Technische Universität Berlin **SS99**
Fachbereich Mathematik
Sekr. MA 6-1 / Frau Marcus

Dr. Michael Joswig

Olaf Jahn
Matthias Müller, André Opitz

2. Übung zur Computerorientierten Mathematik II

Abgabe: 06.05.1999 (vor der Übung)

5. Aufgabe (3 + 1 + 2 + 2 Punkte)

Dünn besetzte Matrizen sind Matrizen mit wenigen Einträgen ungleich Null. Speichert man diese Matrizen in der bisherigen Form als Arrays von Arrays ab, so enthält der größte Teil des Speichers keine wichtigen Information.

- Entwickelt eine dynamische Datenstruktur in Form einer Java-Klasse für dünn besetzte Matrizen. Die Datenstruktur soll nur Einträge ungleich Null enthalten und die dazugehörigen Indizes. Von jedem Matrixelement mit Eintrag ungleich Null soll man in $O(1)$ Zeit Zugriff auf das nächste derartige Element in derselben Zeile bzw. derselben Spalte haben, um arithmetische Operationen effizient durchführen zu können. Weiterhin sollt Ihr den Zugriff auf das jeweils erste Element einer Zeile bzw. Spalte in $O(1)$ Zeit garantieren.
- Seht eine Methode zur Rückgabe des Elementes a_{ij} vor.
- Schreibt einen Konstruktor für diese Klasse, der als Parameter eine Matrix in der bisher bekannten Form erwartet.
- Schreibt eine Methode, die eine als Parameter übergebene dünn besetzte Matrix zur eigenen (`this`) dazuaddiert. Was soll passieren, wenn die beiden Matrizen nicht zueinander passen?

6. Aufgabe (3 Punkte)

Gegeben ist ein Array von n Zahlen aus $\{1, 2, \ldots, C\}$. Entwerft einen Algorithmus (Pseudocode), der die Zahlen in $O(n + C)$ Zeit sortiert. Vergeßt den Beweis der Laufzeit nicht.

Wie kann es sein, daß dieser Algorithmus für konstantes C den Array in $O(n)$ sortiert, wo doch in CoMa I gezeigt wurde, daß die untere Komplexitätsschranke dafür bei $\Omega(n \log n)$ liegt?

7. Aufgabe (5 Punkte)

Gebt die Worst case-Laufzeiten (in O-Notation) für die in der Tabelle aufgeführten Standardoperationen für verschiedene Listentypen an. Erläutert für die nicht-trivialen Fälle Eure Angaben mit ein paar Stichworten.

Die Liste L mit n Elementen wird durch eine Referenz *head* auf den Anfang der Liste, bzw. im Fall doppelt verketteter Listen durch eine weitere Referenz *tail* auf das Ende der Liste repräsentiert. Der zu suchende Wert wird mit k bezeichnet und *ptr* ist eine Referenz auf ein spezielles Listenelement.

„abst. sortiert“ bedeutet, daß die Liste vor und nach der Operation absteigend sortiert ist.

	einfach verkettet		doppelt verkettet	
	unsortiert	abst. sortiert	unsortiert	abst. sortiert
k in L suchen				
k in L einfügen				
Element ptr in L löschen				
ptr auf Nachfolger von ptr setzen				
ptr auf Vorgänger von ptr setzen				
Maximales Element in L finden				
Minimales Element in L finden				

2. Programmieraufgabe

Abnahme: spätestens 13.05.99 zur Rechnervorrangzeit

Ihr sollt eine Application schreiben, die ein Labyrinth aus einer Datei (per Menü und `FileDialog`) einliest und einen Weg von Eingang zum Ausgang berechnet, falls es einen gibt.

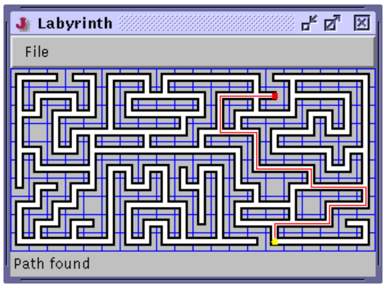
Die Lösung soll samt Weg graphisch dargestellt werden, wozu Ihr von uns eine entsprechende Klasse gestellt bekommt. Durch Mausklick auf ein existierendes Wegstück soll es gelöscht werden, durch Klick auf ein „fehlendes" soll es eingefügt werden. Der Weg ist ggf. dann neu zu berechnen.

Das Labyrinth wird Euch als eine Folge von „Fliesen" samt Koordinaten gegeben. Ihr müßt zunächst die Fliesen, die etwa für Abbiegungen, Geraden oder T-Stücke stehen, zu einem Graphen (mit Adjazenzlisten) zusammensetzen. Im Graphen sucht Ihr dann (irgend)einen Weg. Für die graphische Ausgabe stellen wir Euch eine Klasse zur Verfügung, der Ihr nur Euren Graphen und den Weg übergeben müßt. Der zu benutzende Algorithmus zum Konvertieren der Fliesen zu Graphknoten wird Euch in der Übung und den WWW-Seiten näher vorgestellt.

Da die Aufgabe etwas aufwendiger ist, bekommt Ihr insgesamt zwei Wochen Zeit. Allerdings gibt es nach einer Woche einen Meilenstein abzunehmen:

Bis 6. Mai: Implementiert eine einfach verkettete Liste, die sich selbst sortieren kann (und dem `SortableListWithPointer`-Interface genügt). Es soll Mergesort auf Listen benutzt werden. Testet sie ausgiebig. Implementiert zusätzlich eine Graphenklasse samt Knotenklasse (Interfaces `LabyrinthGraph` und `Node`). Das beinhaltet auch das Einlesen und Konvertieren der Daten. Die Methode `computePath` könnt Ihr zunächst leer lassen.

Bis 13. Mai: Jetzt implementiert die Suche im Graphen, mitsamt einer Klasse für Wege (Interface `Path`). Außerdem sollt Ihr jetzt den `LabyrinthViewer` (von uns gestellt) benutzen und in einen Frame einbetten. Durch Mausklick soll wie oben beschrieben das Löschen und Einfügen von Kanten möglich sein. Das ganze könnte dann so aussehen:



Details zur Aufgabenstellung (Dateiformat, Interfaces, vorgefertigte Klassen und Beispiellabyrinthe) findet Ihr über die CoMa-WWW-Seite.