Programmieren in C

Prof. Dr. Peter Thiemann
Hannes Saffrich
Sommersemester 2021

Universität Freiburg
Institut für Informatik

Übungsblatt 8

Abgabe: Montag, 21.06.2021, 9:00 Uhr morgens

In diesem Übungsblatt geht es um die Interaktion mit dem Dateisystem und rekursive Datenstrukturen.

Aufgabe 8.1 (Doubly Linked Lists; 7 Punkte)

In der Vorlesung haben Sie bereits eine Implementierung von einfach verketteten Listen kennengelernt. Diese werden durch eine Folge von Knoten repräsentiert, wobei jeder Knoten einen Zeiger auf den nachfolgenden Knoten enthält.

Doppelt verkettete Listen, wie sie auch aktuell in der Vorlesung *Algorithmen & Datenstrukturen* behandelt wurden, funktionieren ähnlich. Der Unterschied ist, dass bei doppelt verkettete Listen jeder Knoten zusätzlich einen Zeiger auf den vorherigen Knoten enthält. Dies erlaubt es an beiden Seiten der Liste in konstanter Zeit Elemente anzuhängen und zu entfernen. Dafür wird aber mehr Speicherplatz für die Knoten benötigt, die nun einen weiteren Zeiger enthalten.

In Abbildung 1 wird das Speicherlayout einer doppelt verketteten Liste dargestellt, welche die drei Elemente 10, 20 und 30 enthält.

Die `List`-Struktur enthält einen Zeiger auf den ersten und letzten Knoten und die Anzahl der Elemente die sich in der Liste befinden.

Die `Node`-Struktur enthält Zeiger auf den vorherigen und nachfolgenden Knoten und den Wert eines Elements. Gibt es keinen vorherigen oder nachfolgenden Knoten, so wird der `NULL`-Zeiger verwendet.

Eine leere Liste enthält keine Knoten und die Zeiger `first` und `last` sind beide `NULL`. Bei einer ein-elementigen Liste zeigen `first` und `last` auf den gleichen Knoten. Stellen Sie sicher, dass Ihre Implementierung auch mit diesen Fällen zurecht kommt.

In Ihrem Repository finden Sie die Dateien `list.h` und `list.c`, welche die Struk-

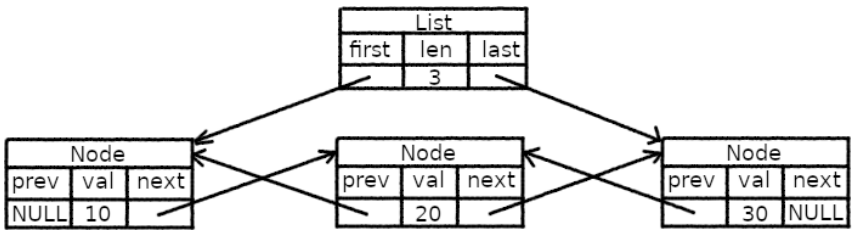

Abbildung 1: Doppelt verkettete Liste [10,20,30]

turen und Funktionsdeklarationen für doppelt verkettete Listen mit Elementen vom Typ `int` enthält. Implementieren Sie die Funktionen in `list.c` so dass Sie die Anforderungen aus `list.h` erfüllen. Die `struct`-Definitionen und Funktionstypen dürfen selbstverständlich nicht verändert werden, da dies die Aufgabenstellung verändern würde. Sie können dabei wie immer nach Design-by-Contract vorgehen, also z.B. annehmen, dass die Funktion zum Entfernen eines Knotens nur auf einer nicht-leeren Liste aufgerufen wird. Die Unittests sind dieses mal bereits vorgegeben und dienen Gleichzeitig als Beispiel wie man die Funktionen der `list.h` verwendet. Wenn `malloc` fehlschlägt, können Sie das Programm einfach mit `exit(1)` abbrechen.

Die Punkteverteilung ist dabei wie folgt:

- 0.5 Punkte für: `list_first`, `list_last`, `list_len`, `node_next`, `node_prev` und `node_val`.
- 1.5 Punkte für `list_new` und `list_free`.
- 2.5 Punkte für `list_push_back` und `list_push_front`.
- 2.5 Punkte für `list_pop_back` und `list_pop_front`

Magic Files. In dem Tutorialvideo zu *Linux, Shells und Terminals* aus Blatt 1, hatten wir gesehen, dass die Eingabe und Ausgabe von Text in Terminals eigentlich über spezielle Dateien geregelt wird, die mit dem Programm verbunden sind, das gerade im Terminal läuft.

Drückt man Tasten im Terminal, so landen die zugehörigen Tastendrücke als `char`s in der Eingabedatei, die dann z.B. über die `getchar`-Funktion vom Programm abgefragt werden können. Wenn das Programm z.B. mit `printf` eine Ausgabe erzeugt, so wird der auszugebende Text in eine Datei geschrieben, die vom Terminal beobachtet wird, welches den Text dann auf dem Bildschirm anzeigt.

