

Hardwarenahe Programmierung / Angewandte Informatik

Musterlösung zu den Übungsaufgaben – 16. Januar 2017

Aufgabe 1: Stack-Operationen

Wir betrachten das folgende Programm ([aufgabe-1.c](#)):

```
#include <stdio.h>

#define STACK_SIZE 10

int stack[STACK_SIZE];
int stack_pointer = 0;

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

int pop (void)
{
    return stack[--stack_pointer];
}

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) Ändern 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) Ändern 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 `-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)

Lösung

- (a) **Ändern 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-1.c](#))

- (b) **Ändern 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[i]` 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[i]`) – siehe [loesung-1b-1.c](#)) oder die Schleife von oben nach unten durchlaufen (siehe [loesung-1b-2.c](#)).

Eine weitere Möglichkeit besteht darin, das Suchen nach der Einfügeposition mit dem Verschieben des Arrays zu kombinieren (siehe [loesung-1.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-1c.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-1d.c](#)) – wie in der Funktion `push_sorted()` im Beispiel-Programm [stack-11.c](#).

Ein wichtiger Unterschied besteht darin, 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.

Aufgabe 2: Iterativer Floodfill

In Übungsaufgabe 3 vom 14. 11. 2016 (siehe [hp-uebung-20161114.pdf](#)) haben wir einen rekursiven Floodfill-Algorithmus implementiert.

- (a) Ergänzen Sie die Funktion `fill()` so, daß Sie verfolgen können („Animation“), in welcher Reihenfolge die Punkte auf dem Bildschirm „ausgemalt“ werden. (4 Punkte)

Hinweis: Eine Möglichkeit, den Bildschirm zu löschen und eine kurze Zeit lang zu warten, können Sie den Beispielaufgaben zum Sortieren (z. B. [sort-2.c](#)) entnehmen. (Um `usleep()` nutzen zu können, ist beim Compilieren mit `gcc` die Option `-std=gnu99` erforderlich.)

- (b) Unter Verwendung eines Stack ist es möglich, den Floodfill-Algorithmus iterativ (also mit Schleife) statt rekursiv zu implementieren.

Informieren Sie sich per Web-Suche über die Details und implementieren Sie einen iterativen Floodfill-Algorithmus mit Stack für die Text-Grafik-Bibliothek. Die Punkte sollen dabei in derselben Reihenfolge „ausgemalt“ werden wie bisher in der rekursiven Floodfill-Implementation. (4 Punkte)

- (c) Ersetzen Sie nun den Stack durch einen FIFO. Der Floodfill-Algorithmus sollte weiterhin funktionieren, nur daß die Punkte in einer anderen Reihenfolge „ausgemalt“ werden. Beschreiben Sie diesen Unterschied. (3 Punkte)

(Ein iterativer Floodfill mit FIFO ist nützlich für die Wegfindung und verwandte Aufgabenstellungen.)

Lösung

- (a) **Ergänzen Sie die Funktion `fill()` so, daß Sie verfolgen können („Animation“), in welcher Reihenfolge die Punkte auf dem Bildschirm „ausgemalt“ werden.**

Siehe die Funktion `fill_animated()` in `textgraph.c` (mit `extern`-Deklaration in `textgraph.h` und Test-Aufruf in `test-textgraph.c`).

Um den Bildschirm zu löschen, wurde eine zusätzliche Funktion `clear_display()` eingeführt; für das Warten wird `usleep()` verwendet.

- (b) **Unter Verwendung eines Stack ist es möglich, den Floodfill-Algorithmus iterativ (also mit Schleife) statt rekursiv zu implementieren.**

Informieren Sie sich per Web-Suche über die Details und implementieren Sie einen iterativen Floodfill-Algorithmus mit Stack für die Text-Grafik-Bibliothek. Die Punkte sollen dabei in derselben Reihenfolge „ausgemalt“ werden wie bisher in der rekursiven Floodfill-Implementation.

