Übungsblatt 5

Abgabe: 28.06.2024

Ab diesem Übungsblatt gibt es keine feste Aufgabenteilung zwischen den Bearbeitenden mehr.

Aufgabe 1 Qual der Wahl (20 %)

Nennt zu jeder der folgenden Methoden eine Datenstruktur, für die sie besonders effizient implementiert werden kann, und eine, die sich weniger gut eignet. Begründet eure Wahl und betrachtet dabei die Komplexitäten der jeweiligen Operationen (E ist der Elementtyp der Datenstruktur).

- **void insert(E e)** – Einfügen eines Elementes.
- **E accessMin()** – Das kleinste Element zurückgeben.
- **boolean contains(E e)** – Ist das gefragte Element enthalten?
- **E successor(E e)** – Nachfolger eines Elements zurückgeben.
- **E kthElement(int k)** – Das k-größte Element zurückgeben.

Aufgabe 2 Regionales Bildungsprogramm

In der Bildverarbeitung versucht man oft, zusammenhängende Regionen in Bildern zu finden, um sie leichter weiterverarbeiten zu können. Die Klasse *ImageViewer*, die im Archiv zu diesem Übungsblatt enthalten ist, erlaubt es bereits, ein Bild zu laden und es im Original sowie als Grauwert und Schwarzweißbild darzustellen. Auch angeboten wird die Darstellung eines segmentierten Bildes, die leider bisher nur ein schwarzes Bild anzeigt. Eigentlich sollen hier alle zusammenhängenden, weißen Regionen aus dem Schwarzweißbild in unterschiedlichen Farben dargestellt werden. Das auch im Archiv enthaltene Escher-Bild „Bond of Union“ (`http://www.artchive.com/artchive/e/escher/escher_bond.jpg`) sollte segmentiert wie im Bild unten anzeigt werden. Dazu muss die Methode *getSegmented* in der bereitgestellten Klasse *ImageHandler* erweitert werden. Ein zweites, einfacheres Bild liegt als weiterer Testfall bei.

Aufgabe 2.1 Horizontal zusammenfassen (20 %)

Um dies effizient zu implementieren, wird zweistufig vorgegangen. Zuerst wird ein Lauflängenbild erstellt, das aus einer Folge von sog. *Runs* besteht, wobei jeder *Run* einen horizontalen *weißen* Streifen des Bildes kodiert. Die Folge kann z.B. durch eine *ArrayList<Run>* repräsentiert werden. Vorerst ist ein *Run* ein Tripel (x_{start}, x_{ende}, y), wobei (x_{start}, y) das linke Ende des Streifens bezeichnet und (x_{ende}, y) die Position unmittelbar rechts hinter dem Streifen, d.h. $x_{ende} - x_{start}$ ist die Länge des Streifens. Zum Erzeugen der *Runs* wird das Bild von oben nach unten (äußere Schleife) und von links nach rechts (innere Schleife) durchlaufen. Dabei werden schwarze Pixel übersprungen und aufeinanderfolgende weiße Pixeln in jeweils einem *Run* zusammengefasst und hinten an die Folge angefügt.[1] Jeder *Run* repräsentiert nur Pixel aus einer Zeile, d.h. spätestens beim Erreichen des rechten Rands endet er.

Aufgabe 2.2 Regionen repräsentieren (15 %)

Regionen werden als Union-Find-Strukturen dargestellt, wobei die *Runs* die Elemente sind, aus denen die Regionen bestehen. Daher wird die Klasse *Run* um eine Referenz auf den übergeordneten (Eltern-)*Run* erweitert. Anfangs zeigt diese Referenz bei jedem *Run* auf sich selbst, was ihn zur Wurzel einer anfangs ein-elementigen Region macht. Fügt eine Methode *getRoot* hinzu, die das Wurzelelement der Region zurückliefert, zu der ein Run gehört, indem sie sich an den Elternreferenzen entlang hangelt. Diese soll den durchlaufenen Pfad auch komprimieren.

Aufgabe 2.3 Regionen vereinigen (10 %)

Schreibt eine Methode, die zwei Regionen zu einer vereinigt. Die Parameter der Methode sind zwei *Runs*, die zu den zu vereinigenden Regionen gehören. Zuerst müssen die Wurzeln beider Regionen bestimmt werden. Dann wird die Wurzel der einen Region als Elternelement der anderen Wurzel gesetzt. Es soll immer die Wurzel die gemeinsame werden, die weiter vorne in der Folge steht, was an den Koordinaten der beiden Kandidaten abgelesen werden kann.

Aufgabe 2.4 Regionenbildung (25 %)

Nun müssen die *Runs* vertikal zu Regionen zusammengefasst werden. Sie werden der Reihe nach mit zwei Indizes i und j durchlaufen. Die Idee ist, dass i vorneweg läuft und j immer eine Zeile hinterher hinkt und dass die jeweiligen *Runs* horizontal immer auf etwa einer Höhe sind, so dass eventuelle Nachbarschaften festgestellt werden können.

In jedem Durchlauf wird überprüft, ob die durch i und j bezeichneten *Runs* benachbart sind. Das ist der Fall, wenn j in der Vorgängerzeile von i ist ($run_j.y + 1 = run_i.y$) und wechselseitig gilt, dass der linke Rand des einen *Runs* kleiner als der rechte Rand des anderen ist ($run_j.x_{start} < run_i.x_{end} \wedge run_i.x_{start} < run_j.x_{end}$). Sind zwei *Runs* benachbart, werden ihre Regionen vereinigt.

Zusätzlich wird in jedem Durchlauf dann entweder i oder j um eins erhöht. j wird weitergezählt, wenn $run_j.y + 1 < run_i.y$ oder wenn j zwar bereits in der richtigen Zeile ist, aber $run_j.x_{end} < run_i.x_{end}$. Ansonsten wird i erhöht. Die Schleife endet, wenn i das Ende der Folge erreicht hat.

Aufgabe 2.5 Einfärben (10 %)

Durchlauft nun die *Runs* und zeichnet diese in das Bild ein. Ist ein *Run* eine Wurzel, wird er in einer zufälligen Farbe gezeichnet, die von der Methode *getRandomColor* erzeugt wird. Ist er keine Wurzel, wird er in der Farbe gezeichnet, in der auch schon seine Wurzel eingetragen wurde.

[1] Pixel können mit *getRGB* aus dem Bild ausgelesen und mit den vordefinierten Konstanten *BLACK* und *WHITE* verglichen werden.