Universität Karlsruhe (TH)
Fakultät für Informatik
Lehrveranstaltung Informatik II

Sommersemester 2003

Prof. Dr. Peter C. Lockemann
(lockemann@ira.uka.de)
Gábor M. Surányi (suranyi@ira.uka.de)
Daniel Pfeifer (pfeifer@ira.uka.de)
Philipp Bender (bender@ira.uka.de)

Übungsblatt 7

Algorithmenentwicklung, Iteratoren und der wp-Kalkül

Abgabe: bis 18.06.2003, 13:30 Uhr in den Einwurfkästen im Untergeschoß des neuen Infobaus

Erreichbare Punkte: 25 Theoriepunkte (25 T), 26 Praxispunkte (26 P)

Aufgabe 1: *Labyrinth-Generator mit Teile-und-Herrsche-Strategie (12 T/ 12 P)*

In dieser Aufgabe sollen Sie einen Algorithmus zur Generierung von Labyrinthen entwickeln. Ein Labyrinth ist dabei ein rechteckiges, umrandetes Gitter der Größe m×n aus dem Gitterelemente entfernt wurden. Die Gitter-Elemente repräsentieren dabei die Wände eines Labyrinths und die Felder zwischen den Gitterelementen begehbare Räume.
Für ein korrektes Labyrinth müssen folgende Regeln gelten:

1. Jeder Raum im Labyrinth ist von jedem anderen Raum erreichbar.
2. Von einem beliebigen Raum im Labyrinth zu einem anderen (beliebigen) Raum gibt es genau einen (zyklenfreien) Pfad.
3. Es gibt genau einen Eingang und einen Ausgang. Der Eingang ist oben auf der linken Seite, der Ausgang unten auf der rechten Seite des Labyrinths (auf Grund fehlender Elemente in der Umrandung des Gitters).

Abbildung 1 zeigt ein korrektes Beispiel-Labyrinth der Größe 20×20.

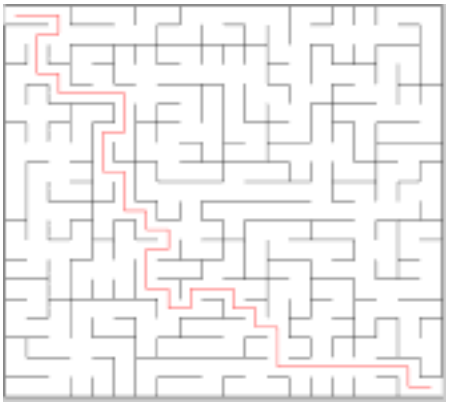

Abbildung 1: Korrektes Labyrinth der Größe 20×20 mit Pfad vom Eingang zum Ausgang

a) Entwickeln Sie nun einen Algorithmus zur Generierung von korrekten Labyrinthen beliebiger Größe. Dabei soll bei jeder Ausführung (mit hoher Wahrscheinlichkeit) ein anderes Labyrinth mit zufälliger Struktur erzeugt werden. Sie können hierzu davon ausgehen, daß Ihnen ein Zufallsgenerator zur Verfügung steht, der zufällige Integer-Zahlen erzeugt. Verwenden Sie für den Labyrinth-Generator die **Teile-und-Herrsche-Strategie** und formulieren Sie Ihren Algorithmus in Java-Pseudo-Code!

   Sie können annehmen, daß Ihnen eine Datenstruktur zur Verfügung steht, die die existierenden bzw. fehlenden Wände eines Labyrinths gegebener Größe speichert. Initial enthält die Datenstruktur alle entspr. Gitterelemente und die Umrandung. (Die Labyrinthregeln von oben sind dann natürlich nicht erfüllt.) (7 T)

Hinweise: Die Beantwortung der folgenden Fragen kann Ihnen bei der Entwicklung des Algorithmus helfen:
- Werden die Labyrinthregeln 1. und 2. für eine gegebenes Labyrinth verletzt, wenn man den Ein- bzw. Ausgang an eine andere Stelle legt?
- Angenommen Sie haben 2 korrekte Labyrinthe der Größe m×n und m×p. Können Sie damit ein korrektes Labyrinth der Größe m×(n+p) erzeugen?
- Welche Struktur hat ein Labyrinth der Größe 1×1 abgesehen von den Ausgängen?

b) Berechnen Sie den zu erwartenden Zeit- **und** Speicheraufwand für die Labyrinth-Erzeugung im O-Kalkül in Abhängigkeit der Größe eines Labyrinths. Bestimmen Sie hierzu eine geeignete aufwandserzeugende Operation, stellen Sie die entspr. Rekurrenzengleichung für den Algorithmus auf und lösen Sie die Gleichung. (2,5 T)
Hinweis: Sie dürfen dabei eine (bestimmte) vereinfachende Annahme über die Größe der Teile bzgl. des Teile-und-Herrsche-Algorithmus machen.

