

**Programmierung für Naturwissenschaften 2**  
**Sommersemester 2021**  
**Übungen zur Vorlesung: Ausgabe am 13.05.2021**

Am 13.05.2021 ist wegen des Feiertags keine Übung. Trotzdem steht dieses Übungsblatt bereits am 13.05. zur Verfügung, so dass Sie auch schon eine Woche vor dem nächsten Übungstermin am 20.05.2021 mit der Bearbeitung beginnen können. Dieser Übungstermin könnte dann genutzt werden, um gezielte Fragen zu stellen oder die eigene Lösung zu komplettieren.

**Aufgabe 6.1 (2 Punkte)**

Aus Zeitgründen können wir in diesem Semester in der Vorlesung das Thema Rekursion und Backtracking am Beispiel des  $n$ -Queens Problems (siehe `C_slides.pdf`, Abschnitt 14) nicht behandeln. Da das ein wichtiges Problem ist, wird es in zwei Peer-Teaching Aufgaben behandelt. Im ersten Teil geht es um die Frames 311-319, in denen das Problem vorgestellt und eine Lösung skizziert wird. Im zweiten Teil (Frames 320-330, nächstes Übungsblatt) geht es um die Implementierung.

In jeder der Kleingruppen muss sich der oder die Studierende mit dem lexikographisch größten Nachnamen anhand der oben genannten Frames 311-319 auf das Thema vorbereiten und das erworbene Wissen an die anderen Mitglieder der Kleingruppe weitergeben.

Falls jemand von diesen Studierenden nicht zur Übung erscheint, werden die übrigen Mitglieder der betroffenen Kleingruppe am Anfang der Übung auf die anderen Kleingruppen verteilt. Geben Sie bitte in einem solchen Fall auch die Namen der externen Mitglieder mit an, damit die Punkte zugeordnet werden können.

Diese Aufgabe soll am Anfang der Übung bearbeitet werden. Der oder die Studierende, die sich vorbereitet hat, kann ggf. bei Unklarheiten Fragen, zumindest zu den Grundbegriffen, beantworten oder Erläuterungen mündlich ergänzen. Die Implementierung wird bereits zur Verfügung gestellt, soll aber erst im zweiten Teil der Peer-Teaching Aufgabe behandelt werden.

Nach der Übung dokumentiert jede Kleingruppe in der Datei `bearbeitung.txt` das Vorgehen bei der Erarbeitung des Themas und ggf. noch bestehende Verständnisfragen oder Hinweise zu Unklarheiten in den Folien. Willkommen sind natürlich auch Bemerkungen zur Lehrform selbst.

**Aufgabe 6.2 (6 Punkte)** In naturwissenschaftlichen Experimenten werden oft viele Messwerte generiert und es kommt vor, dass man nur an den  $k$  kleinsten Messwerten interessiert ist, wobei  $k > 0$  ein Parameter ist. In einem solchen Fall ist es nicht sinnvoll, zunächst alle Messwerte zu generieren und zu speichern. Es ist besser, die experimentellen Messwerte direkt nach Ihrer Generierung zu prozessieren, so dass zu jedem Zeitpunkt nur maximal die  $k$  kleinsten der bisher generierten Messwerte gespeichert werden.

Um diese Idee umzusetzen, sollen Sie in dieser Aufgabe einen Datentyp `FSPriorityStore` implementieren, der Funktionen zur Verfügung stellt, um Paare von Messwerten `key` mit einem assoziierten Identifikator (`index`) zu speichern. Es soll höchstens eine feste Anzahl nach Priorität (entsprechend des Wertes von `key`) gespeichert werden. Daraus ergeben sich die Abkürzungen `FS` (für *fixed size*) und `Prio` (für Priorität). Höchste Priorität haben in unserem Fall die kleinsten Werte.

`FSPrioStore` soll als blickdichter Datentyp (opaque datatype) in der Datei `fs_prio_store.c` implementiert werden, analog zum Beispiel `Linked List` aus der Vorlesung. In der Header-Datei `fs_prio_store.h` finden Sie die Vorwärtsdeklarationen der Funktionen und vor jeder Deklaration wird die zu implementierende Funktionalität beschrieben. Ebenso finden Sie die Deklaration des Typs `KeyIndexPair` für die zu speichernden Werte.

