Bäume

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_UE3.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 **30.11.2021 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 dieser Übung werden Sie Ihr erworbenes Wissen über dynamische Speicher aus der 1. Übungseinheit und der Vorlesung verwenden, um eine Wiedergabeliste (Playlist) in Form eines binären Suchbaumes zu erstellen. Im Zuge dieser Übung lernen Sie daher komplexere Datenstrukturen aufzubauen und anzuwenden.

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`.

✘ In dieser Übung werden Sie eine Wiedergabeliste (Playlist) in Form eines *binären Suchbaums* erstellen. Für die Baumknoten implementieren Sie dazu folgenden Verbunddatentyp:

```
typedef struct node_s {
  char songtitle[256], interpreter[256];
  struct node_s *parent, *left, *right;
} node_t;
```

Jeder Knoten entspricht dabei einem Eintrag in der Wiedergabeliste und speichert die Attribute `songtitle` und `interpreter`. Wie bei einem Baumknoten üblich, werden Verweise auf den linken (`left`) und rechten (`right`) Nachfolger gespeichert, zusätzlich enthält jeder Knoten auch einen Verweis auf seinen Vorgänger (`parent`). Das Attribut `songtitle` soll als Suchschlüssel für den Baum verwendet werden.

✘ Implementieren Sie die Funktion

```
node_t *create_node(char songtitle[], char interpreter[]);
```

die einen neuen Knoten anhand der übergebenen Parameter erstellt und einen Zeiger auf diesen zurückgibt. Achten Sie dabei insbesondere auf die richtige Initialisierung der Verweise!

✘ Implementieren Sie die Funktion

```
node_t *insert_node(node_t *rootnode, char songtitle[],
                    char interpreter[]);
```

die mit Hilfe der Funktion `create_node()` einen neuen Knoten erstellt und an der richtigen Stelle in den Baum einfügt. Vergleichen Sie dazu die Songtitel mittels `strcmp()`. Falls der einzufügende Knoten einen kleineren oder gleichen Schlüssel hat, soll er im linken Zweig untergebracht werden, ansonsten im rechten. Der Rückgabewert soll ein Zeiger auf die Baumwurzel sein, wenn die Einordnung des neuen Knoten erfolgreich war und `NULL` im Fehlerfall. Wenn `rootnode==NULL` soll nur ein neuer Knoten mit `create_node()` erzeugt werden und ein Zeiger darauf zurückgegeben werden.

Teil B

✘ Implementieren Sie die Funktion

```
void print_tree(node_t *rootnode);
```

die den gesamten Baum auf der Konsole ausgeben soll. Sie können die Formatierung beliebig wählen, achten Sie allerdings darauf, dass die Struktur des Baumes sichtbar wird.

✘ Implementieren Sie die Funktion

```
void destroy_tree(node_t *rootnode);
```

die den Speicher für den gesamten Baum wieder freigibt.

✘ Schreiben Sie nun ein zugehöriges Hauptprogramm das zuerst einen noch leeren Baum anlegt (ein `NULL` Zeiger vom Typ `*node_t`) und danach ein Menü der folgenden Form anbietet:

```
0. Programm beenden
1. Gesamten Baum loeschen
2. Neuen Knoten von der Tastatur einlesen und im Baum ablegen
3. Baum ausgeben
```

Teil C

✘ Implementieren Sie die Funktion

```
long count_nodes(node_t *rootnode);
```

welche die Gesamtanzahl der Knoten im Baum bestimmt und zurückgibt.

✘ Implementieren Sie die Funktion

```
node_t *search_node(node_t *rootnode, char songtitle[]);
```

die nach dem ersten Knoten mit übereinstimmenden `songtitle` im Baum sucht und einen Zeiger darauf zurückgibt. Nutzen Sie dabei die Eigenschaften des Suchbaums, um die Suche effizient zu gestalten. Sollte der Knoten nicht gefunden werden, geben Sie `NULL` zurück.

✘ Erweitern Sie Ihr Hauptprogramm um die folgenden Menüpunkte:

```
4. Anzahl der Knoten bestimmen
5. Ersten Knoten mit Songtitle suchen
```

Teil D

✘ Implementieren Sie die Funktion

```
node_t **search_all(node_t *rootnode, char interpreter[], long *found);
```

die nach *allen* Knoten mit einem bestimmten `interpreter` im Baum sucht. Der Rückgabewert soll dann ein dynamisch alloziertes Feld von Zeigern auf die übereinstimmenden Knoten sein. Sollte kein passender Knoten gefunden werden, so wird `NULL` zurückgegeben. Nach der Funktionsausführung soll `found` die Anzahl der gefundenen Knoten beinhalten. Vergessen Sie nicht, das dynamisch dynamisch allozierte Feld nach Gebrauch im Hauptprogramm wieder freizugeben!