Diese Dateien sind bei den meisten Programmiersprachen auch direkt als Dateiobjekte verfügbar. Bei C werden diese z.B. in der `stdio.h` als globale Variablen vom Typ `FILE*` bereitgestellt:

- die Eingabedatei ist `stdin` (“standard input”); und
- die Ausgabedateien sind
  - `stdout` (“standard output”); und
  - `stderr` (“standard error”).

Diese Dateien werden beim Programmstart automatisch geöffnet und zu Programmende wieder automatisch geschlossen.

Der Aufruf `printf("%d", 5)` ruft einfach `fprintf(stdout, "%d", 5)` auf.

Die beiden Ausgabedateien `stdout` und `stderr` sind standardmäßig beide mit der Terminalausgabe verbunden. Die Aufrufe

```
fprintf(stdout, "Hello from stdout!\n");
fprintf(stderr, "Hello from stderr!\n");
```

erzeugen also folgende Ausgabe:

```
Hello from stdout!
Hello from stderr!
```

Per Konvention gibt man Fehlerinformationen in `stderr` aus und normale Ausgaben in `stdout`. Dies ist nützlich, da man beim Starten eines Programms die Terminal Ein- und Ausgabe-Dateien durch andere Dateien austauschen kann, wie auch im Tutorialvideo gezeigt wurde. Zum Beispiel können wir ein Programm so aufrufen, dass es die Fehlermeldungen aus `stderr` in eine Log-Datei schreibt und die normalen Ausgaben von `stdout` weiter auf dem Bildschirm ausgibt:

```
./program 2> errors.log
```

Oder wir können beide Ausgaben in separaten Dateien speichern:

```
./program 1> normal_output.txt 2> errors.log
```

Im Tutorialvideo hatten wir hierzu z.B. `echo text > file` geschrieben. Dies ist eine Kurzform für `echo text 1> file`. Die Zahlen vor dem >-Symbol haben dabei die folgende Bedeutung: 0 steht für `stdin`, 1 steht für `stdout` und 2 steht für `stderr`.

Ähnlich kann man aber auch innerhalb des Programms entscheiden ob man lieber `stdout`, `stderr`, `stdin` verwendet oder eine andere Datei. Dies sieht man häufig bei Unix-Programmen, die optional einen Dateipfad als Kommandozeilenargument nehmen. Wird kein Dateipfad verwendet so liest das Programm stattdessen aus `stdin`. Zum Beispiel gibt `tail -n 2 file.txt` die letzten zwei Zeilen aus `file.txt` zurück. Die Angabe des Dateipfads ist aber optional und man kann `tail` auch wie folgt verwenden:

```
$ echo "line 1\nline 2\nline 3\nline 4" | tail -n 2
line 3
line 4
```

Da sowohl die `stdin` als auch andere Dateien den Typ `FILE*` haben, kann man so ein Programm relativ einfach schreiben:

```
FILE* input_file = stdin;
if (use_real_file) {
  input_file = fopen(file_path, "r");
  if (file == NULL) { ... }
}
/* The rest of the program doesn't have to care about whether we
   read from the terminal/stdin or from a custom file. */
```

Aufgabe 8.2 (Merging Files to CSV Tables; 7 Punkte)

CSV (Comma-separated values) ist ein simples text-basiertes Format zum Speichern von Tabellen. Das Format wird häufig verwendet um Tabellendaten zwischen verschiedenen Programmen auszutauschen.

Jede `.csv`-Datei repräsentiert genau eine Tabelle:

- Die Tabellenzeilen werden durch einen Zeilenumbruch voneinander getrennt.
- Die einzelnen Zellen einer Tabellenzeile werden durch Kommas voneinander getrennt.

Beispiel aus der Buchhaltung eines Supermarkts:

```
Product,Price in Cents,Amount
Milk,150,2000
Olive Oil,399,200
Awesome Sauce,599,5
```

In dieser Aufgabe sollen Sie ein Programm `table` schreiben, welches eine beliebige Anzahl von Dateipfaden als Kommandozeilenargumente nimmt, und den Inhalt dieser Dateien als eine Tabelle im CSV-Format ausgibt. Jede Datei beschreibt dabei eine Spalte der auszugebenden Tabelle und trennt deren Zellen durch Zeilenumbrüche.

Um die Aufgabe nicht unnötig komplex zu machen, nehmen wir an, dass die Zellen der Eingabedateien selbst keine Zeilenumbrüche oder Kommas enthalten.

