Human-Computer Interaction Group

Leibniz Universität Hannover

Programmieren 1 – WS 2020/21

Prof. Dr. Michael Rohs, Tim Dünte, M.Sc.

Übungsblatt 10

Alle Übungen (bis auf die erste) müssen in Zweiergruppen bearbeitet werden. Beide Gruppenmitglieder müssen die Lösung der Zweiergruppe einzeln abgeben. Die Namen beider Gruppenmitglieder müssen sowohl in der PDF Abgabe als auch als Kommentar in jeglichen Quelltextabgaben genannt werden. Wenn Sie für Übungsblatt 1 noch keinen Gruppenpartner haben, geben Sie alleine ab und nutzen Sie das erste Tutorium dazu, mit Hilfe des Tutors einen Partner zu finden. Plagiate führen zum Ausschluss von der Veranstaltung.

Abgabe bis Donnerstag den 07.01. um 23:59 Uhr über https://assignments.hci.uni-hannover.de/WiSe2020/Programmieren1. Die Abgabe muss aus einer einzelnen zip-Datei bestehen, die den Quellcode, ein pdf bei Freitextaufgaben und alle weiteren nötigen Dateien (z.B. Eingabedaten oder Makefiles) enthält. Lösen Sie Umlaute in Dateinamen bitte auf.

Die Dokumentation der Prog1lib finden Sie unter: https://hci.uni-hannover.de/files/prog1lib/index.html

Aufgabe 1: Operationen auf Listen

Das Template für diese Aufgabe ist wolf_goat_cabbage.c. In dieser Aufgabe sollen verschiedene Funktionen auf einer einfach verketteten Liste mit Elementen vom Typ String implementiert werden. Diese Funktionen werden in Aufgabe 2 benötigt. Die Struktur für die Listenknoten ist vorgegeben. Die Benutzung der Listen und Listenfunktionen der Programmieren I Library ist für diese Aufgabe nicht erlaubt. Die Nutzung von `xmalloc` statt `malloc` und `xcalloc` statt `calloc` ist verpflichtend, um die Überprüfung auf korrekte Freigabe des Speichers zu ermöglichen.

a) Implementieren Sie die Funktion `free_list`, die eine List mitsamt Inhalt freigibt. Beachten Sie, dass die Liste Besitzer („owner") der Listenelemente (`value`) ist und alle Listenelemente dynamisch allokiert sind. Diese müssen also auch freigegeben werden.

b) Implementieren Sie die Funktion `bool test_equal_lists(int line, Node* list1, Node* list2)`, die überprüft, ob die beiden Listen inhaltlich gleich sind, ob sie also die gleichen Elemente haben. Die Funktion soll genau dann `true` zurückgeben, wenn das der Fall ist. Außerdem soll sie das Ergebnis in folgender Form ausgeben:
   - `Line 78: The lists are equal.`
   - `Line 82: The values at node 1 differ: second <-> hello.`
   - `Line 86: list1 is shorter than list2.`
   - `Line 90: list1 is longer than list2.`

   Eine Testfunktion mit Beispielaufrufen existiert bereits (`test_equal_lists_test`). Diese muss nicht verändert werden. Die Funktion illustriert auch, wie Listen mit `new_node` generiert werden.

c) Implementieren Sie mindestens drei Beispielaufrufe in der Funktion `length_list_test`. Verwendetn Sie `test_equal_i(actual, expected);`

d) Implementieren Sie die Funktion `int index_list(Node* list, String s)`. Diese soll den Index von `s` in `list` zurückgeben. Wenn `s` nicht in `list` vorkommt, soll -1 zurückgegeben werden. Sichern Sie die Funktion mit einer Precondition ab, falls der String `s == NULL` ist. Implementieren Sie außerdem mindestens drei Beispielaufrufe in der zugehörigen Testfunktion.

e) Implementieren Sie die Funktion `Node* remove_list(Node* list, int index)`. Diese soll den Knoten an Position index löschen (und den Speicher freigeben) und die resultierende Liste zurückgeben. Sichern Sie die Funktion mit entsprechenden Preconditions ab. Nutzen Sie das Makro `ensure_code` um eine Postcondition zu formulieren (bspw. neue Listenlänge == alte Listenlänge -1). Implementieren Sie außerdem mindestens drei Beispielaufrufe in der zugehörigen Testfunktion.

