

Hardwarenahe Programmierung

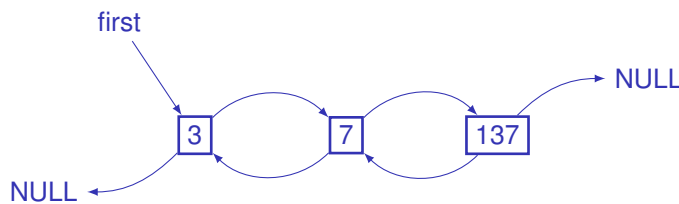
Musterlösung zu den Übungsaufgaben 11 – 16. Januar 2025

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)

Lösung

- (a) **Ergänzen Sie eine Funktion [delete_from_list\(\)](#) zum Löschen eines Elements aus der Liste mit Freigabe des Speicherplatzes.**

Siehe: [loesung-1a.c](#)

Um ein Element aus einer verketteten Liste zu löschen, müssen zuerst die Zeiger umgestellt werden, um das Element von der Liste auszuschließen. Erst danach darf der Speicherplatz für das Element freigegeben werden.

Man benötigt also *das vorherige Element*, dessen [next](#)-Zeiger man dann auf das übernächste Element [next->next](#) setzt.

Bei jedem Zeiger muß man vor dem Zugriff prüfen, daß dieser nicht auf [NULL](#) zeigt. (Die Musterlösung ist in dieser Hinsicht nicht konsequent. Für den Produktiveinsatz müßte z. B. [delete_from_list\(\)](#) auch den übergebenen Zeiger [what](#) auf [NULL](#) prüfen.)

Ein Spezialfall tritt ein, wenn das erste Element einer Liste entfernt werden soll. In diesem Fall tritt [first](#) an die Stelle des [next](#)-Zeigers des (nicht vorhandenen) vorherigen Elements. Da [delete_from_list\(\)](#) *schreibend* auf [first](#) zugreift, muß [first](#) als Zeiger übergeben werden ([node **first](#)).

Um alle Spezialfälle zu testen (am Anfang, am Ende und in der Mitte der Liste), wurden die Testfälle im Hauptprogramm erweitert.

- (b) **Schreiben Sie das Programm um für doppelt verkettete Listen.**

Siehe: [loesung-1b.c](#)

Bei allen Einfüge- und Löschaktionen müssen *jeweils zwei* [next](#)- und [prev](#)-Zeiger neu gesetzt werden. Zum Debuggen empfiehlt es sich sehr, eine Funktion zu schreiben, die die Liste auf Konsistenz prüft (hier: [check_list\(\)](#)).

Das Testprogramm macht von der Eigenschaft der doppelt verketteten Liste, daß man sie auch rückwärts effizient durchgehen kann, keinen Gebrauch. Um diese Eigenschaft als Vorteil nutzen zu können, empfiehlt es sich, zusätzlich zu [first](#) auch einen Zeiger auf das letzte Element (z. B. [last](#)) einzuführen. Dieser muß dann natürlich bei allen Operationen (Einfügen, Löschen, ...) auf dem aktuellen Stand gehalten werden.

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:

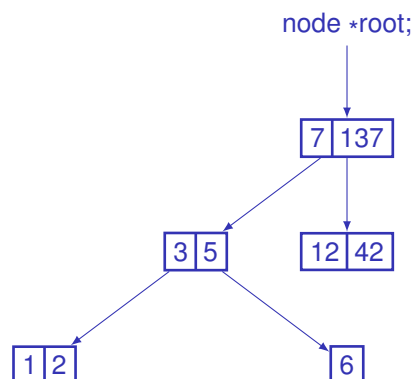
- Innerhalb eines Knotens sind die Werte sortiert: `content_left` muß stets kleiner sein als `content_right`.
- Der Zeiger `left` zeigt auf Knoten, deren enthaltene Werte durchweg kleiner sind als `content_left`.
- Der Zeiger `right` zeigt auf Knoten, deren enthaltene Werte durchweg größer sind als `content_right`.
- 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.)

- 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)

Lösung

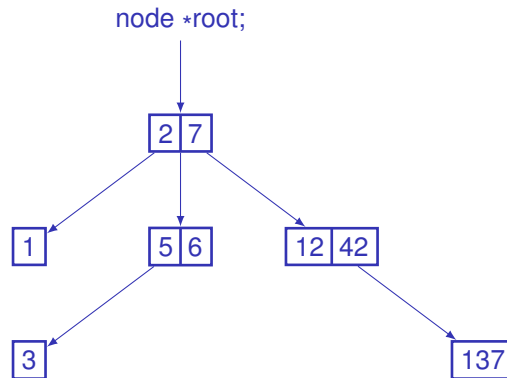
- Zeichnen Sie ein Schaubild, das veranschaulicht, wie die Zahlen 7, 137, 3, 5, 6, 42, 1, 2 und 12 nacheinander und in dieser Reihenfolge in den oben beschriebenen Baum eingefügt werden – analog zu den Vortragsfolien ([hp-20250116.pdf](#)), Seite 21.**



