UNIVERSITÄT HEIDELBERG
Zukunft. Seit 1386.

IWR
Interdisciplinary Center for Scientific Computing

IPI: Einführung in die Praktische Informatik
Wintersemester 2018
Prof. Björn Ommer
ommer@uni-heidelberg.de

Übungsblatt 8: Zeiger und dynamischer Speicher

Abgabe am 19.12.2018, 13:00.

Aufgabe 1: Hase und Igel (6P)

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 k und 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 sowohl die Länge k des Linearteils als auch 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 2: Mandelbrot **(8P)**

Ein äußerst beliebtes Motiv auf Büchern, Postern und Bildschirmhintergründen ist die *Mandelbrot-Menge*. Diese Menge wird wie folgendermassen definiert: Für jede komplexe Zahl $c \in \mathbb{C}$ wird die Folge z_i von Elementen aus $\mathbb{C}$ rekursiv definiert als:

$$z_0 := 0$$
$$z_{n+1} := f(z_n, c) := z_n^2 + c$$

Die Zahl c gehört zur Mandelbrot-Menge M, wenn die dazugehörige Folge sich nicht unendlich weit von Null entfernt, d.h. falls ein $C \in \mathbb{R}$ existiert, so dass $|z_n| := \sqrt{\text{Re}(z_n)^2 + \text{Im}(z_n)^2} \leq C$ für alle $n \in \mathbb{N}$ gilt.

a) Vervollständigen Sie die Implementierung von `add_complex`, `multiply_complex`, `schritt`, `betrag` und `trajektorie` in `mandelbrot.cc`. Im Gegensatz zu der in der Vorlesung vorgestellten Implementierung sollen `add_complex`, `multiply_complex`[1] und `schritt` nun als Prozeduren implementiert werden indem sie *in-place* operieren, d.h. statt eine neue komplexe Zahl zu erzeugen, soll das Ergebnis im ersten Argument abgespeichert werden. *[2 Punkte]*

b) Zur Erkundung der Mandelbrotmenge werden wir ein Bild erzeugen welches den Betrag von z_n für ein fixes n darstellt. Zur digitalen Repräsentation von Bildern nutzen wir Pixel (*picture elements*). In dieser Aufgabe entspricht ein Pixel einem `float`, wobei Werte kleiner gleich 0 schwarze Pixel darstellen, Werte größer gleich 1 weiße Pixel, und Werte zwischen 0 und 1 entsprechende Grauwerte. Bilder besitzen zwei Dimensionen, eine Höhe H (engl. *height*) und eine Breite W (engl. *width*), womit insgesamt $H \cdot W$ Pixel benötigt werden. Während die Reihenfolge der Pixel prinzipiell beliebig gewählt werden kann, folgen wir der *row-major* Konvention, in welcher Pixel Zeile-für-Zeile angeordnet sind. Demnach wird ein Bild der Höhe 2 und Breite 4 zum Beispiel folgendermaßen dargestellt:

$$\begin{matrix} \text{pixel}_0 & \text{pixel}_1 & \text{pixel}_2 & \text{pixel}_3 \\ \text{pixel}_4 & \text{pixel}_5 & \text{pixel}_6 & \text{pixel}_7 \end{matrix} \tag{1}$$

Bei der Arbeit mit Bildern iteriert man häufig über Zeilen $i \in \{0, \ldots, H-1\}$ und Spalten $j \in \{0, \ldots, W-1\}$. Diese müssen dann wieder den entsprechenden Pixeln zugeordnet werden. So entspricht z.B. $(i, j) = (0, 2)$ dem Pixel pixel_2 und $(i, j) = (1, 2)$ dem Pixel pixel_6. Vervollständigen Sie die Funktionen `create_image`, `get_pixel` und `set_pixel` in `mandelbrot.cc`. *[2 Punkte]*

c) Um nun eine Visualisierung der Mandelbrotmenge zu erhalten, erzeugen Sie zunächst zwei Bilder `x_coords` und `y_coords` der Höhe H und Breite W. Implementieren Sie die Funktion `init_grid` welche die Werte der Bilder passend zu

