Human-Computer Interaction Group

Leibniz Universität Hannover

Übungsblatt 11

Programmieren 1 – WiSe 21/22

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

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. Plagiate führen zum Ausschluss von der Veranstaltung.

Abgabe bis Donnerstag den 13.01.um 23:59 Uhr über *https://assignments.hci.uni-hannover.de/WiSe2021/Programmieren1*. Die Abgabe muss aus einer einzelnen Zip-Datei bestehen, die den Quellcode, ein PDF für Freitextaufgaben und alle weiteren nötigen Dateien (z.B. Eingabedaten oder Makefiles) enthält. Lösen Sie Umlaute in Dateinamen auf.

Hinweis: prog1lib

Die Dokumentation der *prog1lib* finden Sie unter der Adresse:
https://postfix.hci.uni-hannover.de/files/prog1lib/

Aufgabe 1: Öffnende und schließende Klammern

In dieser Aufgabe geht es um die Implementierung einer Syntaxprüfung von öffnenden und schließenden Klammern. Zu jeder öffnenden Klammer muss in der Zeichenkette ein passendes schließendes Gegenstück vorkommen. Schließende Klammern müssen zu der jeweiligen öffnenden Klammer passen. Folgende Klammernpaare sollen unterstützt werden: `( )`, `[ ]`, `{ }`, `< >`. Die Zeichenkette „`<{[()]}>`" ist valide. Die Zeichenketten „`(Test`" und „`([)]`" nicht.

Die Template-Datei für diese Aufgabe ist `parentheses.c`. Bearbeiten Sie alle mit `TODO` markierten Stellen.

a) Zum Lösen dieser Aufgabe ist es hilfreich einen *Stack* zu nutzen. Implementieren Sie diese Datenstruktur mit der für diese Aufgabe benötigten Funktionalität. Allokieren Sie Speicher dynamisch.

b) Implementieren Sie die Funktion `bool verify_parentheses(String text)`, die `true` zurück gibt wenn die Klammerung syntaktisch korrekt ist und `false` wenn nicht. Die Funktion muss in der Lage sein Zeichenketten beliebiger Länge zu behandeln.

c) Stellen Sie durch `report_memory_leaks(true)` sicher, dass dynamisch allokierter Speicher wieder freigegeben wird.

Aufgabe 2: Ersetzung in Zeichenketten ohne prog1lib

In dieser Aufgabe geht es um die Implementierung einer Funktion zur Ersetzung von Substrings in Strings. Die Funktion soll in einer Zeichenkette `str` alle Vorkommen der Zeichenkette `rep` durch eine dritte Zeichenkette `new` ersetzen. Alle anderen Zeichen sollen übernommen werden. Beispielsweise soll die Zeichenkette „`This is a tiny tiny test.`" nach „`tiny`" durchsucht werden und alle Vorkommen durch „`fun`" ersetzt werden. Das Ergebnis hierbei wäre: „`This is a fun fun test.`". Dabei soll die ursprüngliche Zeichenkette erhalten bleiben und immer eine neue dynamisch allokierte Zeichenkette erstellt werden.

Die Template-Datei für diese Aufgabe ist `find_replace.c`. Bearbeiten Sie alle mit `TODO` markierten Stellen. In dieser Aufgabe darf die *prog1lib* **nicht** genutzt werden. Funktionen zum Testen sind im Template gegeben.

a) Implementieren Sie die `find_substring` Funktion. Diese soll Ihnen den Index zurückgeben, an dem der Substring `substr` das erste Mal in dem String `str` auftritt. Der Parameter `starting` beschreibt den Index, an dem mit der Suche begonnen werden soll, sodass es möglich ist Substrings zu überspringen. Taucht der Substring nicht in dem String (ab dem Index `starting`) auf, soll `-1` zurückgegeben werden.

b) Implementieren Sie die `count_substrings` Funktion. Diese soll Zählen wie oft der Substring `substr` in dem String `str` vorkommt.

c) Implementieren Sie die `string_replace` Funktion. Der Parameter `str` gibt den zu bearbeitenden String an, `rep` beschreibt den zu ersetzenden Substring und `new` die Ersetzung. Schreiben Sie das Ergebnis in eine neue Zeichenkette, die Sie dynamisch allokieren. Allokieren Sie nur soviel Speicher wie nötig.

Aufgabe 3: Aufbau und Visualisierung eines Quadtree

In dieser Aufgabe geht es um die Implementierung und Darstellung der *Quadtree* Datenstruktur (siehe: *https://de.wikipedia.org/wiki/Quadtree*). Bei dieser Baumstruktur besitzt jeder Knoten, der kein Blatt ist, vier Kindknoten. Aus diesem Grund eignen sich Quadtrees besonders zum sortierten Speichen von Elementen im zweidimensionalen Raum. Jeder Knoten in einem Quadtree repräsentiert einen quadratischen Bereich des Raumes. Dieser Bereich wird definiert über die X- und Y-Koordinate der linken oberen Ecke und über die Breite und Höhe des Quadrats. Besitzt ein Knoten Kinder, so teilen die vier Kindnoten diesen Bereich dann wiederum in vier gleichgroße Teilbereiche (links oben, rechts oben, links unten, rechts unten) auf. Elemente werden dann so in den Baum eingefügt, dass sie in genau dem Blattknoten liegen, in dessen Bereich ihre Position liegt. Ein Blattknoten kann eine Liste aus Elementen beinhalten. Übersteigt die Menge an Elementen im Knoten jedoch einen Grenzwert, werden vier Kindknoten erzeugt und die Elemente auf diese, anhand ihrer Position, aufgeteilt.