Bemerkungen:

- Zeiger mit dem Wert `NULL` sind nicht dargestellt: `right`-Zeiger von 7/137, `middle`-Zeiger von 3/5, sämtliche Zeiger von 1/2, 12/42 und 6.
- Beim Einfügen der 12 wird die sich bereits vorher in diesem `node` befindliche 42 zu `content_right`, und die 12 wird das neue `content_left`.
- Dieser Baum hat sehr einfache Regeln und ist daher *nicht* balanciert. Insbesondere unsere Regel, daß einmal angelegte Knoten nicht mehr verändert werden dürfen, steht dem im Wege. Ein einfaches Beispiel für einen *balancierten* ternären Baum ist der 2-3-Baum – siehe z. B. https://en.wikipedia.org/wiki/2-3_tree.

(b) **Dasselbe, aber in der Reihenfolge 2, 7, 42, 12, 1, 137, 5, 6, 3.**



Bemerkungen:

- Wieder sind Zeiger mit dem Wert `NULL` nicht dargestellt: `middle`- und `right`-Zeiger von 5/6, `left`- und `middle`-Zeiger von 12/42, sämtliche Zeiger von 1, 3 und 137.
- Beim Einfügen der 12 wird wieder die sich bereits vorher in diesem `node` befindliche 42 zu `content_right`, und die 12 wird das neue `content_left`.

(c) **Beschreiben Sie in Worten und/oder als C-Quelltext-Fragment, wie eine Funktion aussehen müßte, um den auf diese Weise entstandenen Baum sortiert auszugeben.**

Die entscheidende Idee ist **Rekursion**.

Eine Funktion, die den gesamten Baum ausgibt, müßte einmalig für den Zeiger `root` aufgerufen werden und folgendes tun:

1. falls der übergebene Zeiger den Wert `NULL` hat, nichts ausgeben, sondern die Funktion direkt beenden,
2. sich selbst für den `left`-Zeiger aufrufen,
3. den Wert von `content_left` ausgeben,
4. sich selbst für den `middle`-Zeiger aufrufen,
5. sofern vorhanden (also ungleich `-1`), den Wert von `content_right` ausgeben,
6. sich selbst für den `right`-Zeiger aufrufen.

Als C-Fragment:

```
void output_tree (node *root)
{
    if (root)
    {
        output_tree (root->left);
        printf ("%d\n", root->content_left);
        output_tree (root->middle);
        if (root->content_right >= 0)
            printf ("%d\n", root->content_right);
        output_tree (root->right);
    }
}
```

Die Datei `loesung-2c.c` erweitert dieses Fragment zu einem vollständigen C-Programm zum Erstellen und sortierten Ausgeben eines ternären Baums mit den Zahlenwerten von Aufgabenteil (a).

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 steht 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.)

Lösung

Die hier vorgestellte Lösung folgt den Hinweisen.

- **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()`.**

Siehe: [loesung-3-1.c](#)

Wir speichern in jedem der acht Bit einer `uint8_t`-Variablen jeweils eine Zahl, die 0 oder 1 sein kann. Dies geschieht durch Setzen bzw. Löschen bzw. Umklappen einzelner Bits in der Variablen.

Das Programm enthält zusätzlich eine Funktion `output()`, mit der man sich den Inhalt des Arrays anzeigen lassen kann, sowie ein Hauptprogramm `main()`, um die Funktionen zu testen.

- **Verallgemeinern Sie nun auf eine konstante Länge, bei der es sich um ein Vielfaches von 8 handelt.**

Siehe: [loesung-3-2.c](#)

In diesem Programm setzen wir die Länge auf konstant `LENGTH` Bits, wobei es sich um eine Präprozessor-Konstante mit dem Wert 32 handelt.

Um `LENGTH` Bits zu speichern, benötigen wir ein Array der Länge `LENGTH / 8` Bytes.

Um auf ein einzelnes Bit zuzugreifen, müssen wir zunächst ermitteln, in welchem der Bytes sich befindet. Außerdem interessieren wir uns für die Nummer des Bits innerhalb des Bytes. Den Array-Index des Bytes erhalten wir, indem wir den Index des Bits durch 8 dividieren. Der bei dieser Division verbleibende Rest ist die Nummer des Bits innerhalb des Bytes.

Diese Rechnungen führen wir in den drei Funktionen `bit_array_set()`, `bit_array_flip()` und `bit_array_get()` durch. (Diese ist eine eher unelegante Code-Verdopplung – hier sogar eine Verdreifachung. Für den Produktiveinsatz lohnt es sich, darüber nachzudenken, wie man diese vermeiden kann, ohne gleichzeitig an Effizienz einzubüßen. Hierfür käme z. B. ein Präprozessor-Makro in Frage. Für die Lösung der Übungsaufgabe wird dies hingegen nicht verlangt.)