Gehen Sie bei der Implementierung schrittweise vor, d.h. deklarieren Sie zunächst `struct FSPrioStore`, dann die Funktionen in der Reihenfolge, wie sie in der Header-Datei aufgeführt sind. Die `KeyIndexPair`-Werte sollen in einem Array, aufsteigend sortiert nach `key` gespeichert werden. Zu jedem Zeitpunkt sollen höchstens  $k + c$ -Werte gespeichert werden, wobei  $k$  die Kapazität ist und  $c$  eine Konstante, die nicht von Anzahl der Eingabewerte abhängt. In der Musterlösung ist z.B.  $c = 1$ .

In der Datei `fs_prio_store_mn.c` finden Sie ein Hauptprogramm, das die genannten Funktionen aufruft. Das Python-Skript `experiment.py` simuliert ein Experiment, dass Werte auf der Standard-Ausgabe liefert. Diese werden durch das kompilierte Programm eingelesen. Z.B. werden durch

```
./experiment.py 5000000 | ./fs_prio_store.x s 10
```

$5 \cdot 10^6$  Zeilen mit zufälligen Messwerten generiert, durch `fs_prio_store.x` werden sie eingelesen, prozessiert und die zehn kleinsten Messwerte werden ausgegeben. Durch `make test` verifizieren Sie die Korrektheit Ihrer Implementierung für verschiedene Testfälle.

Die folgenden Beispiele wurden mit der gleichen Nummerierung in der Datei `fs_prio_store_mn.c` als unit-Tests implementiert. Zur besseren Übersichtlichkeit geben wir hier nur ganzzahlige `key`-Werte an und ignorieren den Index. In der Implementierung sind die `key`-Werte vom Typ `double`. Im Folgenden ist mit dem Begriff *Wert* jeweils ein `key`-Wert gemeint. Sei die Kapazität  $k$  in allen Beispielen jeweils drei.

1. Annahme: Zwei Schlüssel sind gespeichert und der nächste generierte Wert ist 5. Dann wird der Wert 7 an die nächste Indexposition kopiert, und 5 wird an der frei gewordenen Stelle eingefügt (d.h. 7 wird überschrieben):

4	7	
---	---	--

 $\Rightarrow$ 

4	7	7
---	---	---

 $\Rightarrow$ 

4	5	7
---	---	---

2. Annahme: Zwei Schlüssel sind gespeichert und der nächste Wert ist 1. Dann wird 7 an die nächste Indexposition kopiert, 4 wird an die frei gewordene Position kopiert (also 7 überschrieben). An die frei gewordenen Position am Anfang des Arrays wird 1 eingefügt (also 4 überschrieben):

4	7	
---	---	--

 $\Rightarrow$ 

4	7	7
---	---	---

 $\Rightarrow$ 

4	4	7
---	---	---

 $\Rightarrow$ 

1	4	7
---	---	---

3. Annahme: Zwei Schlüssel sind gespeichert und der nächste Wert ist 8. Dann wird dieser Wert an der letzten Position eingefügt:

4	7	
---	---	--

 $\Rightarrow$ 

4	7	8
---	---	---

4. Annahme: Die drei Schlüssel 4, 7, 8 sind in dieser Reihenfolge gespeichert und der nächste Wert ist 9. Dieser Wert wird nicht gespeichert, d.h. das Array wird nicht verändert.
5. Annahme: Drei Schlüssel sind gespeichert und der nächste Wert ist 5. Dann wird 7 an die nächste Indexposition kopiert (also 8 überschrieben) und an der frei werdenden Position wird 5 eingefügt (also 7 überschrieben):

4	7	8
---	---	---

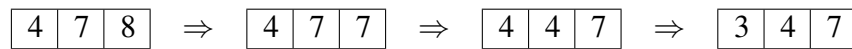
 $\Rightarrow$ 

4	7	7
---	---	---

 $\Rightarrow$ 

4	5	7
---	---	---

6. Annahme: Drei Schlüssel sind gespeichert und der nächste Wert ist 3. Dann wird 7 an die nächste Indexposition kopiert (also 8 überschrieben), 4 wird an die nächste Indexposition kopiert (also 7 überschrieben) und an der frei werdenden Position am Anfang des Arrays wird 3 eingefügt:



