Prof. Dr. F. Sadlo, L. Büch *Heidelberg, 1. Dezember 2015*

Übungsblatt 7

Anmerkung: Auf diesem Übungsblatt operieren Sie viel mit Zeigern. Bei der Verwendung von Zeigern treten sehr schnell subtile Programmierfehler auf. Prüfen Sie stets, ob ein Zeiger ungleich null ist, bevor Sie ihn verwenden, denken Sie an Sonderfälle wie leere Listen oder Listen mit nur einem Eintrag, und geben Sie zur Not die Speicheradressen mit der `print()`-Anweisung aus, falls Sie nicht gleich wissen, wo sich der Null-Zeiger versteckt, den Sie aus Versehen dereferenziert haben.

Aufgabe 7.1 Hase und Igel [6 Punkte]

Sie haben in der Vorlesung die Datenstruktur der einfach verketteten Liste kennengelernt. Diese beginnt mit einem Element, auf das keines der Elemente zeigt, und endet mit einem Element, das auf die spezielle Adresse `0` zeigt. Wenn man die Definition etwas erweitert, kann es jedoch sein, dass einer der Zeiger zurück auf eines der vorherigen Elemente zeigt. Die Kette ist dann nicht linear, sondern ähnelt dem griechischen Buchstaben ρ. Durchläuft man eine solche Liste, so gerät man in eine Endlosschleife, und die besuchten Elemente wiederholen sich zyklisch.

Ziel dieser Aufgabe ist es, einen Algorithmus zu implementieren, der für eine gegebene Liste feststellen kann, ob diese einen Zyklus enthält oder, wie in der Vorlesung vorgestellt, linear ist. Dazu werden zwei Zeiger gleichzeitig durch die Liste bewegt, wobei einer von ihnen (der *Hase*) stets zum übernächsten Element springt, während der zweite (der *Igel*) wie üblich alle Elemente der Reihe nach besucht.

a) Begründen Sie theoretisch, wie man mit dem Konzept von *Hase* und *Igel* erkennt, ob eine Liste einen Zyklus enthält. Betrachten Sie dabei den allgemeinen Fall einer Liste mit n-elementigem Zyklus, welchem ein k-elementiger linearer Teil vorausgeht. Beschreiben Sie außerdem, wie man n algorithmisch bestimmen kann. [*3 Punkte*]

b) Schreiben Sie eine Funktion, die für gegebenes $k \geq 0$ und $n \geq 0$ eine Liste der Länge n erzeugt, den `next`-Zeiger des letzten Elements von `0` auf die Adresse des ersten Elements ändert und somit einen Zyklus der Länge n konstruiert, und schließlich eine Liste der Länge k erzeugt, die auf ein beliebiges Element dieses Zyklus zeigt. Verwenden Sie die Implementierung der einfach verketteten Liste aus der Vorlesung.

Schreiben Sie außerdem eine Funktion, die den Hase-Igel-Algorithmus implementiert und für eine gegebene Liste die Länge n des Zyklus ausgibt. Testen Sie ihre Funktion für

- $k > 0$ und $n > 0$ (Normalfall)
- $k = 0$ und $n > 0$ (reiner Zyklus)
- $k > 0$ und $n = 0$ (lineare Liste)
- $k = 0$ und $n = 0$ (leere Liste)

[*3 Punkte*]

Aufgabe 7.2 Warteschlangen (Queues) [✍ 8 Punkte]

a) Programmieren Sie eine Warteschlange für Integer-Zahlen in Form einer einfach verketteten Liste. Diese soll Funktionen bereitstellen, die ein neues Element in die Schlange einreihen bzw. am anderen Ende entfernen und zurück geben. Benutzen Sie dabei *call by reference*, damit die Liste in der `main()`-Funktion durch Ihre Funktionen geändert wird. Sie können die Listenimplementierung aus der Vorlesung als Basis verwenden. [*2 Punkte*]

b) Erweitern Sie nun Ihr Programm, indem Sie die Liste durch eine doppelt verkettete Liste ersetzen. In einer solchen Liste hat jedes Element zusätzlich einen Zeiger auf seinen Vorgänger. Zudem enthält die Liste selbst einen Zeiger auf ihr letztes Element. [*4 Punkte*]

c) Füllen Sie Ihre beiden Warteschlangen mit einer großen Zahl (bspw. 100000) zufälliger Integer-Zahlen[1] und entnehmen Sie diese danach wieder.

Benutzen Sie dynamische Speicherverwaltung für die Integerzahlen. Was stellen Sie für Laufzeitunterschiede für die Operationen Hinzufügen und Entfernen bei einfach und doppelt verketteter Liste fest? Erklären Sie! Geben Sie die Komplexität dieser Operationen in der Anzahl n der Listeneinträge an.

[*2 Punkte*]

Aufgabe 7.3 Binärbaum für arithmetische Ausdrücke [✍ 6 Punkte]

Erstellen Sie eine rekursive Datenstruktur **`struct`** `BaumElement`, die einen Binärbaum repräsentiert. Jedes Element kann einen Operator (+, -, *, /), eine ganze Zahl oder eine Variable (lateinische Buchstaben) enthalten. Außerdem enthält es zwei Zeiger auf andere Baum-Elemente. Falls es einen Operator enthält, zeigen die Zeiger auf diese Elemente, andernfalls sind sie `null`.

Schreiben Sie eine Methode `create_tree`, mit der man Postfixausdrücke wie die folgenden einlesen und in eine Baumstruktur verwandeln kann:

`67 55 - 54 6 / + 2 *` und auch `X 1 - X 1 + *`

Erzeugen Sie aus den beiden obigen Ausdrücken zwei Bäume. Schreiben Sie Methoden `print_post`, `print_pre` und `print_in`, die für eine gegebene Baumstruktur den entsprechenden Ausdruck (in Postfix-, Präfix- bzw. Infixnotation) als Zeichenkette ausgeben. Testen Sie alle drei Methoden mit den beiden Bäumen aus dem vorherigen Test. [*4 Punkte*]

Schreiben Sie eine Methode `insert`, die in einem gegebenen Baum eine gegebene Variable durch einen anderen Baum ersetzt. Führen Sie dies an dem obigen Beispiel durch. Ersetzen Sie 'X' im rechten Ausdruck durch den linken Ausdruck. Welches Problem kann auftreten, wenn die Baumelemente für entsprechende Anwendungen auch Zeiger zu ihren Elternknoten speichern? Wie kann man es beheben? Geben Sie den resultierenden Ausdruck in Infixschreibweise aus. [*2 Punkte*]

Abgabe: 10. Dezember 2015, 14:15 Uhr,

[1] Sie können dafür mittels `#include<cstdlib>` die Funktion `rand()` benutzten, die passende Zahlen generiert.

in die Zettelkästen in der Ecke zwischen HS 2 und Seifertraum (INF 288)
sowie per E-Mail (✉) an Ihren Tutor