

**Programmierung für Naturwissenschaften 2**  
**Sommersemester 2021**  
**Übungen zur Vorlesung: Ausgabe am 03.06.2021**

Die Termine für die Modulabschlussprüfungen (Take Home Exam) liegen nun fest:

- 19.07.2021, 10:00-11:30
- 17.09.2021, 10:00-11:30

**Aufgabe 8.1 (2 Punkte)**

Aus Zeitgründen können wir in diesem Semester in der Vorlesung das Thema Binärbäume (siehe `C_slides.pdf`, Seite 352-369) nicht behandeln. Da das ein wichtiges Thema ist, wird es in dieser Peer Teaching Aufgabe behandelt.

In jeder Kleingruppe muss sich ein Studierender auf der Basis der oben genannten Seiten auf das Thema vorbereiten und das erworbene Wissen an die anderen Mitglieder der Kleingruppe weitergeben. Es sollte ein Studierender sein, der bisher nicht schon zweimal bei früheren Peer Teaching Aufgaben in diesem Semester diese Rolle übernommen hat. Falls es mehrere Studierende gibt, auf die das zutrifft, dann sprechen Sie sich rechtzeitig untereinander ab, wer die Vorbereitung übernimmt.

Diese Aufgabe soll am Anfang der Übung bearbeitet werden. Die Dateien mit den Programmcodes zu Binärbäumen finden Sie in den Materialien. Da die Erläuterungen zur Funktion `tree_add` und der entsprechende Programmcode nicht auf eine Folienseite passt, ist es sinnvoll, den Programmcode in einem Terminal zu öffnen und gleichzeitig die passenden Seiten im PDF anzuzeigen.

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 8.2 (3 Punkte)**

Schreiben Sie in einer Datei `randseqprob.c` eine C-Funktion

```
char *random_sequence_prob(const char *alphabet, const double *prob,  
                           size_t n)
```

die eine Zufallssequenz der Länge  $n$  über dem Alphabet `alphabet` in einer Laufzeit, die proportional zu  $n \log_2 k$  ist, berechnet.  $k$  ist die Größe des Alphabets, d.h. die Länge des `\0`-terminierten Strings `alphabet`. Dieser String enthält keine Duplikate.  $k$  ist wesentlich kleiner als  $n$ . `prob` ist ein Zeiger auf einen Speicherbereich mit genau  $k$  `double`-Werten, deren Summe 1 ist. Für das  $i$ -te Zeichen des Alphabets mit  $0 \leq i \leq k - 1$  und für alle Positionen der zu erzeugenden Sequenz ist die Wahrscheinlichkeit des Vorkommens dieses Zeichens `prob[i]`.

Der Rückgabewert der Funktion soll ein Zeiger auf einen Speicherbereich mit der Zufallssequenz sein. Diese Sequenz ist nicht `\0`-terminiert. Benutzen Sie die Funktion `drand48()` zum Generieren von Zufallszahlen und `strlen` zur Berechnung der Alphabetgröße. Sie können davon ausgehen,

dass `drand48()` konstante Zeit benötigt. Um die Initialisierung des Seeds für den Zufallsgenerator brauchen Sie sich nicht zu kümmern.

Beispiel: Falls `alphabet` auf den String "ACGT" zeigt und `prob` auf das Array mit den Werten 0.1, 0.2, 0.3, 0.4, dann liefert `random_sequence_prob(alphabet, prob, n)` eine Zufallssequenz der Länge  $n$  in der `a` etwa  $0.1n$  mal vorkommt, `c` etwa  $0.2n$  mal vorkommt, `g` etwa  $0.3n$  mal vorkommt, und `t` etwa  $0.4n$  mal vorkommt.

Vergessen Sie nicht, in Ihre C-Datei mit dem oben genannten Namen die Datei `randseqprob.h` zu inkludieren.

In den Materialien finden Sie ein Hauptprogramm, in dem Ihre Funktion aufgerufen und getestet wird. Durch `make test` wird die Korrektheit Ihrer Implementierung für einige Testfälle verifiziert.

Punkteverteilung:

- Implementierung der Funktion `random_sequence_prob`: 2 Punkte
- bestandene Tests: 1 Punkt

### Aufgabe 8.3 (7 Punkte)

In dieser Aufgabe geht es um die Implementierung noch fehlender Teile eines Programms `sort_simple_mn.x`, das einen Teil der Funktionalität des Kommandozeilen-Werkzeugs `sort` implementiert.

Beim Aufruf erhält das Programm den Namen einer Datei sowie Optionen, durch die die Art der Sortierung der Zeilen und ihre Reihenfolge in der Ausgabe beeinflusst werden. Der Inhalt der übergebenen Datei wird natürlich nicht verändert.

Im Standardfall werden die Zeilen der Datei in aufsteigender Reihenfolge lexikographisch (also entsprechend der Ordnung wie im Telefonbuch) sortiert. Optional kann auch numerisch sortiert werden (Option `-n`), was bei Zeilen mit Zahlenwerten sinnvoll ist.

Für beide Formen der Sortierung kann die Reihenfolge der Ausgabe umgekehrt werden (Option `-r`), also absteigend sortiert werden.

Diese Aufgabe besteht aus mehreren Teilen.

