

# Hardwarenahe Programmierung

## Übungsaufgaben 11 – 16. Januar 2025

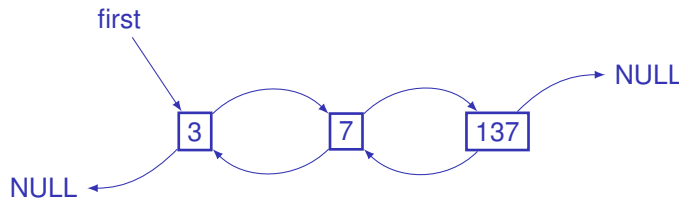
Diese Übung enthält Punkteangaben wie in einer Klausur. Um zu „bestehen“, müssen Sie innerhalb von 140 Minuten unter Verwendung ausschließlich zugelassener Hilfsmittel 24 Punkte (von insgesamt 49) erreichen.

### Aufgabe 1: Einfach und doppelt verkettete Listen

Das Beispiel-Programm [aufgabe-1.c](#) demonstriert zwei Funktionen zur Verwaltung einfach verketteter Listen: [output\\_list\(\)](#) zum Ausgeben der Liste auf den Bildschirm und [insert\\_into\\_list\(\)](#) zum Einfügen in die Liste.

- (a) Ergänzen Sie eine Funktion [delete\\_from\\_list\(\)](#) zum Löschen eines Elements aus der Liste mit Freigabe des Speicherplatzes. (5 Punkte)
- (b) Ergänzen Sie eine Funktion [reverse\\_list\(\)](#) die die Reihenfolge der Elemente in der Liste umdreht. (3 Punkte)

Eine doppelt verkettete Liste hat in jedem Knotenpunkt ([node](#)) *zwei* Zeiger – einen auf das nächste Element ([next](#)) und einen auf das vorherige Element (z. B. [prev](#) für „previous“). Dadurch ist es leichter als bei einer einfach verketteten Liste, die Liste in umgekehrter Reihenfolge durchzugehen.



Der Rückwärts-Zeiger ([prev](#)) des ersten Elements zeigt, genau wie der Vorwärts-Zeiger ([next](#)) des letzten Elements, auf *nichts*, hat also den Wert [NULL](#).

- (c) Schreiben Sie das Programm um für doppelt verkettete Listen. (5 Punkte)

### Aufgabe 2: Ternärer Baum

Der in der Vorlesung vorgestellte *binäre Baum* ist nur ein Spezialfall; im allgemeinen können Bäume auch mehr als zwei Verzweigungen pro Knotenpunkt haben. Dies ist nützlich bei der Konstruktion *balancierter Bäume*, also solcher, die auch im *Worst Case* nicht zu einer linearen Liste entarten, sondern stets eine – möglichst flache – Baumstruktur behalten.

Wir betrachten einen Baum mit bis zu drei Verzweigungen pro Knotenpunkt, einen sog. *ternären Baum*. Jeder Knoten enthält dann nicht nur einen, sondern *zwei* Werte als Inhalt:

```
typedef struct node
{
    int content_left, content_right;
    struct node *left, *middle, *right;
} node;
```

Wir konstruieren nun einen Baum nach folgenden Regeln:

- Innerhalb eines Knotens sind die Werte sortiert: [content\\_left](#) muß stets kleiner sein als [content\\_right](#).
- Der Zeiger [left](#) zeigt auf Knoten, deren enthaltene Werte durchweg kleiner sind als [content\\_left](#).
- Der Zeiger [right](#) zeigt auf Knoten, deren enthaltene Werte durchweg größer sind als [content\\_right](#).
- Der Zeiger [middle](#) zeigt auf Knoten, deren enthaltene Werte durchweg größer sind als [content\\_left](#), aber kleiner als [content\\_right](#).

- Ein Knoten muß nicht immer mit zwei Werten voll besetzt sein; er darf auch *nur einen* gültigen Wert enthalten.  
Der Einfachheit halber lassen wir in diesem Beispiel nur positive Zahlen als Werte zu. Wenn ein Knoten nur einen Wert enthält, setzen wir `content_right = -1`, und der Zeiger `middle` wird nicht verwendet.
- Wenn wir neue Werte in den Baum einfügen, werden *zuerst* die nicht voll besetzten Knoten aufgefüllt und *danach erst* neue Knoten angelegt und Zeiger gesetzt.
- Beim Auffüllen eines Knotens darf nötigenfalls `content_left` nach `content_right` verschoben werden. Ansonsten werden einmal angelegte Knoten nicht mehr verändert.

(In der Praxis dürfen Knoten gemäß speziellen Regeln nachträglich verändert werden, um Entartungen gar nicht erst entstehen zu lassen – siehe z. B. [https://en.wikipedia.org/wiki/2-3\\_tree](https://en.wikipedia.org/wiki/2-3_tree).)

- Zeichnen Sie ein Schaubild, das veranschaulicht, wie die Zahlen 7, 137, 3, 5, 6, 42, 1, 2 und 12 nacheinander und in dieser Reihenfolge in den oben beschriebenen Baum eingefügt werden – analog zu den Vortragsfolien ([hp-20250116.pdf](#)), Seite 33. (3 Punkte)
- Dasselbe, aber in der Reihenfolge 2, 7, 42, 12, 1, 137, 5, 6, 3. (3 Punkte)
- Beschreiben Sie in Worten und/oder als C-Quelltext-Fragment, wie eine Funktion aussehen müßte, um den auf diese Weise entstandenen Baum sortiert auszugeben. (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 folgenden **Hinweise** beschreiben einen möglichen Weg, die Aufgabe zu lösen. Es sieht Ihnen frei, die Aufgabe auch auf andere Weise zu lösen.

- 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.)

## Aufgabe 4: Stack-Operationen

Das folgende Programm ([aufgabe-4.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;
}
```

- 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)
- 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)

*Viel Erfolg – auch in der Klausur!*