Beispiel: Für die Dateien

Datei	`col_products.csv`	`col_prices.csv`	`col_amounts.csv`

Inhalt	`Product`
	`Milk`
	`Olive Oil`
	`Awesome Sauce`
		`Price in Cents`
		`150`
		`399`
		`599`
			`Amount`
			`2000`
			`200`
			`5`

soll das `table`-Programm die CSV-Tabelle des vorherigen Beispiels ausgeben:

```
$ ./table col_products.csv col_prices.csv col_amounts.csv
Product,Price in Cents,Amount
Milk,150,2000
Olive Oil,399,200
Awesome Sauce,599,5
```

In Ihrem Repository finden Sie die Daten `table.c`, `table_lib.c` und `table_lib.h`.

(a) (3 Punkte) Implementieren Sie die Funktion

```
FILE** open_files(char** file_paths, size_t num_file_paths,
                  FILE* error_file);
```

die ein Array von Dateipfaden `file_paths`, dessen Größe `num_file_paths` und eine Datei `error_file` als Argumente nimmt, versucht die durch `file_paths` beschriebenen Dateien zu öffnen und ein Array dieser Dateien zurückgibt. Für jede Datei, die nicht geöffnet werden kann, soll eine Fehlermeldung in die Datei `error_file` geschrieben werden. Gab es mindestens eine Datei die nicht geöffnet werden konnte, so soll `NULL` zurückgegeben werden. Vergessen Sie nicht im Fehlerfall alle bisher geöffneten Dateien zu schliesen und allozierten Speicher wieder freizugeben.

Beispiel 1:

```
char* file_paths[2] = { "col_products.csv", "col_prices.csv" };
FILE** files = open_files(file_paths, 2, stderr);
```

Beispiel 2:

```
char* file_paths[2] = { "i_dont_exist.csv", "i_also_dont_exist.csv" };
FILE** files = open_files(file_paths, 2, stderr);
```

In Beispiel 1 ist `files` ein Array der geöffneten Dateien und es wird keine Ausgabe in `stderr` erzeugt.

In Beispiel 2 ist `files` der `NULL`-Pointer und es wird folgende Ausgabe in `stderr` erzeugt:

```
ERROR: Failed to open file 'i_dont_exist.csv'.
ERROR: Failed to open file 'i_also_dont_exist.csv'.
```

(b) (0.5 Punkte) Implementieren Sie die Funktion

```
void close_files(FILE** files, size_t num_files);
```

welche die Dateien im Array `files` schließt und das Array freigibt.

Beispiel:

```
char* file_paths[2] = { "col_products.csv", "col_prices.csv" };
FILE** files = open_files(file_paths, 2, stdout);

/* do something with the files */

close_files(files, 2);
```

(c) (3 Punkte) Implementieren Sie die Funktion

```
void merge_columns(FILE** input_files, size_t num_input_files,
                   FILE* output_file);
```

die ein Array von Dateien `input_files` als Eingabe nimmt, diese als Tabellenspalten interpretiert und eine CSV-Tabelle der Spalten in `output_file` speichert (wie im Beispiel).

Haben die Dateien aus `input_files` unterschiedlich viele Zeilen, so soll sich das Program so verhalten, als würden die Dateien leere Zeilen enthalten. Hätte im Beispiel also die Datei `col_prices` eine Zeile weniger, dann würde folgende Tabelle ausgegeben werden:

```
$ ./table col_products.csv col_prices.csv col_amounts.csv
Product,Price in Cents,Amount
Milk,150,2000
Olive Oil,399,200
Awesome Sauce,,5
```

Um eine Zeile aus einer Datei zu lesen, können Sie die Funktion `fgetline` verwenden, die in `table_lib.c` bereitgestellt wird.

(d) (0.5 Punkte) Implementieren Sie die `main`-Funktion in `table.c`, sodass ein Programm entsteht, welches die Dateipfade per Kommandozeilenargumenten entgegennimmt, diese mit `open_files` öffnet, dann mit `merge_columns` verarbeitet und mit `close_files` wieder schließt. Verwenden Sie beim Aufruf von `open_files` die Datei `stderr` für die Fehlerausgabe und beim Aufruf von `merge_columns` die Datei `stdout` für die Ausgabe der CSV-Tabelle.

Aufgabe 8.3 (Erfahrungen; 2 Punkte)

Notieren Sie Ihre Erfahrungen mit diesem Übungsblatt in der Datei `erfahrungen.txt` (benötigter Zeitaufwand, Probleme, Bezug zur Vorlesung, Interessantes, etc.).

Der Zeitaufwand *muss* dabei in der ersten Zeile und in exakt dem folgenden Format notiert werden, da wir sonst nicht automatisiert eine Statistik erheben können:

```
Zeitaufwand: 3:30

<...Andere Erfahrungen...>
```

Die Angabe `3:30` steht hier für 3 Stunden und 30 Minuten.