Punkteverteilung:

- Deklaration von **struct** `FSPrioStore`: 1 Punkt
- Deklaration der Funktion `fs_prio_store_add_smallest`: 2 Punkte
- Deklaration der Funktionen `fs_prio_store_new`, `fs_prio_store_delete`, `fs_prio_store_at`: jeweils 0.5 Punkt
- Deklaration der Funktionen `fs_prio_store_is_full` und `fs_prio_store_size` zusammen 0.5 Punkte.
- bestandene Tests: 1 Punkt

**Aufgabe 63** (7 Punkte) Es gibt viele Anwendungen, die zeilenweise auf den Inhalt einer Datei zugreifen, z.B. wenn Zeilen sortiert oder wenn mehrfach zusammenfassende Operationen auf den Spalten eines Dokumentes angewendet werden müssen. In dieser Aufgabe sollen Methoden für den effizienten Zugriff auf die Zeilen einer Datei implementiert werden.

Beispiel: Die Datei, die im Editor wie folgt aussieht

```
MNIDDKL
SVLQ
```

liegt im Speicher als Folge von Zahlen (d.h. 8-bit ASCII codes) 77, 78, 73, 68, 68, 75, 76, 10, 83, 86, 76, 81, 10 vor, die den String `MNIDDKL\nSVLQ\n` repräsentieren. Das Ziel ist, die Zeilen in `\0`-terminierte Strings zu konvertieren (ohne sie zu kopieren) und ein Array mit Zeigern auf diese Strings zu erzeugen. In diesem Beispiel sind das zwei Zeiger: Der eine zeigt auf den String `MNIDDKL`, der andere auf den String `SVLQ`.

`PfNLineStore` ist der zentrale Typ in dieser Aufgabe. Die Funktionen, die auf diesem Typ basieren, werden in der Datei `pfn_line_store.c` implementiert. Es gibt keine anderen Funktionen, die die Interna der Struktur kennen. Daher sagt man auch, das `PfNLineStore` ein blickdichter Datentyp (engl. opaque datatype) ist.

Die Datei `pfn_line_store.h` aus den Materialien enthält die Prototypen einiger zu implementierender Funktionen sowie die **typedef**-Deklaration für `PfNLineStore`. Wie üblich müssen Sie in der Datei `pfn_line_store.c` nach den notwendigen `include`-Anweisungen für die System-Headerdateien auch `pfn_line_store.h` inkludieren. Danach folgt diese **struct**-Definition

```
struct PfNLineStore
{
    size_t nextfree; /* number of lines */
    PfNLine *lines; /* array with references to lines */
    char separator;
};
```

Dabei ist `PfNLine` ein Typsynonym für `const char *`.

Ihre Aufgabe ist es nun, die folgenden Funktionen zu implementieren:

- `PfNLineStore *pfn_line_store_new(unsigned char *file_contents, size_t size, char sep)`

erzeugt aus dem Dateiinhalt mit `size` Bytes, referenziert durch `file_contents`, eine neue `PfNLineStore`-Struktur und liefert einen Zeiger auf die Struktur zurück.

Die Struktur muss mit `malloc` dynamisch allokiert werden. Der Parameter `sep` gibt das Zeichen an, das nach jeder Einheit in der Datei folgt. In den meisten Anwendungen ist `sep` das Zeichen `\n`, d.h. die Einheiten sind Zeilen. Zur Vereinfachung sprechen wir ab jetzt von Zeilen.

Am Ende der Funktion enthält `nextfree` in der genannten Struktur die Anzahl der Zeilen (2 im obigen Beispiel), und `lines` zeigt auf einen Speicherbereich mit `nextfree` vielen Einträgen des Basistyps `PfNLine`. Der  $i$ -te Eintrag zeigt auf den `\0`-terminierten String, der die  $i$ -te Zeile in der Datei repräsentiert (Nummerierung ab 0). Daher müssen im Speicherbereich, auf den `file_contents` zeigt, die Vorkommen von `\n` durch `\0` ersetzt werden. Es darf keine Kopie des Dateiinhalts erzeugt werden.

