Aufgabe 8.1 Binäre Heaps

In der Datei `heap.c` findest Du ein Ausgangssystem für die Datenstruktur Binärer Heap. Um die Index-Berechnung für Väter und Söhne zu vereinfachen, verwenden wir das Array-Element an der Position 0 nicht für Nutzdaten. Kannst Du einen sinnvollen Wert darin speichern, der die Schleifenbedingung beim Einfügen vereinfacht?

Für Debugging-Zwecke sind bereits zwei hilfreiche Funktionen implementiert:

```
/* Schreibt den Heap zweidimensional auf die Konsole. */
void heap__print(const struct heap * self);
/* Liefert das Element am angegebenen Index zurück bzw. beendet
das Programm, falls der Index ungültig ist. Viele illegale
Zugriffe außerhalb des Heaps blieben ansonsten unerkannt. Dazu
musst Du heap_at(self, i) statt self->data[i] verwenden. */
int heap__at(const struct heap * self, int index);
```

In einer weiteren Debugging-Funktion musst Du noch den Rumpf vervollständigen:

```
/* Stellt sicher, dass kein Kind einen größeren Vater hat. */
void heap__assert(const struct heap * self);
```

Implementiere zuletzt die beiden eigentlich interessanten O(log n)-Operation für Heaps:

```
void heap__insert(struct heap * self, int element);
int heap__remove_minimum(struct heap * self);
```

Aufgabe 8.2 Heapsort mit separatem Heap

Verwende den binären Heap aus Aufgabe 8.1, um Heapsort zu implementieren:

```
void heapsort(int * data, int len);
```

Dazu musst Du einfach nur einen lokalen Heap mit den Zahlen aus dem Array befüllen, und anschließend die Zahlen wieder aus dem Heap entfernen und ins Array schreiben.

Vergleiche die Laufzeit Deines Algorithmus mit der von `qsort`. Um vernünftige Messergebnisse zu erhalten, musst Du die Projekt-Konfiguration auf „Release“ schalten und ein sehr großes Array sortieren. Der Sortiervorgang sollte mehrere Sekunden dauern.

Achtung: Wenn Du Sortierverfahren miteinander vergleichst, sollten alle auf denselben Eingabedaten arbeiten. Auf keinen Fall solltest Du ein Array erzeugen und dieses Array nacheinander durch alle Sortierverfahren schicken, weil dann lediglich das erste Sortierverfahren wirklich sortieren muss und die restlichen Verfahren anschließend auf einem bereits sortierten Array arbeiten. Vorsortierte Eingaben können den Sortiervorgang, je nach verwendetem Sortierverfahren, extrem beschleunigen (aber auch ausbremsen).

Aufgabe 8.3 Monatsnamen in Kurzform

In der Datei `months.c` befinden sich vier Funktionen mit derselben Funktionalität; sie wandeln eine Monatsnummer (1-12) in den entsprechenden Monatsnamen in Kurzform um (Jan, Feb, Mar, Apr, Mai, Jun, Jul, Aug, Sep, Okt, Nov, Dez). Die zugrundeliegende Datenstruktur unterscheidet sich dabei von Funktion zu Funktion. Verwendet werden:

a) ein zweidimensionales Array von Zeichen

b) ein eindimensionales Array von Zeigern auf Zeichen

c) ein eindimensionales Array von Zeichen

d) ein Zeiger auf Zeichen

Verdeutliche Dir den Unterschied dieser vier Varianten auf Papier. Mache Dir Gedanken darüber, in welchen Speicherbereichen die Bestandteile der Datenstruktur liegen.

An welchen Stellen ist das `static` wegen der Lebensdauer dringend erforderlich, und an welchen Stellen ist es lediglich eine sinnvolle Optimierung, damit die Variable `names` nicht bei jedem Funktionsaufruf erneut initialisiert werden muss?

Entferne testweise jeweils ein `static` und überprüfe, wie sich dies auf die Ausgabe auswirkt. Achte auch darauf, ob der Compiler entsprechende Warnungen ausgibt.

Aufgabe 8.4 Interne Darstellung von Gleitkommazahlen

Tippe das folgende Programm ab und probiere es aus:

```
#include <stdio.h>
#include <string.h>   // memcpy


int main()
{
    float pi = 3.14159265;
    unsigned int bits;
    memcpy(&bits, &pi, 4);
    printf("%.8f -> %08x\n", pi, bits);
    return 0;
}
```

Wenn Du alles richtig gemacht hast, dann sollte folgendes auf der Konsole erscheinen:

```
3.14159274 -> 40490fdb
```

Auf der linken Seite sehen wir die beste Näherung an π im float-Wertebereich. Auf der rechten Seite sehen wir, mit welchen Bits dieser Wert gemäß IEEE754-Standard kodiert ist:

```
   4    0    4    9    0    f    d    b
0100 0000 0100 1001 0000 1111 1101 1011
```

Mache Dir anhand der Folien klar, was diese 32 Bits bedeuten.

Wie ändert sich das Bitmuster, wenn…

a) wir die Zahl negieren?
b) wir die Zahl verdoppeln?
c) wir die Zahl halbieren?

Schreibe nun eine Funktion `void study_float(float x)`, welche die einzelnen Komponenten der Gleitkommazahl (Vorzeichen, Mantisse, Exponent) in lesbarer Form auf die Konsole schreibt. Die Klasse (Normalisiert, Denormalisiert, Null, Unendlich, Keine Zahl) soll ebenfalls bestimmt werden. Erneut am Beispiel der Zahl π:

```
3.14159274 -> 40490fdb
sgn: +
man: 1.921fb6
exp: 1
normalized number
```