Aufgabe 2: Der Wolf, die Ziege und der Kohlkopf

Das Template für diese Aufgabe ist ebenfalls wolf_goat_cabbage.c. Entfernen Sie die Kommentare vor `make_puzzle` und `play_puzzle` (am Ende der `main`-Funktion). In dieser Aufgabe soll ein Spiel implementiert werden, in dem ein Bauer einen Wolf, eine Ziege und einen Kohlkopf in einem Boot über einen Fluss transportieren möchte. Zunächst sind Bauer, Wolf, Ziege, Kohlkopf und Boot am linken Ufer des Flusses. Leider hat das Boot nur einen freien Platz (neben dem Bauern, der das Boot rudert). Der Bauer darf aber den Wolf und die Ziege nicht alleine lassen, weil sonst der Wolf die Ziege frisst. Er darf auch die Ziege mit dem Kohlkopf nicht alleine lassen, weil sonst die Ziege den Kohlkopf frisst. Ist der Bauer in Reichweite, besteht keine Gefahr. Das Spiel ist dann erfolgreich gelöst, wenn Wolf, Ziege und Kohlkopf sicher am rechten Ufer angelangt sind.

a) Implementieren Sie die Funktion `print_puzzle`, die den aktuellen Spielzustand ausgibt. Der Anfangszustand soll z.B. mit folgender Zeile dargestellt werden:

```
[Wolf Ziege Kohl][]      []
```

Dabei sind linkes Ufer, Boot und rechtes Ufer durch Listen repräsentiert. Im Ausgangszustand sind alle Objekte am linken Ufer, das Boot liegt am linken Ufer und ist leer und das rechte Ufer ist ebenfalls leer. Wenn das Boot leer nach rechts fährt, ändert sich die Ausgabe in:

```
[Wolf Ziege Kohl]      [][]
```

b) Implementieren Sie die Funktion `finish_puzzle`, die allen dynamisch allokierten Speicher freigibt und das Programm beendet. Zum Beenden kann `exit(0);` verwendet werden.

c) Implementieren Sie die Funktion `evaluate_puzzle`, die aktuelle Situation analysiert und ausgibt. Die Funktion soll das Programm mit einer Meldung beenden, wenn die Aufgabe gelöst wurde bzw. die Situation kritisch ist.

d) Implementieren Sie die Funktion `play_puzzle`, die den Anfangszustand des Spiels ausgibt, Eingaben des Spielers entgegen nimmt und die Listen entsprechend manipuliert. Ist beispielsweise das Boot leer und auf der linken Seite, dann soll nach Eingabe von `Wolf` der Wolf vom linken Ufer genommen und in das Boot geladen werden. Die Eingabe `l` bewegt das Boot nach links, die Eingabe `r` nach rechts. Die Eingabe von `q` (für quit) soll das Spiel beenden. Nach jeder Eingabe soll diese Funktion die neue Situation evaluieren und ausgeben. Es folgt ein (nicht erfolgreicher) Beispielablauf:

[Wolf Ziege Kohl][] []	→ Anfangszustand, alle am linken Ufer
Wolf	← Eingabe: Spieler lädt den Wolf ins Boot
[Ziege Kohl][Wolf] []	→ Wolf im Boot
r	← Eingabe: r für nach rechts übersetzen
[Ziege Kohl] [Wolf][]	→ Boot mit Wolf ist am rechten Ufer
Die Ziege frisst den Kohl.	→ Ziege ist am linken Ufer allein mit dem Kohl und frisst ihn

Beispielablauf: Hinüberbringen der Ziege:

[Wolf Ziege Kohl][] []	→ Anfangszustand, alle am linken Ufer
Ziege	← Eingabe: Spieler lädt die Ziege ins Boot
[Wolf Kohl][Ziege] []	→ Ziege im Boot
r	← Eingabe: r für nach rechts übersetzen
[Wolf Kohl] [Ziege][]	→ Boot mit Ziege ist am rechten Ufer
Ziege	← Eingabe: Spieler lädt die Ziege aus dem Boot
[Wolf Kohl] [][Ziege]	→ Ziege am rechten Ufer, Boot leer

Hinweise: Verwenden Sie `String s_input(100)`, um einen dynamisch allokierten String von der Standardeingabe einzulesen. Verwenden Sie `bool s_equals(String s, String t)`, um zu prüfen, ob zwei Strings gleich sind. Nutzen Sie die in Aufgabe 1 definierten Listenoperationen. Geben Sie dynamisch allokierten Speicher wieder frei. Sie dürfen, falls notwendig, beliebige Hilfsfunktionen implementieren.

