Aufgaben für die Konsultationen – Programmierübungen in C

Hendrik Fehr

30. April 2024

Aufgaben zweiter Teil

Aufgabe 7: Byte-Reihenfolge *(endianness)*

Bei Datentypen zu deren Darstellung mehrere Bytes notwendig sind, gibt es zwei offensichtliche Festlegungen der Byte-Reihenfolge, wenn man komplexere Zuordnungen außer acht lässt:

little endian Die niederwertigen Bytes stehen im Speicher an niedrigeren Adressen.

big endian Die niederwertigen Bytes stehen im Speicher an höheren Adressen.

Gelegentlich müssen diese Datentypen zerlegt oder zusammengesetzt werden, z. B. wenn sie von einer Kommunikationseinheit empfangen wurden oder wenn Sie gesendet werden sollen. Oft wird dies durch schieben und maskieren mit logischen Verknüpfungen erledigt, oder es kommen Unions zum Einsatz. Entwickeln Sie mindestens zwei Lösungen für jede der Aufgaben:

(a) Extrahieren des höherwertigen und niederwertigen Bytes eines Integers, sog. *low Byte* und *high Byte*.

(b) Setzen Sie ein Integer zusammen aus dem *low Byte* und *high Byte*.

Aufgabe 8: Rekursionen

Schreiben sie eine rekursive Funktion, zur Berechnung

(a) der Fakultät von n: `int fakultaet(int n)`. Die Fakultät für positive ganze Zahlen n ist definiert als $n! = \begin{cases} n(n-1)! & n > 1 \\ 1 & n = 1. \end{cases}$

(b) der Fibonacci-Zahl F_n: `int fibonacci(int n)`. Die Fibonacci-Zahlen sind $F_0 = 1$, $F_1 = 1$ und $F_n = F_{n-1} + F_{n-2}$ für $n \geq 2$.

Aufgabe 9: Primzahlsiebe

Implementieren Sie ein Primzahlsieb, bei dem in einer Primzahlkandidatenliste die Einträge als *nicht prim* markiert werden, für die ein Teiler ermittelt wurde. Die Varianten und Erweiterungen:

(a) Die Liste der Primzahlkandidaten enthält Einträge für alle Zahlen im Suchbereich.

(b) Die Liste der Primzahlkandidaten enthält nur Einträge für ungerade Zahlen im Suchbereich.

(c) Die untere und obere Grenze des Suchbereichs soll einstellbar sein.

(d) (Optional) Implementieren das Primzahlsieb von Atkin.

Aufgabe 10: Conways Spiel des Lebens

Das zweidimensionale Spielfeld hat diskrete Felder bzw. Zellen, deren binärer Zustand (*lebendig* oder *tot*) im nächsten Zeitschritt vom Zustand der Nachbarfelder im betrachteten Zeitschritt abhängig ist:

- Eine tote Zelle mit genau drei lebenden Nachbarn wird lebendig.
- Eine lebende Zelle mit weniger als zwei lebenden Nachbarn stirbt.
- Eine lebende Zelle mit zwei oder drei lebenden Nachbarn bleibt am Leben.
- Eine lebende Zelle mit mehr als drei lebenden Nachbarn stirbt.

(a) Implementieren Sie zunächst eine Schleife in welcher das Spielfeld und der Zeitschritt und ggf. weitere Informationen mit `printf` ausgegeben bevor auf Benutzereingabe gewartet wird. Durch Betätigen der Return-Taste wird der nächste Durchlauf gestartet. Ein Zeichen gefolgt von der Return-Taste, beendet die Anwendung. Das Spielfeld muss zunächst mit Platzhaltern versehen werden, weil die Spieldynamik noch nicht implementiert ist.

(b) Implementieren Sie die Spieldynamik gemäß der klassischen Regeln, siehe oben.

Aufgabe 11: Labyrinthproblem von Horowitz, Sahni und Anderson-Freed (für erfahrene)

Ein zweidimensionales Labyrinth mit einem Eingang A und einem Ausgang B kann durch ein Array von Zeichenketten dargestellt werden:

```
1 #define LAB_WIDTH 60
2 #define LAB_HEIGHT 16
3 char lab_1[LAB_HEIGHT][LAB_WIDTH] =
4    {"XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXBX",
```

```
    "XXXXXXXXXXXXXXX   XXXXXXXXXXXXXXXXX X",
    "X       X   XX  X X X       X X   X X",
    "X X   X   X     X X X   X       X X X",
    "X X X X XXXXXXXXX X X XXXXX XXXXX X X",
    "  X X   X         X       X   X     X",
    " XX X   X XXXXXXX XXXXXXX X   X XXXXX",
    "     XX       X X       X XXX X     X",
    "X X  XXXXXX X XXXXXXXXX X         X X",
    "X XXXX      X X       X XXXXXXXXXXX X",
    "X   XXXXXXXXX X XXXXX X       X   X X",
    "X X   X       X X   XXXXXXX X X XXX X",
    "XXX XXX X   X   X   X     X X X   X X",
    "X       X X XXXXX X XXXXX   X   XXX X",
    "X X       X       X   X   X   X     X",
    "XAXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"};
```

Eingang und Ausgang sind mit A bzw. B markiert. Mauern versperren den Weg und werden durch `"X"` markiert, freie Strecke wird durch `" "` markiert. Der gesuchte Algorithmus soll einen Weg durch das Labyrinth finden, wenn es einen gibt.

