementierte `compare` Funktion. Die Funktion soll `true` zurückgeben, wenn das Array sortiert ist, ansonsten `false`.

d) Implementieren Sie die Funktion `int random_sort(Car* a, int length)`, die solange zwei zufällige Autos in dem Array vertauscht, bis die Liste sortiert ist. Nutzen Sie die Funktion `sorted` nach jedem Tausch, um zu testen ob, das Array nun sortiert ist. Die Funktion soll zunächst nur 0 zurückgeben, später (e) soll die Anzahl der Vertauschungen zurückgegeben werden.

e) Überprüfen Sie, wie oft Ihre Funktion Elemente vertauscht. Je öfter dies passiert, desto ineffizienter ist die Funktion. Fügen Sie dazu eine Zählvariable `swaps` in Ihre `random_sort` Funktion ein, die jedes Mal inkrementiert wird, wenn ein Tausch stattfindet. Nach der Sortierung soll `swaps` zurückgeben werden. Können Sie über diese Variable die Anzahl der Aufrufe der `compare` Funktion bestimmen? Wenn ja, wie hängt diese von `swaps` ab?

f) Testen Sie Ihren Algorithmus für mindestens 5 verschieden große zufällige Arrays jeweils 100-mal (Arraygrößen im Bereich von 3 – 10). Geben Sie jeweils den Durchschnittswert von `swaps` für die verschiedenen Arraylängen auf der Konsole aus, sowie die Anzahl der stattgefundenen Aufrufe von `compare` (falls möglich). Ein Array mit beliebiger Länge und zufälligen Autos können Sie mit der `create_car_park(int car_count)` Methode erstellen.

Aufgabe 3: Meine erste eigene Postfix REPL

In dieser Aufgabe geht es darum einen kleinen Teil eines Postfix Laufzeitsystems zu implementieren. Dieser soll Ausdrücke wie „1 2 + 3 –“ evaluieren können. Ihre Postfix REPL soll am Ende dieser Aufgabe die 4 Grundrechenarten + - * und / mit Ganzzahlen umsetzen können. Das Template für diese Aufgabe ist `postfix_repl.c`. Die zu verwendenden Strukturen sind schon gegeben. Ein StackElement ist dabei von dreierlei Art: Es kann leer sein `tag == EMPTY`, einen Wert enthalten `tag == VALUE` oder eine Operation enthalten `tag == OPERATION`. Die Struktur `Stack` speichert ein Array vom Typ `StackElement` sowie einen Verweis auf den Index des zuletzt hinzugefügten Elements.

a) Implementieren Sie folgende Funktionen: `StackElement make_stack_element(Tag tag)`, `StackElement make_value(int value)` und `StackElement make_operation(char operation)`. Diese sollen die verschiedenen Typen von `StackElement` initialisieren und zurückgeben.

b) Implementieren Sie die Funktion `void print_stack_element(StackElement ele)`, die ein `StackElement` sinnvoll auf der Konsole ausgibt. Bspw. [109] für die Zahl 109 oder [+], wenn der Operator + ist oder [ ] für EMPTY.

c) Implementieren Sie die Funktion `void clear_stack(Stack* stack)`, die den Stack mit EMPTY Elementen initialisiert. Die Größe des Stacks ist in der Struktur stack

gespeichert. Der Stackpointer soll auf -1 gesetzt werden, da noch kein Element dem Stack hinzugefügt wurde. Erstellen Sie sinnvolle Pre- und Postconditions. Nutzen Sie für die Postcondition das Macro forall (in der Prog1lib Dokumentation) oder das Macro forall_i (im Template) um in der Postcondition sicherzustellen, dass alle Elemente richtig initialisiert sind.

d) Implementieren Sie die Funktion `void print_stack(Stack* stack, int n)`, die einen Stack auf der Konsole ausgibt. Der Parameter n soll dabei angeben, wie viele Elemente ausgegeben werden. Bspw. sollte Folgendes nach dem Befüllen des Stacks mit den Werten 1, 2 und 3 ausgegeben werden:

   [3]<- auf dieses Element zeigt der Stackpointer

   [2]

   [1] <- unterstes Element

   Erstellen Sie auch hier sinnvolle Preconditions.

e) Schreiben Sie als Kommentar, was die Methoden `pop` und `push` leisten. Welche Parameter der Funktionen sind Eingabe- und welche Ausgabeparameter? Die Funktionen push und pop gehen unterschiedlich mit problematischen Situationen um. Sollten Ihrer Meinung nach lieber Preconditions genutzt werden, um ungültige Zustände des Stacks zu verhindern oder sollten diese Zustände mit if/else Verzweigungen abgefangen werden?

f) Implementieren Sie die Funktion `void compute(Stack* stack)`, die überprüft, ob oben auf dem Stack ein Element vom Typ `OPERATION` liegt und dieses ausführt. Bspw.:

   [+]<- auf dieses Element zeigt der Stackpointer

   [2]

   [1] <- unterstes Element

   Sollte zu nachfolgendem evaluieren.

   [3] <- unterstes Element, Stackpointer zeigt auf dieses Element.

   Nutzen Sie für Ihre Implementation die gegebenen Funktionen push und pop. Sichern Sie die Funktion mit sinnvollen Preconditions: Ein valider Stack (!=NULL), eine ausreichende Stackhöhe, etc. Erstellen Sie eine sinnvolle Postcondition.

g) Implementieren Sie in der main() Methode das Einlesen und Auswerten der Eingabe. Sie können die Hilfsfunktionen `is_operation(char c)` und `is_digit(char c)` nutzen. Erstellen Sie beim Einlesen einer ganzen Zahl ein neues `StackElement` und fügen Sie es dem gegebenen Stack hinzu. Beim Einlesen einer Operation soll diese als erstes auf den Stack gelegt werden und dann mithilfe der `compute` Funktion ausgewertet werden. Nach jeder Änderung des Stacks soll dieser auf der Konsole ausgeben werden. Hinweis: Sie müssen keine Fehlerbehandlung implementieren. Gehen Sie davon aus, dass nur ganze Zahlen oder die vier Operatoren getrennt von Leerzeichen eingegeben werden können.

h) OPTIONAL: Erweitern Sie Ihre REPL um beliebige Features. Bspw. Support von Doubles.