

Hardwarenahe Programmierung

Übungsaufgaben – 23. Januar 2020

Diese Übung enthält Punkteangaben wie in einer Klausur. Um zu „bestehen“, müssen Sie innerhalb von 90 Minuten unter Verwendung ausschließlich zugelassener Hilfsmittel 16 Punkte (von insgesamt 32) erreichen.

Aufgabe 1: Stack-Operationen

Das folgende Programm ([aufgabe-1.c](#)) implementiert einen Stapelspeicher (Stack). Dies ist ein Array, das nur bis zu einer variablen Obergrenze (Stack-Pointer) tatsächlich genutzt wird. An dieser Obergrenze kann man Elemente hinzufügen (push).

In dieser Aufgabe sollen zusätzlich Elemente in der Mitte eingefügt werden (insert). Die dafür bereits existierenden Funktionen `insert()` und `insert_sorted()` sind jedoch fehlerhaft.

```
#include <stdio.h>

#define STACK_SIZE 10

int stack[STACK_SIZE];
int stack_pointer = 0;

void push (int x)
{
    stack[stack_pointer++] = x;
}

void show (void)
{
    printf ("stack_content:");
    for (int i = 0; i < stack_pointer; i++)
        printf ("_%d", stack[i]);
    if (stack_pointer)
        printf ("\n");
    else
        printf ("_(empty)\n");
}

void insert (int x, int pos)
{
    for (int i = pos; i < stack_pointer; i++)
        stack[i + 1] = stack[i];
    stack[pos] = x;
    stack_pointer++;
}

void insert_sorted (int x)
{
    int i = 0;
    while (i < stack_pointer && x < stack[i])
        i++;
    insert (x, i);
}

int main (void)
{
    push (3);
    push (7);
    push (137);
    show ();
    insert (5, 1);
    show ();
    insert_sorted (42);
    show ();
    insert_sorted (2);
    show ();
    return 0;
}
```

- (a) Korrigieren Sie das Programm so, daß die Funktion `insert()` ihren Parameter `x` an der Stelle `pos` in den Stack einfügt und den sonstigen Inhalt des Stacks verschiebt, aber nicht zerstört. (3 Punkte)
- (b) Korrigieren Sie das Programm so, daß die Funktion `insert_sorted()` ihren Parameter `x` an derjenigen Stelle einfügt, an die er von der Sortierung her gehört. (Der Stack wird hierbei vor dem Funktionsaufruf als sortiert vorausgesetzt.) (2 Punkte)
- (c) Schreiben Sie eine zusätzliche Funktion `int search (int x)`, die die Position (Index) des Elements `x` innerhalb des Stack zurückgibt – oder die Zahl `-1`, wenn `x` nicht im Stack enthalten ist. Der Rechenaufwand darf höchstens $\mathcal{O}(n)$ betragen. (3 Punkte)
- (d) Wie (c), aber der Rechenaufwand darf höchstens $\mathcal{O}(\log n)$ betragen. (4 Punkte)

Aufgabe 2: Iterationsfunktionen

Wir betrachten das folgende Programm ([aufgabe-2.c](#)):

```
#include <stdio.h>

void foreach (int *a, void (*fun) (int x))
{
    for (int *p = a; *p >= 0; p++)
        fun (*p);
}

void even_or_odd (int x)
{
    if (x % 2)
        printf ("%d_ist_ungerade.\n", x);
    else
        printf ("%d_ist_gerade.\n", x);
}
```

```
int main (void)
{
    int numbers[] = { 12, 17, 32, 1, 3, 16, 19, 18, -1 };
    foreach (numbers, even_or_odd);
    return 0;
}
```

- (a) Was bedeutet **void (*fun) (int x)**, und welchen Sinn hat seine Verwendung in der Funktion **foreach()**? (2 Punkte)
- (b) Schreiben Sie das Hauptprogramm **main()** so um, daß es unter Verwendung der Funktion **foreach()** die Summe aller positiven Zahlen in dem Array berechnet. Sie dürfen dabei weitere Funktionen sowie globale Variable einführen. (4 Punkte)

Aufgabe 3: Dynamisches Bit-Array

Schreiben Sie die folgenden Funktionen zur Verwaltung eines dynamischen Bit-Arrays:

- **void bit_array_init (int n)**
Das Array initialisieren, so daß man **n** Bits darin speichern kann.
Die Array-Größe **n** ist keine Konstante, sondern erst im laufenden Programm bekannt.
Die Bits sollen auf den Anfangswert 0 initialisiert werden.
- **void bit_array_set (int i, int value)**
Das Bit mit dem Index **i** auf den Wert **value** setzen.
Der Index **i** darf von 0 bis **n - 1** gehen; der Wert **value** darf 1 oder 0 sein.
- **void bit_array_flip (int i)**
Das Bit mit dem Index **i** auf den entgegengesetzten Wert setzen, also auf 1, wenn er vorher 0 ist, bzw. auf 0, wenn er vorher 1 ist.
Der Index **i** darf von 0 bis **n - 1** gehen.
- **int bit_array_get (int i)**
Den Wert des Bit mit dem Index **i** zurückliefern.
Der Index **i** darf von 0 bis **n - 1** gehen.
- **void bit_array_resize (int new_n)**
Die Größe des Arrays auf **new_n** Bits ändern.
Dabei soll der Inhalt des Arrays, soweit er in die neue Größe paßt, erhalten bleiben.
Neu hinzukommende Bits sollen auf 0 initialisiert werden.
- **void bit_array_done (void)**
Den vom Array belegten Speicherplatz wieder freigeben.

Bei Bedarf dürfen Sie den Funktionen zusätzliche Parameter mitgeben, beispielsweise um mehrere Arrays parallel verwalten zu können. (In der objektorientierten Programmierung wäre dies der implizite Parameter **this**, der auf die Objekt-Struktur zeigt.)

Die Bits sollen möglichst effizient gespeichert werden, z. B. jeweils 8 Bits in einer **uint8_t**-Variablen.

Die Funktionen sollen möglichst robust sein, d. h. das Programm darf auch bei unsinnigen Parameterwerten nicht abstürzen, sondern soll eine Fehlermeldung ausgeben.

Die **Hinweise** auf der nächsten Seite beschreiben einen möglichen Weg, die Aufgabe zu lösen.
Es sieht Ihnen frei, die Aufgabe auch auf andere Weise zu lösen.

Hinweise:

- Setzen Sie zunächst voraus, daß das Array die konstante Länge 8 hat, und schreiben Sie zunächst nur die Funktionen `bit_array_set()`, `bit_array_flip()` und `bit_array_get()`.
- Verallgemeinern Sie nun auf eine konstante Länge, bei der es sich um ein Vielfaches von 8 handelt.
- Implementieren Sie nun die Überprüfung auf unsinnige Parameterwerte. Damit können Sie sich gleichzeitig von der Bedingung lösen, daß die Länge des Arrays ein Vielfaches von 8 sein muß.
- Gehen Sie nun von einem statischen zu einem dynamischen Array über, und implementieren sie die Funktionen `bit_array_init()`, `bit_array_done()` und `bit_array_resize()`.

(14 Punkte)

(Hinweis für die Klausur: Abgabe in digitaler Form ist erwünscht, aber nicht zwingend.)

Viel Erfolg – auch in der Klausur!