c) Den Zeit- und Speicheraufwand bei der Labyrinth-Generierung kann man verringern, indem man das Erzeugen und Verschmelzen von Labyrinth-Datenstrukturen vermeidet. Anstatt separate Objekte für Teillabyrinthe zu Erzeugen, kann man direkt mit einem einzigen Labyrinth-Objekt der initial geforderten Größe m×n arbeiten. Die Generierung der Teillabyrinthe erfolgt dann direkt auf (sich nicht überlappenden) Teilen des initialen Labyrinth-Objekts. Damit spart man sich auch das Kopieren von gesetzten/entfernten Wänden bei der Vereinigung von Teillabyrinthen.
Berechnen Sie auch für diesen Fall den mittleren Zeit- **und** Speicheraufwand für die Labyrinth-Erzeugung im O-Kalkül in Abhängigkeit der Größe eines Labyrinths. Bestimmen Sie wiederum eine geeignete aufwandserzeugende Operation, stellen Sie die entspr. Rekurrenzengleichung für den Algorithmus auf und lösen Sie die Gleichung. (2,5 T)

d) Implementieren Sie (wahlweise) den Algorithmus der aus a) bzw. c) folgt mit Hilfe des Labyrinth-Frameworks, das dem Übungsblatt beigefügt ist. Entwickeln Sie hierzu die Klasse **`StudentMazeSolver`** mit der Methode
```
public Maze generateMaze(int width, int height,
                         MazeGeneratorListener listener);
```
Den Parameter **`listener`** können/dürfen Sie im Rahmen der Aufgabenstellung ignorieren.
Hinweis: Es ist nötig sich mit dem (gut kommentierten) Interface **`Maze`** vertraut zu machen, das im Framework enthalten ist. (12 P)

e) **Sonderaufgabe (für diese Teilaufgabe gibt es keine Punkte):**
- Machen Sie sich mit der Schnittstelle **`MazeGeneratorListener`** und erweitern Sie **`generateMaze()`** entsprechend, so daß relevante Ereignisse an den **`listener`**-Parameter gesendet werden. (Hierzu eignet sich eher der Generierungs-Algorithmus, der aus Teilaufgabe c) folgt.) Danach können Sie die Generierung von Labyrinthen im GUI beobachten!
- Spielen Sie mit den Kontrollparametern für die vertikale/horizontale Zerteilung des Labyrinths im Teile-und-Herrsche Algorithmus und beobachten Sie die Einfluß auf die Ästhetik der entstehenden Labyrinthe.

Aufgabe 2: *Iteratoren (14 P)*

a) Implementieren Sie einen Iterator für den Stack aus Aufgabe 4 auf Übungsblatt 5. Erweitern Sie hierzu die entsprechende Stack-Klasse um eine Methode **`public java.util.Iterator iterator()`**. Die Methode soll ein Iterator-Objekt liefern, mit dem man über die Stack-Elemente iterieren kann und zwar in der Reihenfolge, mit der die entspr. Elemente durch die **`pop()`** Methode vom Stack entfernt würden.
Implementieren Sie auch die Methode **`java.util.Iterator.remove()`** gemäß der API-Dokumentation, so daß die Methode nicht die Ausnahme **`UnsupportedOperationException`** wirft. (8 P)
Hinweise: Sie dürfen für die Implementierung sowohl den Code aus der Musterlösung von Übungsblatt 5 verwenden als auch Ihre eigene Implementierung.
Von dem Fall, daß der Stack durch **`push()`** oder **`pop()`** verändert wird, bevor die Zugriffe auf entsprechendes Iterator-Objekt abgeschlossen sind, dürfen Sie absehen.

b) Erweitern Sie die Testklasse aus Aufgabe 4 c) auf Übungsblatt 5 um einen Anweisungsüberdeckungtest für den Iterator, und testen Sie damit Ihre Implementierung aus a). (6 P)
Hinweis: Auch hier dürfen Sie für die Implementierung sowohl den Code aus der Musterlösung von Übungsblatt 5 verwenden als auch Ihre eigene Implementierung.

Aufgabe 3: *Korrektheit von Horners Regel (13 T)*

Zeigen Sie, daß das folgende Programm das Polynom $P(x) = \sum_{k=0}^{a.length-1} a[k]x^k$ an der Stelle x auswertet!

```
public class Horner {
  static double horner(double[] a, double x) {
    double y = 0;
    int i = a.length - 1;
    while (i >= 0) {
      y = a[i] + x * y;
      i = i – 1;
    }
    return y;
  }
}
```

Verwenden Sie hierzu den Ansatz der Schleifeninvariante und den wp-Kalkül und gehen Sie wie folgt vor:

a) Bestimmen Sie eine geeignete Vorbedingung P und eine geeignet Nachbedingung Q für die Methode **`horner()`**! (2 T)
b) Bestimme Sie eine geeignete die Schleifeninvariante I und zeigen Sie mit dem wp-Kalkül, daß diese korrekt ist (also daß die Invariante vor und nach der Schleife gilt). (7 T)
c) Folgern Sie aus I und der negierten Schleifenbedingung, daß am Ende Q gilt. (3 T)
d) Begründen Sie, warum das Programm terminiert und zeigen Sie damit die dessen totale Korrektheit. (1 T)