Hier sei ein Beispiel gegeben:

- Der Quadtree wird durch einen Wurzelknoten ohne Kindnoten initialisiert. Der Wurzelknoten hat eine Höhe und Breite von 512 und liegt an Koordinate $(0, 0)$.
- Es werden sukzessive Elemente in den Baum eingefügt, deren Koordinaten im Bereich $[0, 512)$ liegen. Anfangs werden diese alle dem Wurzelknoten hinzugefügt.
- Übersteigt die Anzahl an Elementen im Wurzelknoten einen Grenzwert, werden dem Wurzelknoten vier Kindknoten hinzugefügt. Da diese den Bereich vierteilen, haben sie jeweils eine Höhe und Breite von 256. Die Koordinaten sind jeweils: $(0, 0)$ (links oben), $(256, 0)$ (rechts oben), $(0, 256)$ (links unten), $(256, 256)$ (rechts unten). Die Elemente im Wurzelknoten werden auf die vier neuen Knoten aufgeteilt, abhängig davon in welchem der Bereiche sie liegen.
- Werden nun weitere Elemente in den Baum eingefügt, wird dieser zuerst solange durchlaufen bis ein passender Blattknoten gefunden wurde. Dort wird das Element dann eingefügt. Gegebenenfalls muss der Blattknoten dann wieder aufgeteilt werden.

Die Template-Datei für diese Aufgabe ist `quad_tree.c`. Bearbeiten Sie alle mit `TODO` markierten Stellen.

a) Die Strukturen `Element` und `Node` sowie deren Konstruktorfunktionen sind bereits vorgegeben, machen Sie sich mit diesen vertraut.

b) Implementieren Sie die `free_node` Funktion, um den Speicher eines Baumes rekursiv freigeben zu können.

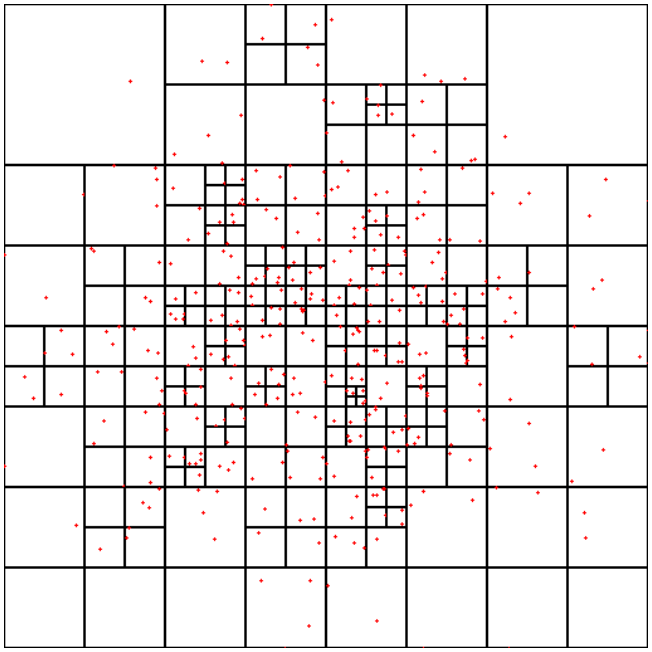
c) Implementieren Sie die `insert_node` Funktion, die ein Element in einen Knoten einfügt. Besitzt dieser Knoten Kindknoten, so soll das Element direkt an den jeweiligen Kindknoten gereicht werden. Ist der Knoten jedoch ein Blattknoten, soll das Element in die Liste des Knotens eingefügt werden.

   Befinden sich nach dem Hinzufügen mehr als `MAX_ELEMENTS` Elemente in dem Blattknoten, sollen diese auf vier Kindknoten aufgeteilt werden. Das soll jedoch **nur** passieren, wenn der Blattknoten nicht tiefer als `MAX_DEPTH` im Baum liegt. Nutzen sie das Feld `depth` der `Node` Struktur um diese Information zu verwalten.

d) In der `main` Funktion befindet sich bereits Programmcode der Ihren Baum mit zufällig platzierten Elementen füllt. Um das Verhalten Ihres Codes zu verifizieren soll der Zustand des Baumes nach dem Hinzufügen graphisch ausgegeben werden. Zu diesem Zweck wird in der `main` Funktion das Array `canvas` erzeugt und als `PNG` Datei abgespeichert. `canvas` ist ein eindimensionales Array, in dem zeilenweise die zweidimensionalen Bilddaten gespeichert werden. Jeder Pixel in dem Bild wird durch drei Bytes repräsentiert, in denen der jeweilige Rot-, Grün- und Blauanteil gespeichert wird.

   Implementieren Sie zuerst die `set_pixel` Funktion, die den Rot-, Grün- und Blauwert eines Pixels an der gegebenen X und Y Position auf dem `canvas` setzt.

e) Implementieren Sie nun die `draw_node` Funktion, die den gegebenen Knoten rekursiv auf das `canvas` malt. Um die Bereiche, die von Blattknoten abgedeckt werden, sollen schwarze Rahmen gezeichnet werden. Elemente sollen durch rote Punkte gekennzeichnet werden. Nutzen Sie Ihre `set_pixel` Funktion. Die entstehende Grafik könnte beispielsweise so aussehen:



Hinweis: In dem gegebenen Template werden die Listen aus der Programmieren 1 Bibliothek genutzt. In der Liste eines Knotens werden Elemente direkt (nicht als Pointer) gespeichert.

Folgendermaßen kann über eine Liste iteriert werden:

```
ListIterator it = l_iterator(node->elements);
while(l_has_next(it)) {
    Element * e = l_next(&it);
}
```

So kann ein Element in ein Liste eingefügt werden:

```
Element e = new_element(0, 0);
l_append(node->elements, &e);
```

Es genügt ein einzelner Aufruf von `l_free` um den Speicher einer Liste mitsamt Inhalt freizugeben.