Eine systematische Lösung besteht darin, vom aktuellen Punkt aus die nächste der bisher noch nicht probierten Bewegungsrichtungen zu versuchen. Dabei geht man den nächsten Schritt solange der Weg frei ist und man vorher noch nicht dort gewesen ist. Wenn alle Bewegungsrichtungen erschöpft sind geht man zum vorherigen Feld zurück. Der Algorithmus terminiert wenn das Zielfeld erreicht ist, oder wenn alle Möglichkeiten erschöpft sind. In letzterem Fall gibt es keinen freien Weg von A nach B.

Die Teilaufgaben greifen relevanten Aspekte einer möglichen Lösung auf.

(a) Zur Eingabe eignen sich Array von Zeichenketten, wie oben gezeigt. Zur Implementierung der Suche ist es bequemer ein Array von Integerwerten zu nutzen. Außerdem soll das Labyrinth mit Mauern umrandet, also erweitert werden, damit die später zu implementierende Suche im Inneren des Labyrinths immer auf Feldern arbeitet, welche garantiert Nachbarfelder haben. Implementieren Sie das entsprechende Array und belegen die Werte gemäß der Zeichenketten vor. Sehen Sie geeignetes Verhalten bei fehlerhaften Eingaben (wie falschen Zeichen) vor.

(b) Ein Feld kann durch `[y][x]` indiziert werden und seine Nachbarfelder können durch Erhöhen bzw. Verringern von `x` und `y` angesprochen werden, solange `[y][x]` nicht ein Randfeld ist. Wenn man auf die Nachbarfelder bequem zugreifen kann, zum Beispiel um zu Testen ob der Weg frei ist, lässt sich der Algorithmus besonders lesbar implementieren.

Diese Idee wird in Tabelle 1 aufgegriffen. Diese gibt die Offsets für `x` und `y` an, wobei die Nachbarfelder durch Himmelsrichtungen bezeichnet werden. Die Himmelsrichtungen sind durch `dir` kodiert. Die Werte von `dir` können auch als Index für Arrays genutzt werden, welche die zugehörigen `x`- und `y`-Offsets enthalten. Schreiben Sie die entsprechenden Definitionen und Deklarationen.

Tabelle 1: Offsets für `x` und `y` in Abhängigkeit der Himmelsrichtung.

Himmelsrichtung	`dir`	`y`-Offset	`x`-Offset
N	0	-1	0
NO	1	-1	1
O	2	0	1
SO	3	1	1
S	4	1	0
SW	5	1	-1
W	6	0	-1
NW	7	-1	-1

(c) Der Verlauf von Position und Richtung soll in einem Stapel gespeichert werden, um das Problem auf praktikable Weise zu lösen. Schreiben sie die Funktionen `int add( int *top, struct element item)` und `struct element delete(int *top)`, die auf einen globalen Stapel wirken. Der Stapel besteht aus einem Array des noch zu definierenden Typs `struct element` sowie aus dem Index `int top` des obersten Eintrags. Ein leerer Stapel wird durch `top == -1` signalisiert. Die Initialisierung eines Stapels mit einer Kapazität von 100 Einträgen sieht dann aus wie folgt:

```
struct element { /* dummy-Felder nachher geeignet definieren */
  int dummy_a;
  int dummy_b;
};
#define MAX_STACK_SIZE 100
struct element stack[MAX_STACK_SIZE];
int top = -1;
```

(d) Der Algorithmus speichert den Kandidat für die Lösung im Stapel durch Angabe des Feldes und der zu wählenden Richtung. Definieren Sie die Einträge von `struct element`, damit das möglich ist.

(e) Horowitz, Sahni und Anderson-Freed schlagen folgende Struktur für den Algorithmus vor, angegeben in Pseudocode:

```
Initialisiere einen Stapel auf die Koordinaten des Eingangs des
   Labyriths und der Eingangsrichtung
Initialisiere das Array mark, indem die probierten Felder vermerkt
   werden.
while(Stapel ist nicht leer) {
        /* gehe auf die Spitze des Stapels */
        <y, x, dir> = Entferne die Spitze des Stapels;
```

```
        while(es andere Bewegungsrichtungen von hier aus gibt) {
                <next_y, next_x> = Koordinaten des naechsten Zuges;
                dir = Richtung des Zuges;
                if((next_y == EXIT_X) && (next_y == EXIT_Y) {
                        Erfolg = Ja;
                } else if(labyrinth[next_y][next_x] == 0
                                && mark[next_y][next_x] == 0) {
                        /* Erlaubter Zug und auch
                           noch nicht dort gewesen */
                        mark[next_y][next_x] = 1;
                        /* Sichere die momentane
                           Position und Richtung */
                        add <y, x, dir> oben auf den Stapel;
                        y = next_y;
                        x = next_x;
                        dir = erste zu probierende Richtung;
                }
        }
}
if(Erfolg?) {
        /* Ausgabe des Wegs */
} else {
        printf("Keinen Weg gefunden\n");
}
```

Vervollständigen, implementieren und testen Sie den oben spezifizierten Algorithmus.