Hinweis: Es gibt mehrere Varianten solcher Flussüberquerungsrätsel. Siehe: https://de.wikipedia.org/wiki/Flussüberquerungsrätsel

Aufgabe 3: Bleep Censor

Das Template für diese Aufgabe ist die Datei `bleep_censor.c`. In dieser Aufgabe sollen Sie unschöne Wörter aus einer Zeichenkette durch entsprechend zensierte Versionen ersetzen. Beispiel: „Du bloeder Esel " soll zu „Du bl***** Esel " verändert werden. Die Wörter, die ersetzt werden sollen, sind in einer unsortierten Zeichenkette vorgegeben und durch Leerzeichen voneinander getrennt. Sie können selber Wörter hinzufügen. Diese Wörter sollen in einen binären Baum sortiert eingefügt werden um eine schnelle Ersetzung zu gewährleisten. Dort wird nicht neuer Speicher allokiert, sondern es wird eine Tokenstruktur erstellt mit einem Zeiger auf den Start des Wortes und einen Zeiger auf das Ende des Wortes (vgl. Präsenzübung 9).

a) Implementieren Sie die Funktoin `void print_token(Token* t, bool censored)`, die ein Token ausgibt. Wenn `censored == true` ist, soll nur der erste Buchstabe des Tokens ausgegeben werden und danach entsprechend der Länge des Wortes Sterne `'*'` folgen.
b) Implementieren Sie die Funktion `int compare_token(Token* t1, Token* t2)`, die zwei Token vergleicht und prüft ob ein Token lexikographisch vor das andere gehört. Schauen Sie sich auch die Testfälle an. Wenn `t1` vor `t2` gehört, soll -1 zurückgegeben

werden. Wenn `t1` gleich `t2` ist, soll 0 zurückgegeben werden. Wenn `t1` nach `t2` gehört soll 1 zurückgegeben werden. Groß- und Kleinschreibung soll bei dem Vergleich keine Rolle spielen.

c) Implementieren Sie die Funktion `void insert_in_tree(TreeNode** tree, Token* token)`, die einen Token in den Baum sortiert einfügt. Nutzen Sie für das sortierte einfügen die Funktion `compare_token` aus b). Ist das Ergebnis negativ muss es im linken Teilbaum eingefügt werden, ist es 0 muss nichts geschehen und ist es größer als 0 muss `token` im rechten Teilbaum eingefügt werden.

d) Implementieren Sie die Funktion `TreeNode* create_bleep_tree(char* beep_words)`, die aus einer Zeichenkette einen Baum erstellt. Nutzen Sie die Funktion aus c).

e) Lesen Sie von der Standardeingabe eine Zeile mit der Funktion `get_line()` aus der Programmieren 1 Bibliothek ein. Geben Sie die eingelesene Zeichenkette wieder aus und zensieren Sie mithilfe des Baums. Sie müssen nur Wörter bzw. Sätze verarbeiten können, in denen die Wörter durch Leerzeichen getrennt sind und die mit einem Leerzeichen enden. Beispiel: "Der Affe sitzt im Baum ". Wird zu: "Der A*** sitzt im Baum"

f) Geben Sie den gesamten allokierten Speicher wieder frei. Implementieren Sie dazu geeignete Funktionen.

Aufgabe 4: Wunschbaum (1 Bonuspunkt)

Das Template für diese Aufgabe ist `wish_tree.c`. Der Weihnachtsmann investiert dieses Jahr in seine IT und möchte von handgeschriebenen Wunschzetteln und Papierlisten mit Geschenken auf einen modernen binären Wunschbaum umstellen. Glücklicherweise wurden bereits in diesem Jahr moderne Scanner mit Texterkennung angeschafft, die auch mühelos Kinderhandschrift erkennen können. D.h. Sie bekommen die Wunschzettel bereits digital in einer Textdatei. Ihre Aufgabe besteht nun darin, die Datei mit den Wunschzetteln einzulesen und die Wünsche in einen binären Baum einzutragen. Der binäre Baum soll nach dem Wunschtext sortiert sein. Jeder Knoten im Baum, soll sowohl den Wunsch, als auch die Häufigkeit speichern wie oft der Wunsch insgesamt von Kindern gewünscht wurde. Jeder Knoten soll zudem eine Liste mit Kindernamen enthalten, die den Wunsch geäußert haben.

a) Die Elfen aus der IT-Abteilung haben bereits eine Baumstruktur `TreeNode` erstellt. Prüfen Sie, ob diese Ihre Anforderungen erfüllen. Machen Sie sich auch mit dem weiteren Template-Code vertraut.

b) Implementieren Sie eine Struktur `Element`, in der der Wunschtext, die Häufigkeit sowie eine Liste von Kindern gespeichert wird, die diesen Wunsch haben. Erstellen Sie auch eine Konstruktorfunktion `Element* new_element(char* wish, char* child)`, die alle Elemente übergeben bekommt und dann eine dynamisch allokierte Struktur `Element` zurückgibt. Die Struktur `Node` kann Ihnen helfen.

c) Implementieren Sie eine rekursive Funktion `TreeNode* add_wish(TreeNode* tree, char* wish, char* child)`. Diese soll den Binärbaum nach dem Wunschtext durchsuchen und entweder einen vorhandenen Knoten aktualisieren oder einen neuen Knoten geordnet einfügen. Nutzen Sie für die Sortierung innerhalb des Binärbaums die Funktion strcmp, die Ihnen 0 zurückgibt, wenn der Wunschtext des Elements gleich dem gesuchten Wunschtext ist.

d) Schreiben Sie eine rekursive Funktion `void print_tree_as_list(TreeNode* tree)`, die den Baum auf der Konsole im Listenformat lexikographisch sortiert nach

dem Wunschtext wie nachfolgend dargestellt ausgibt.

