Heaps und Hashes

Allgemeine Anmerkungen zur Aufgabenstellung:

- Beantworten Sie die Kontrollfragen in `TUWEL` bevor Sie mit der Übung beginnen.
- Legen Sie für `Teil A` der Angabe ein neues `Code::Blocks`-Projekt an und erweitern Sie dieses schrittweise für die darauffolgenden Punkte.
- Stellen Sie während der Übung laufend sicher, dass Ihr Programm kompilierbar ist und richtig funktioniert.
- Die Abgabe erfolgt durch Hochladen Ihres vollständigen `Code::Blocks`-Projektordners als zip-Datei (`Matrikel-Nr_UE4.zip`) in `TUWEL`. Anschließend melden Sie sich für das Abgabegespräch in `TUWEL` an. Dieses erfolgt über die Telekonferenz-App `Zoom` (www.zoom.us). Stellen Sie sicher, dass diese App rechtzeitig vor Beginn des Abgabegespräches auf Ihrem PC installiert ist und dass Sie Ihr aktuelles Übungsprogramm in `Code::Blocks` für die Abgabe geöffnet haben. Testen Sie bitte auch die korrekte Funktionalität Ihrer Videokamera, da Sie zu Beginn Ihres Abgabegespräches Ihren Studentenausweis herzeigen müssen und auch während des Gesprächs eine aktive Videoverbindung erforderlich ist.
- Beachten Sie, dass Upload und Terminauswahl für das Abgabegespräch erst möglich sind, wenn Sie 80% der Kontrollfragen richtig beantwortet haben. Bis **14.12.2021 23:59** müssen diese Fragen positiv beantwortet, Ihre Ausarbeitung hochgeladen und ein Termin für das Abgabegespräch ausgewählt werden.
- Inhaltliche Fragen stellen Sie bitte in der Fragestunde oder im zugehörigen Forum in `TUWEL`.
- Für eine positive Beurteilung Ihrer Abgabe **muss** diese kompilierbar sein, Ihr Programm fehlerfrei terminieren und die in der Angabe angeführten Funktionen mit den vorgegebenen Funktionsköpfen enthalten. Des Weiteren müssen die Funktionen hinreichend getestet sein. Sie müssen in der Lage sein, Ihr Programm beim Abgabegespräch zu erklären!

Folgende Hilfsmittel stehen Ihnen in `TUWEL` zur Verfügung:

- Alle bisherigen Referenzbeispiele sowie Vorlesungsfolien
- Interaktive Jupyter-Notebooks auf `prog1.iue.tuwien.ac.at`
- Das Buch zur LVA "Programmieren in C" (Robert Klima, Siegfried Selberherr)
- Kurzanleitung für die Entwicklungsumgebung `Code::Blocks`

Hinweis: Es wird erwartet, dass alle Abgaben zu den Übungen **eigenständig** erarbeitet werden und wir weisen Sie darauf hin, dass alle Abgaben in einem standardisierten Verfahren auf Plagiate überprüft werden. Bei Plagiatsvorfällen entfällt das Abgabegespräch und es werden keine Punkte für die entsprechende Abgabe vergeben.

Stoffgebiet
Sie benötigen für diese Übung zusätzlich zum gesamten Lehrinhalt der Lehrveranstaltung Programmieren 1 folgenden Stoff:

- Rekursive Funktionen (Kapitel 18)
- Datenstrukturen (Heap, Hash, Zirkularbuffer, Baum) (Kapitel 19)
- Speicher reservieren und freigeben (Kapitel 20)

Einleitung
In der letzten Übung haben Sie eine Wiedergabeliste (Playlist) in Form eines binären Suchbaumes erstellt. Neben der Baumstruktur gibt es noch weitere fundamentale Datenstrukturen wie etwa den *Heap* oder den *Hash*. In dieser Übung werden Sie sich nun mit diesen beiden Strukturen vertraut machen und deren jeweilige Vor- und Nachteile erkennen. Dazu realisieren Sie die Wiedergabeliste aus der letzten Übung sowohl als Heap als auch in Form eines Hashs.