## Teil 1

Benennen Sie die Datei `pfn_line_store_template.c` um in `pfn_line_store.c`. Diese Datei erweitern Sie um eine Funktion

```
void pfn_line_store_sort(PfNLineStore *pfn_line_store, CompareFunc compar)
```

Diese soll mit Hilfe der C-Bibliotheksfunktion `qsort` die Zeilen sortieren, die in `pfn_line_store` gespeichert sind, und zwar unter Verwendung des Funktionszeigers `compar`, der auf die Vergleichsfunktion zeigt.

Dabei ist der Typ `CompareFunc` in `pfn_line_store.h` wie folgt deklariert:

```
typedef int (*CompareFunc)(const void *, const void *);
```

## Teil 2

Der zweite Teil besteht darin, in der Datei `sort_simple.c` eine Funktion

```
void sort_simple(PfNLineStore *line_store, bool numerical_order,  
                bool reverse_order)
```

zu implementieren, die entsprechend der Parameter `numerical_order` und `reverse_order` die passende Vergleichsfunktion auswählt und diese dann beim Aufruf von `pfn_line_store_sort` verwendet, um die Zeilen aus `line_store` zu sortieren. Selbstverständlich müssen Sie vorher die vier Vergleichsfunktionen, nennen wir sie `lex_cmp`, `num_cmp`, `lex_reverse_cmp`, und `num_reverse_cmp`, implementieren. Dabei steht `lex` für die lexikographische Ordnung, `num` für die numerische Ordnung und `reverse` für die Umkehrung der Reihenfolge. Es ist sinnvoll, zunächst die ersten beiden Funktionen zu implementieren und sich dann zu überlegen, wie man diese für die beiden `_reverse`-Funktionen wiederverwenden kann.

Für die lexikographische Ordnung sollen Sie die Funktion `strcmp` verwenden. Da `PfNLine` der Basistyp des `lines`-Arrays in `PfNLineStore` ist, müssen Sie die `void`-Zeiger der `_cmp`-Funktion entsprechend casten, analog zur Funktion `double_cmp` aus der Vorlesung.

Für die numerische Sortierung müssen Sie testen, ob der entsprechende String einen numerischen Wert darstellt. Wenn das für beide Strings gilt, müssen Sie den Vergleich auf der Basis des numerischen Wertes durchführen. Verwenden Sie Variablen vom Typ `double` für die numerischen Werte. Es gelten die folgenden besonderen Regeln, wenn bei der numerischen Sortierung einer oder beide zu vergleichende Strings keine numerischen Werte darstellen:

- Falls beide Strings keine numerischen Werte darstellen, wird mit `strcmp` verglichen.
- Falls einer der beiden Strings einen numerischen Wert darstellt und der andere den leeren String, dann wird der leere String beim Vergleich wie der numerische Wert 0.0 behandelt.
- Ein String mit dem numerischen Wert 0.0 ist kleiner als jeder nicht leere String, der keinen numerischen Wert darstellt.
- Ein String mit einem numerischen Wert  $\neq 0.0$  ist größer als jeder nicht leere String, der keinen numerischen Wert darstellt.

## Teil 3

In den Materialien finden Sie eine Datei `sort_simple_mn_template.c`, die Sie bitte in `sort_simple_mn.c` umbenennen. Diese Datei enthält die `main`-Funktion. Sie müssen noch einen Optionsparser implementieren. Dieser besteht aus der Deklaration eines `struct`-Typs `Options` und den Funktionen `usage`, `options_new` und `option_delete`. Die auszugebenden `Usage`-Zeilen stehen in der Datei `usage.txt`. Die Bezeichner für die Komponenten der Struktur und die jeweiligen Typen ergeben sich aus der `main`-Funktion. Bei der Entwicklung des Optionsparsers orientieren Sie sich bitte am Optionsparser aus der Datei `pfn_line_store_mn.c` (siehe Aufgabe 6.3).

In der Funktion `main` wird der Optionsparser aufgerufen, die durch den Benutzer spezifizierten Dateien werden eingelesen und die entsprechende `PfNLineStore`-Struktur aufgebaut. Schließlich folgt der Aufruf von `sort_simple` und die Freigabe des Speichers.

Durch `make` können Sie Ihr Programm kompilieren. Durch `make test` verifizieren Sie die

Korrektheit für einige Test-Dateien.

Nachdem Sie verifiziert haben, dass `make test` funktioniert, rufen Sie bitte `make test_large` auf. Beschreiben Sie in einem Satz, auf welche Art von Daten die Programme `sort_simple_mn.x` und `sort` angewendet werden. Notieren Sie die Laufzeiten der Programme (in Sekunden) und beschreiben Sie mögliche Gründe für die unterschiedlichen Laufzeiten. Es geht hier nicht um eine asymptotische Analyse (diese Form der Analyse kennen die Studierenden, die bereits an InfB-AD teilgenommen haben), sondern um konkrete Laufzeiten bei Ausführung des Programms auf Ihrem Rechner.

Punkte-Verteilung:

- 1 Punkt für `pfn_line_store_sort`,
- 1 Punkt insgesamt für die beiden Funktionen zur lexikographischen Sortierung,
- 2 Punkte für die Funktionen zur numerischen Sortierung,
- 1 Punkt für den Optionsparser,
- 1 Punkt für funktionierende Tests,
- 1 Punkt für eine nachvollziehbare Auswertung und Erklärungen.

**Bitte die Lösungen zu diesen Aufgaben bis zum 08.06.2021 um 18:00 Uhr an `pfn2@zbh.uni-hamburg.de` schicken.**