Human-Computer Interaction Group

Leibniz Universität Hannover

Programmieren 1 – WS 2020/21

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

Übungsblatt 13

Dieses Übungsblatt kann auf freiwilliger Basis bearbeitet werden. Das Übungsblatt kann nicht über das Abgabesystem abgegeben werden. Es erfolgt keine Korrektur durch Ihren Tutor.

Aufgabe 1: Fußballstatistik

In dieser Aufgabe sollen Ergebnisse von Fußballspielen erfasst werden und für jedes Team eine Statistik geführt werden. Die Template-Datei für diese Aufgabe ist soccer.c. Für jedes Team werden in `struct Result` die Anzahl der bisher gespielten Spiele, die Anzahl selbst geschossener Tore, die Anzahl Gegentore und der Punktestand erfasst. Die Daten liegen in der Textdatei soccer_results.txt mit folgendem Format:

```
Bayern München          // Heim-Team
Bayer Leverkusen        // Auswärts-Team
3                       // Tore Heim-Team
1                       // Tore Auswärts-Team
```

Jeweils vier Zeilen gehören also zu einer Begegnung.

a) Lesen Sie die Datei zeilenweise ein und aktualisieren Sie jeweils die Map m. Diese bildet für jedes Team den Teamnamen (`String`) auf dessen Ergebnis (`Result*`) ab. Das Einlesen einer Zeile kann über `String s = s_input(100);` erfolgen. Die Konvertierung von `String` nach `int` kann über `int i = i_of_s(s);` erfolgen. Wenn für ein Team noch kein Resultat in der Map vorliegt, muss dieses erstellt werden. Ein verlorenes Spiel zählt 0, ein unentschiedenes 1 und ein gewonnenes Spiel zählt 3 Punkte.

b) Implementieren Sie in map.c die Funktion `int size_map(Map* m)`, um die Anzahl der Einträge in einer Map zu ermitteln.

c) Implementieren Sie in map.c die Funktion `Entry* entries_map(Map* m)`, um ein dynamisch allokiertes Array der Einträge der Map zu erzeugen. Die keys und values selbst sollen dabei nicht dupliziert werden.

d) Optional: Sortieren Sie die Teams nach Punktzahl und Tordifferenz, um eine Tabelle zu erhalten und geben Sie diese aus.

Aufgabe 2: Iteratoren für Baumstrukturen

Zu dieser Aufgabe gehören folgende Dateien: iterator.{c|h}, btree.{c|h}, tree.{c|h}, dyn_list.{c|h}, dyn_queue.{c|h}und dyn_stack.{c|h}. In dieser Aufgabe sollen Sie verschiedene Iteratoren für zwei verschiedene Datenstrukturen schreiben. Die Datenstrukturen sind 1. ein Binärbaum (vgl. btree.h) und 2. ein Baum mit beliebig vielen Nachfolgern (vgl. tree.h). Die Iteratoren sollen den Baum in „level order" (ebenenweise), sowie in „pre order" durchlaufen können (vgl. https://en.wikipedia.org/wiki/Tree_traversal). Die Methoden und Strukturen in dyn_queue.{c|h} und dyn_stack.{c|h} können für die Aufgabe hilfreich sein.

a) Machen Sie sich mit der Struktur Iterator in iterator.h vertraut. Diese speichert einen Zustand in `state` und bietet 3 Funktionen an: `has_next` gibt true zurück, wenn noch ein weiteres Element vorhanden ist. `next_element` gibt das nächste Element zurück und free gibt den Inhalt von `state` frei. Diese 3 Funktionen werden von den Funktionen `has_next(Iterator* iterator)`, `next_element(Iterator* iterator)` und `free_iterator(Iterator* iterator)` aufgerufen, die am Anfang der iterator.c Datei implementiert sind. Diese 3 Funktionen werden dann genutzt um mit dem Iterator zu interagieren vgl. `main` in iterator.c.

b) Implementieren Sie die Funktion `get_iterator_level_order_BT`, die einen Iterator zurückgibt, der den Binärbaum ebenenweise durchläuft. Schauen Sie sich die Definition von `get_iterator_pre_order_BT` an und wie die einzelnen Funktionen gesetzt werden und wie der Zustand `state` des Iterator definiert ist. Schreiben Sie entsprechende Funktionen um diesen Iterator zu erstellen.