DUMA Bibliothek
Die unsachgemäße Verwendung von dynamischen Speicherbereichen kann zu schwer auffindbaren Programmfehlern zur Laufzeit führen. Um die Fehlersuche zu vereinfachen und die korrekte dynamische Speicherverwaltung sicherzustellen, verwenden Sie die `DUMA` Bibliothek. Mit Hilfe dieser Bibliothek bekommen Sie eine Rückmeldung über die Verwendung von dynamisch alloziertem Speicher. Diese Bibliothek ersetzt die Systemfunktionen `malloc()`, `calloc()`, `realloc()` und `free()` durch eigene Varianten, die bei falscher Verwendung unmittelbar eine Fehlermeldung zurück liefern. Für die Übung haben wir `DUMA` so verändert, dass über eine Umgebungsvariable (`DUMA_MALLOC_ERROR`, siehe unten) die Fehlerwahrscheinlichkeit für die Speicherallozierung eingestellt werden kann.

Um die `DUMA`-Bibliothek zu installieren und in Ihrem Projekt zu verwenden, folgen Sie den Anweisungen in der auf `TUWEL` zur Verfügung gestellten `DUMA`-Anleitung.

Um ein gescheitertes Anfordern von Speicherbereichen zu simulieren, starten Sie unter Linux Ihr Programm in der Konsole mit dem Befehl

```
DUMA_MALLOC_ERROR=50 ./<Programmname>
```

Unter Windows verwenden Sie `cmd` und starten Ihr Programm wie folgt

```
set DUMA_MALLOC_ERROR=50
<Programmname>.exe
```

Die Umgebungsvariable `DUMA_MALLOC_ERROR` gibt dabei die Fehlerwahrscheinlichkeit in Prozent an. In diesem Beispiel würde also im Mittel jede zweite Speicheranforderung fehlschlagen.

Hinweis: Beachten Sie, dass während des Abgabegesprächs auch die korrekte Verwendung der dynamischen Speicherverwaltung überprüft wird. Achten Sie daher u.a. auf Speicherlecks und korrekte Freigaben. Ihr Programm muss bei einer gescheiterten Speicher-Allozierung eine definierte Fehlerbehandlung durchführen und darf unter **keinen Umständen** abstürzen!

Teil A

- ✘ Fügen Sie die `DUMA` Bibliothek zu Ihrem Projekt hinzu bevor Sie mit der eigentlichen Implementierung beginnen. Die genauen Anweisungen zur Installation und Verwendung dieser Bibliothek finden Sie in der zugehörigen `DUMA`-Anleitung auf `TUWEL`.

- ✘ Definieren Sie zunächst einen neuen Verbunddatentyp, der einen Eintrag der Wiedergabeliste repräsentiert:

```
typedef struct entry_s {
        char songtitle[256], interpreter[256];
} entry_t;
```

- ✘ Definieren Sie weiters den Verbunddatentyp

```
typedef struct heap_s {
        entry_t *entries;
        long size;
        long next_free;
} heap_t;
```

  `heap_t` repräsentiert dabei einen Heap mit der Größe `size`. `next_free` soll der Index des ersten freien Elements im Heap sein. Beachten Sie, dass zur Vereinfachung der Heap-Algorithmen der Eintrag mit Index 0 freigelassen wird. `songtitle` wird als Schlüssel für die Heap-Bedingung verwendet.

- ✘ Implementieren Sie die Funktion

```
heap_t *heap_create(long size);
```

  die Speicher für den Heap selbst sowie seine `size` Einträge anlegt und den Heap anschließend richtig initialisiert. Geben Sie dann einen Zeiger auf den erzeugten Heap zurück.

- ✘ Implementieren Sie die Funktion

```
long heap_insert(heap_t *heap, char songtitle[], char interpreter[]);
```

  die einen neuen Eintrag an der ersten freien Stelle im Heap einfügt und anschließend die Heap-Bedingung wiederherstellt. Sollte beim Einfügen ein Fehler auftreten, so geben Sie 0 zurück, andernfalls 1.