Eine Web-Suche führt z. B. auf den Abschnitt „Iterative Flutfüllung“ des Wikipedia-Artikels zu „Floodfill“, https://de.wikipedia.org/wiki/Floodfill#Iterative_Flutf.C3.BCllung.

Für die Umsetzung des dort beschriebenen Algorithmus’ benötigt man einen Stack, der Koordinatenpaare speichern kann. Hierzu dienen die Deklarationen `struct pair`, `pair stack[STACK_SIZE]` und `int stack_pointer` zusammen mit den Funktionen `stack_push()` und `stack_pop()` in `textgraph.c` und `textgraph.h`.

Man beachte, daß die Funktion `stack_pop()` Zeiger verwendet, um die aus dem Stack geholten Werte `x` und `y` zurückzugeben. Beim Aufruf müssen entsprechend Adressen (`&x`, `&y`) übergeben werden.

Die Funktion `fill_stack()` realisiert den iterativen Floodfill.

Damit die Reihenfolge der ausgemalten Punkte dieselbe ist wie beim rekursiven Floodfill, müssen die Punkte im Vergleich zu den rekursiven Aufrufen in `fill()` in *umgekehrter Reihenfolge* auf den Stack gelegt werden. Dies liegt an der *First-In-First-Out*-Natur des Stack.

- (c) **Ersetzen Sie nun den Stack durch einen FIFO. Der Floodfill-Algorithmus sollte weiterhin funktionieren, nur daß die Punkte in einer anderen Reihenfolge „ausgemalt“ werden. Beschreiben Sie diesen Unterschied.**

Die Realisierung des FIFO erfolgt analog zu der des Stack, nur daß es nicht einen einzelnen `stack_pointer` gibt, sondern zwei Indizes `fifo_write` und `fifo_read`.

Für das ringförmige Inkrementieren von `fifo_write` und `fifo_read` verwenden die Funktionen `fifo_push()` und `fifo_pop()` die Modulo-Operation (`% FIFO_SIZE`).

Die Funktion `fill_fifo()` realisiert den iterativen Floodfill mit FIFO. Hierbei wurde nur überall „Stack“ durch „FIFO“ ersetzt und die `while`-Bedingung angepaßt: Die Bedingung, ob sich Daten im FIFO befinden, lautet nicht `stack_pointer > 0` wie beim Stack, sondern `fifo_write != fifo_read`.

Die Reihenfolge, in der die Punkte „ausgemalt“ werden, geht nun von nahen zu fernen Punkten. Damit eignet sich dieser Algorithmus dazu, unter den erreichbaren („auszumalenden“) Punkten den nächstgelegenen zu finden, um z. B. einen autonomen Roboter dorthin zu lenken. (Dieses Verfahren wurde im Wintersemester 2011/12 eingesetzt, um ein RP6-Roboterfahrzeug autonom durch ein Labyrinth steuern zu lassen.)

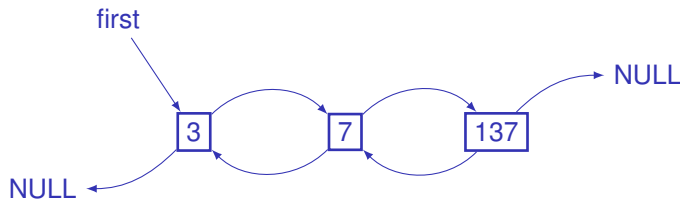
Im Gegensatz dazu geht beim iterativen Floodfill mit Stack – oder beim rekursiven Floodfill, der ja implizit ebenfalls einen Stack verwendet, nämlich den CPU-Stack – die Reihenfolge erst in eine Richtung (z. B. nach oben) und dann erst in andere Richtungen (z. B. noch ausstehende Punkte im unteren Bereich).

Aufgabe 3: Doppelt verkettete Liste

In der Vorlesung wurde ein Beispiel-Programm (`lists-5.c`) zur Verwaltung einfach verketteter Listen erstellt (an die Liste anhängen, in die Liste einfügen, Liste ausgeben usw.).

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

- (b) 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-3a.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.

Da das Beispielprogramm ([lists-5.c](#)) nicht mit dynamischem Speicher arbeitet, stellen wir dieses zunächst auf dynamischen Speicher um, z. B.:

```
node *element5 = malloc(sizeof (node));
```

Danach bezeichnet **element5** die Adresse der **struct**-Variablen; es wird also **element5** an die Funktionen übergeben und nicht **&element5** (die Adresse des Zeigers).

Um nun ein Element aus der Liste zu entfernen, benötigt man *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-3b.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.