c) Implementieren Sie die Funktion `get_iterator_pre_order_T`, die einen Iterator zurückgibt, der den Baum ebenenweise durchläuft. Implementieren Sie entsprechende Methoden und setzten Sie diese im Iterator.

d) Implementieren Sie die Funktion `get_iterator_level_order_T`, die einen Iterator zurückgibt, der den Baum in „level order" durchläuft. Implementieren Sie entsprechende Methoden und setzten Sie diese im Iterator.

e) Mit dem Iterator, der ebenenweise den Baum durchläuft, können Sie nun die Breitensuche (vgl. https://en.wikipedia.org/wiki/Breadth-first_search) implementieren. Implementieren Sie die Funktion `contains_Btree` und `contains_tree`, die `true` zurückliefern, wenn der key in der übergebenen Datenstruktur enthalten ist.

f) Optional: Schreiben Sie weitere `contains` Funktionen, die mit Tiefensuche (vgl. https://en.wikipedia.org/wiki/Depth-first_search) die Bäume durchsuchen.

Aufgabe 3: Multimengen

In dieser Aufgabe geht es um Multimengen, das sind Mengen, in denen Elemente beliebig oft vorkommen können. Wie bei einfachen Mengen sind die Elemente nicht geordnet. Die Template-Dateien für diese Aufgabe sind multiset.{c|h}. Machen Sie sich mit dem Template-Code vertraut. Insbesondere ist ein Eintrag in der Multimenge repräsentiert als `MultisetNode`-Struktur, die einen Zeiger auf das eigentliche Element (`void* element`) und die Anzahl des Auftretens des Elements in der Menge enthält (`int count`). Die Menge selbst ist durch die Struktur `Multiset` repräsentiert. Diese umfasst neben dem bucket-Array die Anzahl der verschiedenen Elemente (`int n_elements`) und die Anzahl der Instanzen (`int n_instances`). Beispielsweise gilt für die Menge {30, 4, 4, 7, 7 }: n_elements = 3 und n_instances = 5. `MultisetNode` enthält außerdem Zeiger auf eine Hash- und eine Vergleichsfunktion für Elemente. Kommentieren Sie die Beispielaufrufe in der `main`-Funktion geeignet ein bzw. aus.

a) Implementieren Sie die interne Hilfsfunktion `find_multiset`, um den Knoten zu ermitteln, in dem ein bestimmtes Element gespeichert wird (Vergleich mit `s->equal`).

b) Implementieren Sie die Funktion add_multiset, um ein Element in einer bestimmten Anzahl zu einer Multiset hinzuzufügen. Der Zustand der Multiset wird dadurch verändert.

c) Implementieren Sie die Funktion `copy_multiset`, um eine `Multiset` zu kopieren. Wenn `copy` ungleich `NULL` ist, sollen die Elemente selbst mit dieser übergebenen Funktion kopiert werden.

d) Implementieren Sie die Funktion `free_multiset`, um eine `Multiset` freizugeben. Wenn `free_element` ungleich `NULL` ist, sollen die Elemente selbst mit dieser übergebenen Funktion freigegeben werden.

e) Implementieren Sie die Funktion `intersection_multiset`, um die Schnittmenge zweier Multisets zu berechnen. Die Funktion soll die Eingabemengen nicht verändern und die Elemente selbst nicht kopieren. Es wird eine neue Ergebnismenge erzeugt. Die Schnittmenge ist wie folgt definiert. Sei c:= intersect(a, b). Dann gilt für alle x: count(c, x) = min(count(a, x), count(b, x)). Beispiel: a = { 1, 1, 2 }, b = { 1, 1, 1, 2, 2, 3 } ➔ c = a ∩ b = { 1, 1, 2 }.