Teil B

- ✘ Implementieren Sie die Funktion

```
void heap_print(heap_t *heap);
```

  die den kompletten Heap ausgibt. Sie können die Formatierung beliebig wählen, achten Sie allerdings darauf, dass die Baumstruktur des Heaps sichtbar wird.

- ✘ Implementieren Sie die Funktion

```
void heap_free(heap_t *heap);
```

  die den Heap löscht. Beachten Sie dabei, dass Sie alle Elemente wieder freigeben, die Sie in der Funktion `heap_create()` initialisiert haben.

✘ Schreiben Sie nun ein zugehöriges Hauptprogramm das zuerst einen noch leeren Heap der Größe 100 anlegt und danach ein Menü der folgenden Form anbietet:

```
0. Programm beenden
1. Gesamten Heap loeschen
2. Neuen Eintrag im Heap ablegen
3. Heap ausgeben
```

Teil C

✘ Implementieren Sie die Funktion

```
long heap_search(heap_t *heap, char songtitle[], char interpreter[]);
```

die den Heap nach dem Eintrag mit den übergebenen Parametern durchsucht. Geben Sie den Index des gefundenen Elements zurück bzw. 0, wenn kein Element gefunden wurde. Nutzen Sie die Heap-Bedingung um die Suche zu optimieren!

✘ Erweitern Sie Ihr Hauptprogramm um den Menüpunkt

```
4. Eintrag im Heap suchen
```

Teil D

✘ Implementieren Sie die Funktion

```
long heap_delete_first(heap_t *heap);
```

welche den Eintrag an der ersten Stelle des Heaps ausgibt und danach löscht. Stellen Sie anschließend die Heap-Bedingung wieder her. Geben Sie 0 zurück, wenn nach dem Löschen der Heap leer ist, ansonsten 1.

✘ Erweitern Sie Ihr Hauptprogramm um den Menüpunkt

```
5. Ersten Eintrag aus dem Heap löschen
```

✘ Überlegen Sie sich, in welcher Reihenfolge die Einträge bei wiederholtem Aufruf von `heap_delete_first` entfernt werden.

Teil E

✘ Um die Wiedergabeliste nun in Form eines Hash-Containers abzubilden, definieren Sie die folgenden beiden Verbunddatentypen:

```
typedef struct hasharray_s {
        entry_t *entries;
        long num_entries;
} hasharray_t;
```

Das `hasharray` sammelt alle Einträge, die den gleichen Hashwert besitzen im Feld `entries` und dient damit der Kollisionsvermeidung. `num_entries` bezeichnet dabei die Anzahl der Einträge in diesem Feld. Beachten Sie, dass der verfügbare Speicher in `*entries` beim Hinzufügen eines neuen Eintrags mittels `realloc()` vergrößert werden muss.

```
typedef struct hash_s {
        hasharray_t *hasharrays;
        long hashsize;
} hash_t;
```

Der Datentyp `hash_t` repräsentiert die eigentliche Datenstruktur des Hashes mit der Größe `hashsize`.

✘ Zur Generierung eines Hashwerts für einen Eintrag verwenden Sie folgende Funktion:

```
long hash_key(char songtitle[], char interpreter[], long hashsize)
{
        unsigned long index = 0, i;
        for (i = 0; i < strlen(songtitle); ++i)
                index = 64 * index + (long)(songtitle[i]);
        for (i = 0; i < strlen(interpreter); ++i)
                index = 64 * index + (long)(interpreter[i]);
        return index % hashsize;
}
```

✘ Implementieren Sie die Funktion

```
hash_t *hash_create(long hashsize);
```

die einen Hash mit einer geeigneten Größe `hashsize` erstellt und richtig initialisiert. Setzen Sie zu Beginn `*entries=NULL` und `num_entries=0` für jedes Element von `hasharrays`. Beachten Sie, dass `hashsize` eine Primzahl sein sollte, um die Kollisionswahrscheinlichkeit zu verringern.

✘ Implementieren Sie die Funktion

```
void hash_insert(hash_t *hash, char songtitle[], char interpreter[]);
```

welche einen neuen Eintrag im Hash einfügt. Im Falle einer Kollision vergrößern Sie das Feld `entries` des betroffenen `hasharray` und speichern den neuen Eintrag am Ende des Feldes `entries` ab.

