**Institut für Informatik**
**Albert-Ludwigs-Universität Freiburg**
**Prof. Dr. Peter Thiemann**
**Dr. Philipp Klaus Krause**

Sommersemester 2019

**6. Programmieren in C**

Abgabe bis 18. Juni, 10:15 GMT+2.

**Aufgabe 0:**

Implementiere eine Warteschlange von `int` in `fifo.c` und `fifo.h`!

Das Interface besteht aus dem Typen `fifo_t` und den Funktionen

```
fifo_t *fifo_init(size_t n)                // Erstellt eine neue Warteschlange,
                                           // die bis zu n Einträge verwalten kann.
                                           // Gibt im Fehlerfall 0 zurück.
int fifo_enqueue(fifo_t *fifo, int i);     // Fügt Eintrag ein.
                                           // Gibt im Fehlerfall 1 zurück, sonst 0.
int fifo_dequeue(fifo_t *fifo);            // Entfernt unter den enthaltenen Einträgen
                                           // den zuerst eingefügten und gibt ihn zurück.
void fifo_destroy(fifo_t *fifo);           // Gibt Speicher frei.
```

Beispiel:

```
fifo_t *f = fifo_init(2);
fifo_enqueue(f, 7);
fifo_enqueue(f, 8);
assert(fifo_dequeue(f) == 7);
fifo_enqueue(f, 9);
assert(fifo_dequeue(f) == 8);
fifo_destroy(f);
```

**Aufgabe 1:**

Erweitere die Implementierung aus der vorigen Aufgabe wie folgt!

Falls bei `fifo_enqueue` nicht ausreichend Speicher verfügbar ist (und nur dann), wird mehr Speicher alloziert. Dabei soll in einer Sequenz von $n$ Aufrufen von `fifo_enqueue` höchstens $O(\log(n))$ mal Speicher alloziert werden. Die Menge des von `fifo_enqueue` allozierten Speichers soll höchstens linear in der Größe des benutzen Speichers sein. In einer Sequenz von $n$ Aufrufen von `fifo_dequeue` soll höchstens $O(\log(n))$ mal Speicher freigeben werden. Die Warteschlange soll (falls es mindestens so viele Einträge wie bei `fifo_init`angegeben gibt) nie mehr als zehnmal soviel allozierten Speicher halten, wie zur Speicherung der Einträge nötig.

Beispiel:

```
fifo_t *f = fifo_init(0);
fifo_enqueue(f, 8);
assert(fifo_dequeue(f) == 8);
fifo_destroy(f);
```

**Aufgabe 2:**

Erweitere das Spiel wumpus aus Aufgabe 3, Blatt 4 wie folgt!

Es soll in der Höhle in zwei Räumen Abgründe geben. Wenn der Spieler einen solchen Raum betritt, stürzt er hinein und stirbt. In Räumen, aus denen man einen Abgrund in einem Schritt erreichen kann, ist ein Luftzug zu bemerken.

**Aufgabe 3:**

Erweitere das Spiel wumpus aus Aufgabe 2 wie folgt! Der Spieler verfügt über 3 magische Pfeile. Jeder Pfeil kann nach dem Abschuss durch bis zu 3 andere Räume fliegen. Der Spieler gibt bei Abschuss diese drei Räume in ihrer Reihenfolge an. Sollte der Pfeil einen Raum erreichen, in dem es nicht möglich ist, wie vom Spieler gewünscht weiterzufliegen, fliegt der Pfeil in einen zufälligen erreichbaren Raum. Fliegt ein Pfeil durch den Raum, in dem sich der Wumpus befindet, trifft er diesen und das Spiel ist gewonnen. Fliegt ein Pfeil durch den Raum, in dem sich der Spieler befindet, trifft er diesen und das Spiel ist verloren. Ansonsten bemerkt der Wumpus den Schuss, und begibt sich in einen zufälligen für ihn in einem Schritt erreichbaren Raum (aufgrund seiner Größe stürzt er nicht in Abgründe, und wird auch nicht von Fledermäusen herumgetragen).

**Aufgabe 4:**

Ändere deine Lösung von Aufgabe 1, Blatt 3 so, dass im String das erste "x" gefunden wird, das direkt auf ein "a" oder ein "ä" folgt.

**Aufgabe 5:**

Im Archiv auf der Website findet sich eine Datei `gles_square.c`, sowie einige Headerdateien in einem Unterverzeichnis `GLES`. Diese lassen sich mittels `gcc -I. gles_square.c -lGLESv1_CM -lglfw` zu einen ausführbaren Programm kompilieren. Das Programm stellt mittels OpenGL ES ein weißes Quadrat in einem ansonsten schwarzen Fenster dar. Übernimm die Dateien, und passe dein Makefile so an, dass aus `gles_square.c` eine Binärdatei `gles_square` erstellt wird (Kompilieren und Linken sollen dabei getrennt erfolgen, also aus `gles_square.c` erst eine `gles_square.o` erstellt werden). Tests sind nicht nötig.

Hinweise (gelten für dieses Blatt):

- Obwohl es mehr als 4 Aufgaben gibt, gibt es pro Aufgabe 4 Punkte. Die Gesamtzahl der zu Bestehen nötigen Punkte ändert sich nicht. Man kann also zwei der Aufgaben als Bonusaufgaben ansehen.
- Warteschlangen von Elementen bekannter Größe lassen sich als Ringpuffer wie folgt implementieren: Man verwendet ein Feld, sowie zwei Zeiger oder Indizes für das Feld, die Anfang und Ende des als Puffer genutzten Bereichs im Feld angeben. Dabei erfolgt ein Wraparound, d.h. wenn an einem Ende des Puffers etwas angefügt wird, und noch Platz im Feld ist wird dieser anderen Ende des Feldes genutzt (bei Verwendung von Indizes für Anfang und Ende würde man diese also modulo Feldgröße rechnen - nach dem letzten Element käme dann als nächstes das Element 0 dran, entsprechend der Addition in einem Restklassenring $\mathbb{Z}/n\mathbb{Z}$ für eine Zahl $n$).
- Wenn man Aufgabe 0, aber nicht Aufgabe 1 bearbeitet, reicht es, für den Ringpuffer ein Feld in der Initalisierungsfunktion anzulegen, und dessen Größe später bei `fifo_enqueue` nicht mehr zu verändern. In diesem Fall würde `fifo_enqueue` 1 zurückgeben, wenn dieses Feld nicht mehr ausreicht.