Aus Effizienzgründen sollen alle Werte in einer einzigen Iteration über `file_contents` berechnet werden. Die effizienteste Möglichkeit besteht darin die Vorkommen von `\n` mit der Funktion `memchr`<sup>1</sup> zu bestimmen. In der selben Iteration müssen auch die Zeiger auf die Zeilenanfänge in `lines` gesetzt werden und die genannte Zeichenersetzung erfolgen.

Beachten Sie, dass die Anzahl der Einträge in `lines` nicht vorher bekannt ist. Sie müssen daher die Größe des Speicherbereichs, auf den `lines` zeigt, sukzessive mit `realloc` vergrößern, so wie es in `C_slides.pdf` im Abschnitt über parsing and storing dates an einem Beispiel gezeigt wurde.

- `void pfn_line_store_delete(PfNLineStore *pfn_line_store)` gibt den Speicherbereich für eine `PfNLineStore`-Struktur, referenziert über den Zeiger `pfn_line_store*`, wieder frei, falls `pfn_line_store` nicht `NULL` ist.
- `size_t pfn_line_store_number(const PfNLineStore *pfn_line_store)`, liefert die Anzahl der Zeilen, die durch die `PfNLineStore`-Struktur repräsentiert werden.
- `PfNLine pfn_line_store_access(const PfNLineStore *pfn_line_store, size_t i)` liefert die  $i$ -te Zeile aus der `PfNLineStore`-Struktur.
- `char pfn_line_store_sep(const PfNLineStore *pfn_line_store)` liefert das Separator-Zeichen, entsprechend der die Zerlegung des Dateiinhalts erfolgte, zurück.
- `void pfn_line_store_show(const PfNLineStore *pfn_line_store)` schreibt alle Zeilen, die durch die übergebene `PfNLineStore`-Struktur repräsentiert werden, mit einem abschließenden `\n` auf die Standard-Ausgabe.

In den letzten vier Funktionen wird jeweils auf Komponenten der Struktur zugegriffen, auf die `pfn_line_store` zeigt. Daher muss mit `assert` überprüft werden, dass `pfn_line_store` nicht `NULL` ist.

In den Materialien finden Sie das Hauptprogramm `pfn_line_store_mn.c` mit einem Optionsparser, der die C-Bibliotheksfunktion `getopt` nutzt. Schauen Sie sich die Funktionen und `pfn_line_store_options_new` und `pfn_line_store_options_delete` genauer an und beantworten Sie einige Fragen, die Sie im Programmcode finden. Bitte verwenden Sie in Ihren Kom-

---

<sup>1</sup>siehe <https://man7.org/linux/man-pages/man3/rawmemchr.3.html>

mentaren keine Umlaute oder Sonderzeichen, da die Implementierung für alle Ihre Dateien im Materialverzeichnis dieser Aufgabe getestet wird und einige Standardwerkzeuge Probleme mit der Umlautkonvertierung haben.

In der Funktion `main` wird der Optionsparser aufgerufen, die durch den Benutzer spezifizierte Datei wird eingelesen und die entsprechende `PfnLineStore`-Struktur aufgebaut. Schließlich wird je nach Option für die Datei ein Wert bzw. der Dateinhalt in veränderter Form ausgegeben. Am Ende erfolgt die Freigabe des Speichers.

Durch `make` kompilieren Sie Ihr Programm. Durch `make test` verifizieren Sie die Korrektheit für einige Testdateien. Dabei wird vorausgesetzt, dass Standard-Programme wie `wc`, `rev` und `mktemp` vorhanden sind. Falls das nicht der Fall ist, müssen Sie diese noch installieren.

Punkte-Verteilung:

- für die Implementierung von `pfn_line_store_new`: 3.5 Punkte
- für die Implementierung aller anderen Funktionen: 1.5 Punkt insgesamt.
- für die akzeptable Beantwortung der Fragen: 2 Punkte

**Bitte die Lösungen zu diesen Aufgaben bis zum 25.06.2021 um 18:00 Uhr an [pfn2@zbh.uni-hamburg.de](mailto:pfn2@zbh.uni-hamburg.de) schicken.**