- **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.**

Siehe: [loesung-3-3.c](#)

Um weitere Code-Verdopplungen zu vermeiden, führen wir eine Funktion `check_index()` ein, die alle Prüfungen durchführt.

Wenn die Länge des Arrays kein Vielfaches von 8 ist, wird das letzte Byte nicht vollständig genutzt. Die einzige Schwierigkeit besteht darin, die korrekte Anzahl von Bytes zu ermitteln, nämlich die Länge dividiert durch 8, aber nicht ab-, sondern aufgerundet. Am elegantesten geht dies durch vorherige Addition von 7: `#define BYTES ((LENGTH + 7) / 8)`. Es ist aber auch zulässig, die Anzahl der Bytes mit Hilfe einer `if`-Anweisung zu ermitteln: Länge durch 8 teilen und abrunden; falls die Division nicht glatt aufging, um 1 erhöhen.

- **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()`.**

Siehe: [loesung-3-4.c](#). Damit ist die Aufgabe gelöst.

Aus den Präprozessor-Konstanten `LENGTH` und `BYTES` werden nun globale `int`-Variable. Die Funktion `bit_array_init()` berechnet die korrekten Werte für diese Variablen und legt das Array mittels `malloc()` dynamisch an. Eine Größenänderung des Arrays erfolgt mittels `realloc()`, das Freigeben mittels `free()`.

Das Programm setzt Variable, die aktuell nicht verwendet werden, auf den Wert 0 bzw. `NULL`. Dies ermöglicht es der Funktion `check_index()`, auch zu prüfen, ob das Array vorher korrekt mit `bit_array_init()` erzeugt wurde – oder ob es vielleicht schon wieder mit `bit_array_done()` freigegeben wurde.

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)
- 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)
- Wie (c), aber der Rechenaufwand darf höchstens $\mathcal{O}(\log n)$ betragen. (4 Punkte)

Lösung

- 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.**

Die `for`-Schleife in der Funktion `insert()` durchläuft das Array von unten nach oben. Um den Inhalt des Arrays von unten nach oben zu verschieben, muß man die Schleife jedoch von oben nach unten durchlaufen.

Um die Funktion zu reparieren, ersetze man also

```
for (int i = pos; i < stack_pointer; i++)
```

durch:

```
for (int i = stack_pointer - 1; i >= pos; i--)
```

(Siehe auch: [loesung-4.c](#))

- (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.)**

Der Vergleich `x < stack[j]` als Bestandteil der `while`-Bedingung paßt nicht zur Durchlaufrichtung der Schleife (von unten nach oben).

Um die Funktion zu reparieren, kann man daher entweder das Kleinerzeichen durch ein Größerzeichen ersetzen (`x > stack[j]`) – siehe [loesung-4b-1.c](#)) oder die Schleife von oben nach unten durchlaufen (siehe [loesung-4b-2.c](#)).

Eine weitere Möglichkeit besteht darin, das Suchen nach der Einfügeposition mit dem Verschieben des Arrays zu kombinieren (siehe [loesung-4.c](#)). Hierdurch spart man sich eine Schleife; das Programm wird schneller. (Es bleibt allerdings bei $\mathcal{O}(n)$.)

- (c) **Schreiben Sie eine zusätzliche Funktion `int search (int x)`, die die Position (Index) des Elements `x` innerhalb des Stack zurückgibt – oder `-1`, wenn `x` nicht im Stack enthalten ist. Der Rechenaufwand darf höchstens $\mathcal{O}(n)$ betragen.**

Man geht in einer Schleife den Stack (= den genutzten Teil des Arrays) durch. Bei Gleichheit gibt man direkt mit `return` den Index zurück. Nach dem Schleifendurchlauf steht fest, daß `x` nicht im Stack vorhanden ist; man kann dann direkt `-1` zurückgeben (siehe [loesung-4c.c](#)).

Da es sich um eine einzelne Schleife handelt, ist die Ordnung $\mathcal{O}(n)$.

- (d) **Wie (c), aber der Rechenaufwand darf höchstens $\mathcal{O}(\log n)$ betragen.**

Um $\mathcal{O}(\log n)$ zu erreichen, halbiert man fortwährend das Intervall von (einschließlich) `0` bis (ausschließlich) `stack_pointer` (siehe [loesung-4d.c](#)).

Wichtig ist, daß man nach dem Durchlauf der Schleife noch auf die Gleichheit `x == stack[left]` (insbesondere nicht: `stack[right]`) prüfen und ggf. `left` bzw. `-1` zurückgeben muß.