Wunsch	Anzahl	Kinder
Barbie Meerjungfrau	13	Finn, Paul, Theresa, Noah, Juna, Leon, Romy, Luise, Niklas, Elias, Jonas, Emely, Konstantin
Barbie und Ken Geschenkset mit Huendchen	12	Finn, Julia, Luis, Theresa, Juna, Romy, David, Luise, Amelie, Ben, Konstantin, Lukas
Haarkreide	9	Paul, Theresa, Juna, Philipp, David, Luise, Ben, Amy, Ella
Hasbro Kroko Doc	9	Julia, Theresa, Juna, Luise, Niklas, Finja, Jonas, Ben, Amy
Kidisecrets	5	Finn, Theresa, Luise, Emely, Lukas
Lego Das Disney Schloss	10	Finn, Luis, Theresa, Philipp, Frieda, Elias, Jonas, Emely, Ella, Lukas
Lego Millenium Falcon	14	Paul, Eva, Luis, Theresa, Noah, Romy, Philipp, Oskar, Amelie, Elias, Emely, Konstantin, Ella, Lucy
Mattel Exklusiv Hot Wheels Doppel-Looping Wettrennen	10	Finn, Luis, Juna, Philipp, Frieda, Luise, Jonas, Emely, Ben, Lukas
My Fairy Garden	13	Finn, Eva, Luis, Noah, Juna, Philipp, Luise, Amelie, Elias, Jonas, Ben, Amy, Lukas
MyToys Holzeisenbahn	12	Finn, Paul, Juna, Romy, Philipp, David, Luise, Elias, Jonas, Amy, Ella, Benjamin
MyToys Puppenhaus mit Garten und Moebel	12	Finn, Julia, Luis, Theresa, Philipp, Katharina, Luise, Amelie, Finja, Ben, Amy, Benjamin
MyToys-Collection Puppenwagen Trendy	11	Luis, Theresa, Philipp, Oskar, David, Luise, Niklas, Emely, Ben, Amy, Lukas
Nintendo Switch	9	Katharina, David, Amelie, Finja, Ben, Amy, Ella, Lukas, Benjamin
PLAYMOBIL Polizei-Einsatzwagen	8	Juna, Romy, Philipp, David, Luise, Ben, Konstantin, Ella
Piano Matte	9	Finn, Theresa, Juna, Philipp, David, Luise, Jonas, Ben, Konstantin
Play-Doh Verrueckte Haufen	14	Finn, Luis, Theresa, Juna, Romy, David, Niklas, Amelie, Elias, Finja, Jonas, Konstantin, Lukas, Lucy
Pop up Pirate	6	Juna, Philipp, Luise, Elias, Jonas, Konstantin
Schleich Mobile Tieraerztin	12	Finn, Theresa, Juna, Romy, Philipp, Oskar, David, Finja, Ben, Amy, Konstantin, Ella
Spin Master Monster Jam - Grave Digger	14	Finn, Paul, Theresa, Juna, Leon, Philipp, David, Luise, Elias, Emely, Ben, Amy, Konstantin, Lukas
Tonies 30 Lieblings-Kinderlieder	9	Finn, Paul, Luis, Juna, Philipp, Elias, Emely, Ben, Ella

e) Der Weihnachtsmann hat leider nicht genug Elfen um alle Geschenke zu produzieren. Als Heuristik, nimmt er an, dass es ausreichend ist die 11 häufigsten Geschenke herzustellen, sodass alle Kinder (hoffentlich) versorgt sind. Erstellen Sie als erstes eine Liste, in der die Wünsche absteigend nach Ihrer Häufigkeit sortiert sind. Implementieren Sie dafür die Funktion `ElementNode* insert_ordered_by_count(ElementNode* result, TreeNode* tree)`, die Ihnen eine Liste aus Strukturen vom Typ `ElementNode` erstellt, diese soll die Strukturen vom Typ Element aus dem Baum `tree` absteigend sortiert nach der Wunschhäufigkeit enthalten. Die Funktion soll für die Strukturen vom Typ Element keinen neuen Speicher allokieren.

f) Schreiben Sie in der main Funktion eine Routine um zu überprüfen, ob alle 29 Kinder Geschenke bekommen. Geben Sie das Resultat auf der Konsole aus. Nutzen Sie die Funktion aus e) und die bestehenden Funktionen und Strukturen im Template.

g) Implementieren Sie Funktionen, um den gesamten Speicher wieder freizugeben. Am Ende soll es keine Memory Leaks geben.