Teil F

✘ Implementieren Sie die Funktion

```
void hash_print(hash_t *hash);
```

die den übergebenen `hash` ausgibt. Geben Sie dabei für jeden gespeicherten Eintrag den jeweiligen Interpreten und Songtitel aus.

✘ Implementieren Sie die Funktion

```
void hash_free(hash_t *hash);
```

die den übergebenen `hash` löscht. Beachten Sie, dass Sie alle Elemente, die Sie mit der Funktion `hash_create()` erstellt haben, hier freigeben müssen.

✘ Um einen vorher erstellten Heap in einen Hash zu portieren, schreiben Sie die Funktion

```
hash_t *convert_heap_to_hash(heap_t *heap);
```

Dabei sollen Kopien aller Heap-Einträge in einen neuen Hash eingefügt werden und ein Zeiger auf diesen neuen Hash zurückgegeben werden.

✘ Erweitern Sie Ihr Hauptprogramm um die Menüpunkte

```
6. Hash aus Heap erzeugen
7. Gesamten Hash löschen
8. Neuen Eintrag im Hash ablegen
9. Hash ausgeben
```

Teil G

✘ Implementieren Sie die Funktion

```
entry_t *hash_search(hash_t *hash, char songtitle[], char interpreter[]);
```

welche einen bestimmten Eintrag im `hash` sucht und einen Verweis darauf zurückgibt. Berechnen Sie dazu den Hash-Wert des zu suchenden Eintrags mit Hilfe der Funktion `hash_key()`, um zuerst das richtige `hasharray` zu finden. Anschließend durchsuchen Sie dieses Feld, bis der richtige Eintrag gefunden wird.

✘ Implementieren Sie die Funktion

```
long hash_delete_entry(hash_t *hash, char songtitle[],
                       char interpreter[]);
```

die einen bestimmten Eintrag im `hash` sucht und aus diesem entfernt. Sollte kein passender Eintrag gefunden werden oder ein Fehler auftreten, geben Sie 0 zurück, ansonsten 1.

Hinweis:
Um die entstandene Lücke in `entries` wieder zu schließen, verwenden Sie die Funktion `memcpy()`, bevor Sie das Feld `entries` verkleinern.

✘ Erweitern Sie Ihr Hauptprogramm um die Menüpunkte

```
10. Eintrag im Hash suchen
11. Eintrag aus dem Hash löschen
```

Teil H

✘ Untersuchen Sie, wie der Rechenaufwand für das Befüllen und Durchsuchen Ihrer beiden Datenstrukturen mit der Anzahl der Einträge skaliert. Gehen Sie dabei wie folgt vor:

- Erstellen Sie einen leeren Heap/Hash.
- Befüllen Sie die Datenstruktur mit jeweils N zufällig generierten Einträgen. Messen Sie dabei die insgesamt benötigte Zeit.
- Suchen Sie nach allen Einträgen in der Datenstruktur und messen Sie die benötigte Zeit sowie die Anzahl der benötigten Vergleichsoperationen.
- Berechnen Sie daraus die mittlere Rechenzeit und die mittlere Anzahl von Vergleichen für eine Suchoperation.
- Wiederholen Sie diese Schritte für $N \in \{100, 200, 400, 800, \ldots, 51200\}$.
- Erkennen Sie einen funktionalen Zusammenhang?
- Wie ändert sich die Performance des Hashes mit `hashsize`?

Entwickeln Sie dazu die Funktion

```
void benchmark()
```

die diese Schritte durchführt und fügen Sie einen entsprechenden Menüpunkt im Hauptmenü ein. Geben Sie nach Beendigung des Benchmarks alles wieder richtig frei.

Hinweis: Wie bereits bei der letzten Übung können Sie das Beispielprogramm `benchmark.zip` als Vorlage verwenden. Die verstrichene Zeit kann mit der C-Funktion `clock()` aus `time.h` gemessen werden. Weitere Informationen entnehmen Sie dem zur Verfügung gestellten Beispielprogramm.