[1] Definition der Multiplikation für komplexe Zahlen:
$z_1 \cdot z_2 := \{\text{Re}(z_1)\,\text{Re}(z_2) - \text{Im}(z_1)\,\text{Im}(z_2)\} + \{\text{Re}(z_1)\,\text{Im}(z_2) + \text{Im}(z_1)\,\text{Re}(z_2)\}\, i$

einem Gitter $(x_{\min}, x_{\max}, y_{\min}, y_{\max})$ initialisiert. So soll `x_coords` am linken Rand den Wert $x_{\min}$ haben und am rechten Rand den Wert $x_{\max}$. Entsprechend soll `y_coords` am oberen Rand den Wert $y_{\min}$ haben und am unteren Rand den Wert $y_{\max}$. Für $H = 2$, $W = 4$ und $(x_{\min}, x_{\max}, y_{\min}, y_{\max}) = (0.0, 1.0, 0.0, 1.0)$ zum Beispiel:

$$\text{x\_coords} = \begin{matrix} 0.0 & 0.25 & 0.5 & 0.75 & 1.0 \\ 0.0 & 0.25 & 0.5 & 0.75 & 1.0 \end{matrix} \quad \text{y\_coords} = \begin{matrix} 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \end{matrix}$$

Erzeugen Sie ein weiteres Bild derselben Größe. Zu jedem Pixel korrespondiert nun eine komplexe Zahl c, deren Realteil durch `x_coords` und Imaginärteil durch `y_coords` gegeben ist. Nutzen Sie die Funktion `trajektorie` aus Aufgabenteil a) um jedem Pixel den Betragswert des N-ten zu c gehörigen Folgenelements zuzuweisen. Nutzen Sie die Funktion `save_image` um das Bild als PNG Datei abzuspeichern, welche von den meisten Bildbetrachtern geöffnet werden kann. Um ein Bild namens `bild` vom Typ `Image` unter dem Namen `"bild.png"` abzuspeichern, rufen Sie `save_image("bild.png", bild.pixel, bild.W, bild.H)` auf. Überprüfen Sie das Bild welches Sie für die Werte $H = 256, W = 256, N = 100$ und $(x_{\min}, x_{\max}, y_{\min}, y_{\max}) = (-1.5, 0.5, -1.0, 1.0)$ erhalten. Beachten Sie allerdings, dass Ihr Programm auch mit unterschiedlichen Werten für Höhe und Breite laufen sollte. *[2 Punkte]*

d) Eine einfache Methode um Bilder zu verkleinern, besteht darin lediglich jeden zweiten Pixel zu behalten. In obigem Beispiel (1) wäre das entsprechende Resultat also

$$\text{pixel}_0 \quad \text{pixel}_2$$

Implementieren Sie die entsprechende Funktion `downsample` in `mandelbrot.cc`. Achten Sie darauf Speicherplatz effizient zu nutzen, d.h. nach der Operation soll kein ungenutzter Speicherplatz mehr reserviert sein. *[2 Punkte]*

Sehenswürdigkeiten in der Mandelbrotmenge: Die Mandelbrotmenge besitzt unendlich viele Sehenswürdigkeiten. Als Einstieg sei zum Beispiel

$$\begin{aligned} x_{\min} &= -0.7453 - \delta \\ x_{\max} &= -0.7453 + \delta \\ y_{\min} &= 0.1127 - \delta \\ y_{\max} &= 0.1127 + \delta \\ \delta &= 10^{-4} \end{aligned}$$

empfohlen. Einen ausgiebigen Reiseführer finden Sie unter `http://www.cuug.ab.ca/dewara/mandelbrot/images.html`.

Prof. Björn Ommer
ommer@uni-heidelberg.de

Aufgabe 3: Binärbaum **(6P)**

Erstellen Sie eine rekursive Datenstruktur `struct BaumElement`, die einen Binärbaum repräsentiert. Jedes Element kann einen Operator (+, -, *, /), eine natürliche 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 dessen Argumente, 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. Benutzen Sie die Vorlage in `baum.cc` und den Datentyp `std::string` um die Ausdrücke zu generieren. Objekte von diesem Typ können durch den `+` Operator konkateniert werden. *[4 Punkte]*

Implementieren Sie die 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? *[2 Punkte]*