Aufgabe 7.0 Dynamisch wachsende Listen

Ziel dieser Aufgabe ist es, ein Programm zu schreiben, das beliebig viele Wörter aus einer Textdatei einliest, sortiert und auf die Konsole schreibt. Da wir aber nicht von vornherein wissen können, wie viele Wörter später einmal verarbeitet werden sollen, benötigen wir eine dynamische Datenstruktur, deren Größe sich zur Laufzeit anpassen kann.

Zu diesem Zweck wird in der Datei `stringlist.c` folgende Datenstruktur definiert:

```
struct stringlist
{
    char * * data;
    int capacity;
    int used;
};

enum { INITIAL_CAPACITY = 5 };

void stringlist__init(struct stringlist * self)
{
    self->data = malloc(INITIAL_CAPACITY * sizeof(char *));
    self->capacity = INITIAL_CAPACITY;
    self->used = 0;
}
```

Der Grundgedanke besteht darin, zunächst ein Array mit Platz für 5 `char*` auf dem Heap zu reservieren (die 5 ist dabei völlig beliebig gewählt). Die Tatsache, dass Platz für 5 `char*` vorhanden ist, merken wir uns in der Variable `capacity`. In der Variable `used` wird gespeichert, wie viele `char*` bisher genutzt werden, zu Beginn natürlich 0.

Falls sich später herausstellen sollte, dass 5 zu knapp bemessen war, werden die Daten einfach vor dem Einfügen des sechsten Eintrags in ein doppelt so großes Array umkopiert. Falls auch das nicht ausreichen sollte, wird die Kapazität auf 20 verdoppelt und so weiter.

Schritt 1: In der `main`-Funktion wird beispielhaft ein Wörterbuch `words` erzeugt, befüllt und gelöscht. Mache Dir die Struktur einer Stringliste klar, indem Du eine Zeichnung der Variable `words` anfertigst, und zwar nach dem dritten Aufruf der `add`-Funktion. Welche Werte haben dann `capacity` und `used`? Worauf zeigt `data`? Worauf zeigt `*data`?

Man beachte, dass in `stringlist__init` per `malloc` dynamischer Speicher angefordert wird. Dieser Speicher muss analog in einer „Aufräumfunktion“ `stringlist__done` wieder freigegeben werden. Klienten einer Stringliste müssen daran denken, die Funktion `stringlist__done` auch aufzurufen, wenn sie die Liste nicht mehr benötigen.

Schritt 2: Implementiere die „Aufräumfunktion“ `stringlist__done`.

Man beachte, dass `stringlist__add` nicht einfach nur den Zeiger `p` kopiert, sondern genügend Speicherplatz auf dem Heap reserviert (`malloc`) und anschließend die komplette Zeichenkette kopiert (`strcpy`). Diese Kopien „gehören“ also der Liste und müssen entsprechend in `stringlist__done` wieder freigegeben werden!

Schritt 3: Gib die kopierten Zeichenketten in `stringlist__done` wieder frei.

Wenn man versucht, mehr als fünf Wörter einzufügen, wird das Programm momentan abgebrochen, weil `stringlist__double_capacity` noch nicht implementiert ist.

Schritt 4: Mache eine Zeichnung einer (vollen) Stringliste mit fünf Wörtern und eine weitere Zeichnung einer Stringliste mit verdoppelter Kapazität und sechs Wörtern.

Schritt 5: Implementiere `stringlist__double_capacity`. In den meisten Sprachen würde man so vorgehen, dass man zunächst ein größeres Array anlegt, dann die Elemente kopiert und zuletzt den Zeiger auf das neue Array umbiegt (altes Array löschen nicht vergessen!). Geht das in C nicht einfacher, dank einer speziellen Bibliotheksfunktion?

Schritt 6: Lege manuell eine Textdatei an, die pro Zeile ein Wort enthält, und lies diese Wörter in das Programm ein.

Schritt 7: Sortiere das Wörterbuch mit `qsort` und einer geeigneten Vergleichsfunktion.

Schritt 8: Schaffst Du es, Duplikate aus dem Wörterbuch zu entfernen? Wenn das Wörterbuch erst einmal sortiert ist, kann man die Duplikate relativ einfach aufspüren.

Aufgabe 7.1 Wortsuche

Erweitere die stringlist um folgende Funktion:

```
int stringlist__contains(const struct stringlist * self, const
char * s)
```

Diese Funktion soll 1 liefern, wenn s in der Liste enthalten ist, ansonsten 0.

Aufgabe 7.2 Wortgrenzen finden

Schreibe zwei Funktionen:

```
char * first_letter(char * s)

char * first_non_letter(char * s)
```

Die erste Funktion soll, beginnend bei s, das erste Zeichen suchen, das ein Buchstabe ist. Die zweite Funktion soll, beginnend bei s, das erste Zeichen suchen, das *kein* Buchstabe ist:

```
char * p = first_letter("*** dies ist ein test ***");
//                          ^   ^

char * q = first_non_letter(p);
```

Im obigen Beispiel soll `p` auf das ‚d' zeigen und `q` auf das Leerzeichen hinter „dies".

Aufgabe 7.3 Liste als Menge

Lies alle Zeilen aus der Datei mobydick.txt ein, trenne die Zeile in Wörter auf und füge jedes Wort in eine stringlist ein, sofern es noch nicht in dieser stringlist enthalten ist.

Dies sollte ein paar Sekunden dauern, und am Ende sollten 18261 Wörter in der Liste enthalten sein.

Aufgabe 7.4 Kleinbuchstaben

Manche Wörter kommen immer noch doppelt in der Liste vor, und zwar mit unterschiedlicher Groß- bzw. Kleinschreibung. Konvertiere jedes eingelesene Wort in Kleinbuchstaben. Dann sollten am Ende nur noch 16690 Wörter übrig bleiben.

Aufgabe 7.5 Hashverfahren

Füge zwei weitere Dateien hash.c und hash.h hinzu und implementiere eine Hashtabelle. Die einfachste Variante wäre ein Array fixer Größe mit stringlist als Elementtyp:

```
struct hashtable
{
     struct stringlist table[1000];
};
```

Implementiere passende Funktionen `hashtable__add` und `hashtable__size`.

Jetzt sollte das Bestimmen der Anzahl unterschiedlicher Wörter in Moby Dick deutlich schneller sein. Experimentiere mit verschiedenen Größen für das Array. Hat dies einen messbaren Einfluss auf die Laufzeit?

Aufgabe 7.6 Binäre Suchbäume

Füge zwei weitere Dateien tree.c und tree.h hinzu und implementiere einen Binären Suchbaum für Wörter. Folgende Datentypen könnte man dafür verwenden:

```
struct tree_node
{
     char * word;
     struct tree_node * before;
     struct tree_node * after;
};

struct tree
{
     struct tree_node * root;
};
```

Implementiere passende Funktionen `tree__add` und `tree__size`.