✘ Erweitern Sie Ihr Hauptprogramm um den Menüpunkt

```
6. Alle Knoten mit Interpreter suchen
```

Teil E

✘ Implementieren Sie die Funktion

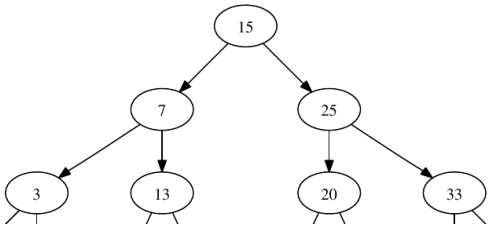
```
long max_depth(node_t *rootnode);
```

welche die maximale Tiefe des übergebenen Baums bestimmt. Die maximale Tiefe ist dabei die Anzahl der Knoten zwischen der Wurzel und dem am weitesten entfernten Blattknoten.

✘ Schreiben Sie eine Funktion

```
void print_level(node_t *rootnode, long level);
```

welche die Knoten des Baumes auf einer Ebene `level` von links nach rechts ausgibt. Für den Baum in der obigen Abbildung würde die Ausgabe für `level==2` bspw. `3 13 20 33` lauten.



✘ Erweitern Sie Ihr Hauptprogramm um die folgenden Menüpunkte:

```
7. Maximale Tiefe des Baumes ausgeben
8. Ebene x des Baumes ausgeben
```

Teil F

✘ Implementieren Sie die Funktion

```
node_t *delete_node(node_t *rootnode, char songtitle[], char interpreter[]);
```

welche nach dem ersten Knoten mit `songtitle` und `interpreter` sucht und diesen aus dem Baum löscht. Es soll die (eventuell) neue Wurzel des Baumes zurückgegeben werden. Achten Sie darauf, dass nach dem Löschen die Eigenschaften des binären Suchbaums erhalten bleiben.

✘ Erweitern Sie Ihr Hauptprogramm um den Menüpunkt

```
9. Bestimmten Eintrag löschen
```

Teil G

✘ Schreiben Sie eine Hilfsfunktion

```
double average_comparisons(node_t *rootnode)
```

welche für jeden Knoten im Baum die Anzahl der notwendigen Vergleichsoperationen ermittelt um diesen Knoten ausgehend von der Baumwurzel zu finden. Geben Sie dann die mittlere Anzahl zurück.

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

- Erstellen Sie einen leeren Baum.
- Befüllen Sie Ihren Baum mit N zufällig generierten Einträgen. Messen Sie dabei die insgesamt benötigte Zeit.
- Suchen Sie nach allen Einträgen im Baum 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?
  Ist die Laufzeit proportional zu $N, N^2, \log(N), \sqrt{N}$ etc.?

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: In TUWEL finden Sie das Beispielprogramm `benchmark.zip`, welches die obigen Schritte für ein einfaches Feld ausführt. Sie können dieses Programm als Vorlage für Ihre `benchmark` Funktion verwenden. Um zufällige Strings zum Befüllen der Datenstruktur zu erzeugen, können Sie die Hilfsfunktionen

```
char *get_random_string(size_t length);

char **generate_random_string_array(size_t array_size, size_t string_length);
```

im Modul `container` verwenden. Die verstrichene Zeit kann mit der C-Funktion `clock()` aus `time.h` gemessen werden. Weitere Informationen entnehmen Sie dem zur Verfügung gestellten Beispielprogramm.

Teil H

✘ Implementieren Sie die Funktion

```
long check_if_order_OK(node_t *rootnode);
```

die die Ordnung innerhalb des Baumes überprüft. Testen Sie dabei für jeden Knoten, ob sein linker Nachfolger auch wirklich einen kleineren (oder gleichen) Schlüssel besitzt. Verfahren Sie analog mit dem rechten Nachfolger.

✘ Implementieren Sie die Funktion

```
node_t *rebalance(node_t *rootnode);
```

die den Baum ausbalancieren soll und einen Zeiger auf die neue Wurzel des Baumes zurückgibt.

Hinweis: Ein Baum ist balanciert, wenn sich die Tiefe der einzelnen Blattknoten um maximal 1 unterscheidet. Sie können den Baum balancieren in dem Sie die Knoten in einem Feld zwischenspeichern und dann in der richtigen Reihenfolge wieder in einen neuen Baum einfügen.

✘ Erweitern Sie Ihr Hauptprogramm um den Menüpunkt

```
11. Baum ausbalancieren
```

✘ Erstellen Sie mit Ihrem Programm zunächst einen zu einer Liste entarteten Baum, in dem Sie bspw. Knoten mit Songtiteln von “A” bis “Z” einfügen. Überprüfen Sie mit Hilfe von `average_comparisons()` wie viele Vergleichsoperationen durchschnittlich zum Finden eines Knotens notwendig sind. Wie ändert sich die Suchperformance nachdem Sie den entarteten Baum balanciert haben?