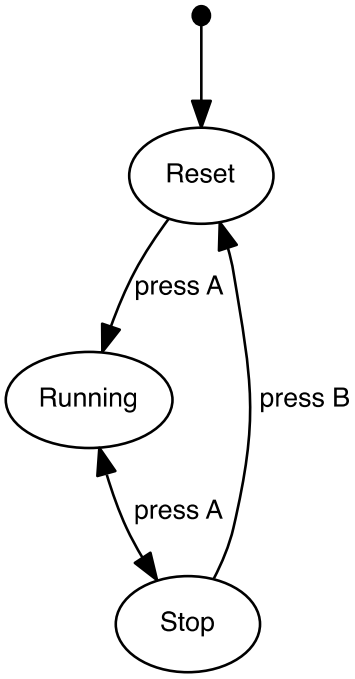
f) Implementieren Sie die Funktion `union_multiset`, um die Vereinigungsmenge zweier Multisets zu berechnen. Die Funktion soll die Eingabemengen nicht verändern und die Elemente selbst nicht kopieren. Es wird eine neue Ergebnismenge erzeugt. Die Vereinigungsmenge ist wie folgt definiert. Sei c:= union(a, b). Dann gilt für alle x: count(c, x) = max(count(a, x), count(b, x)). Beispiel: a = { 1, 1, 2 }, b = { 1, 1, 1, 2, 2, 3 } ➔ c = a ∪ b = { 1, 1, 1, 2, 2, 3 }.

g) Implementieren Sie die Funktion `sum_multiset`, um die Summe zweier Multisets zu berechnen. Die Funktion soll die Eingabemengen nicht verändern und die Elemente selbst nicht kopieren. Es wird eine neue Ergebnismenge erzeugt. Die Summe ist wie folgt definiert. Sei c:= sum(a, b). Dann gilt für alle x: count(c, x) = count(a, x) + count(b, x). Beispiel: a = { 1, 1, 2 }, b = { 1, 1, 1, 2, 2, 3 } ➔ c = a + b = { 1, 1, 1, 1, 1, 2, 2, 2, 3 }.

h) Implementieren Sie die Funktion `remove_multiset`, um die gegebene Anzahl Instanzen eines bestimmten Elements aus der `Multiset` zu entfernen. Wenn `free_element` ungleich `NULL` ist, soll das Element mit dieser übergebenen Funktion freigegeben werden. Diese Teilaufgabe ist schwieriger als die anderen Teilaufgaben.

Aufgabe 4: Stoppuhr als endlicher Automat

(Thematik erst in 14. Vorlesung)



Eine Stoppuhr soll als endlicher Automat implementiert werden. Die Stoppuhr hat zwei Knöpfe: „A" und „B". Nach Einlegen der Batterie befindet sie sich im Zustand „Reset". Die Zeitnahme läuft durch Druck auf Knopf „A". Ein weiterer Druck auf Knopf „A" dient zur Anzeige der aktuellen Zwischenzeit, aber die Zeitnahme läuft intern weiter. Ein erneuter Druck auf „A" zeigt wieder die laufende Zeit an. Ein Druck auf „B" führt zum Abbruch der Zeitnahme und zur Rückstellung der Anzeige auf Null. Siehe nebenstehende Abbildung. Die Template-Datei für diese Aufgabe ist stopwatch.c.

a) Erklären Sie als Kommentar im Quelltext die Aufgabe der Typen `Event` und `State`. Wozu dienen und was repräsentieren sie?
b) Erklären Sie als Kommentar im Quelltext die Aufgabe der Funktion `operate_fsm`. Was macht die Funktion genau? Gibt es Eingaben, für die sich die Funktion nicht korrekt verhält?
c) Implementieren Sie die Funktionen `running` und `stop`. Im Zustand „Running" soll nicht die laufende Zeit angezeigt werden (dies ist auf der Konsole schwierig zu implementieren). Stattdessen soll nur bei Zustandsübergängen eine Ausgabe erzeugt werden, nämlich beim Übergang von „Running" nach „Stop":

```
transition to state stop
time: 2 s
```

beim Übergang von „Stop" nach „Running":

```
transition to state running
```

und geim Übergang von „Stop" nach „Reset":

```
transition to state reset
```

Hinweise zum Editieren, Compilieren und Ausführen:

- mit Texteditor `file.c` editieren und speichern
- `make file` ← ausführbares Programm erstellen
- `./file` ← Programm starten (evtl. ohne `./`)
- Die letzten beiden Schritte lassen sich auf der Kommandozeile kombinieren zu: `make file && ./file`