Übungen zur Vorlesung
Praktische Informatik II
Sommersemester 2003

Serie 5
(version 1.1)

Bäume und Graphen

Aufgabe 1: Breitensuche in Graphen (40%)

Das in der Vorlesung vorgestellte Verfahren der Breitensuche in Graphen soll in Java implementiert werden. Dabei soll die Klasse **Graph** eine Menge von Knoten verwalten, die wiederum ihre Transitionen als Nachbarschaftsliste verwaltet. Diese Klasse soll die Möglichkeit bieten, Knoten und Kanten einzufügen, sowie eine Methode
public void breitenSuche(IVertex v0, IProperty eigenschaft, IAction aktion) für den Suchalgorithmus. In dieser Methode werden Eigenschaften von Knoten überprüft, und Aktionen auf diesen durchgeführt, die anwendungsabhängig sind.
Deshalb soll das Interface **IVertex** dafür Methoden definieren: Mit
boolean hasProperty(IProperty eigenschaft) wird getestet, ob der Knoten **eigenschaft** besitzt,
void actOnNode(IAction aktion) soll angeben, das **aktion** auf dem Knoten auszuführen ist, und
String getNodeInfo() soll in einem String geeignete Information über den Knoten zurückgeben.
Da **IVertex** seine Transitionen selbst verwaltet, muss das Einfügen von Transitionen hier geeignet möglich sein.
Die konkreten Aktionen **CAction** erfüllen dann das Interface

```
public interface IAction {

  void run (IVertex vertex);
}
```

Damit diese vom Knoten ausgeführt werden können, muß **IVertex** eine entsprechende Methode spezifizieren. Als *Eigenschaften* betrachten wir Paare (Schlüssel, Wert), z.B. (FARBE, rot). Eure konkrete Vertexklasse soll dann folgendes leisten: Knoten sollen beliebig viele Eigenschaften verwalten. **getNodeInfo()** ählt diese als Ausgabe auf. Zur Evaluation eines Knotens soll **hasProperty(IProperty eigenschaft)** aufgerufen werden und testen, ob dieser Knoten die entsprechende Eigenschaft besitzt. Mit **actOnNode(IAction aktion)** wird dann die entsprechende Aktion auf diesem Knoten ausgeführt.
Eure Anwendung soll schliesslich einen Graphen erstellen, bei dem Knoten neben einem Namen auch eine Farbe haben. Als Test für Euer Paket kann dann etwa jeder grüne Knoten in einen roten umgefärbt werden, was durch geeignete Ausgaben dokumentiert wird.

Aufgabe 2: Balancierte Bäume – AVL-Bäume (60%)

Bei sortierter Speicherung von Daten bieten binäre Bäume gegenüber Listen potentiell den Vorteil, daß die Suche nach gespeicherten Datensätzen schneller ist, da in jedem Knoten entschieden werden kann, ob das gesuchte Datum im linken oder rechten Teilbaum zu finden sein könnte und dann der Inhalt des jeweils anderen Baums ignoriert werden kann. Dieser Vorteil geht jedoch verloren, falls der Baum nicht halbwegs gleichmäßig aufgebaut ist, d.h. wenn die Struktur sehr ungleichmäßig ist und das Abschneiden eines Teilbaumes den Suchraum nur marginal einschränkt – im Extremfall kann ein binärer Baum die Struktur einer Liste besitzen. Man spricht hierbei auch von degenerierten Bäumen.

Einen möglichen Ausweg aus diesem Problem bieten die 1962 von Adelson-Velskij und Landis entwickelten (höhen)balancierten Bäume (nach ihren Entwicklern auch kurz AVL-Bäume genannt). Ein Baum ist höhenbalanciert, wenn für jeden seiner Knoten gilt, daß der Unterschied zwischen der Höhe seines linken und der Höhe seines rechten Teilbaumes maximal 1 beträgt. Wählt man eine geklammerte Darstellung der Art (t_1, x, t_2) zur Repräsentation eines binären Baumes mit linkem Teilbaum t_1, rechtem Teilbaum t_2 und Knotendatum x, so läßt sich das Balanciertheitskriterium für einen Baum formal ausdrücken als

$$\begin{aligned}
\text{depthbal}(\varepsilon) &= \text{True} \\
\text{depthbal}((t_1, x, t_2)) &= \big(\,|\,\text{depth}(t_1) - \text{depth}(t_2)\,| \leq 1\big) \wedge \text{depthbal}(t_1) \wedge \text{depthbal}(t_2)
\end{aligned}$$

wobei ε wieder den leeren Baum bezeichnet und „depth“ wie üblich definiert ist als:

$$\begin{aligned}
\text{depth}(\varepsilon) &= 0 \\
\text{depth}((t_1, x, t_2)) &= \max(\text{depth}(t_1), \text{depth}(t_2)) + 1
\end{aligned}$$

Der von Adelson-Velskij und Landis entwickelte Algorithmus sieht vor, einen AVL-Baum nach jedem sortierten Einfügen eines neuen Datums zu rebalancieren, d.h. den Baum so zu modifizieren, daß das Balanciertheitskriterium wieder gilt. Dafür müssen (Teil-)Bäume rotiert werden, d.h. einer der Nachfolgeknoten wird neuer Wurzelknoten und die restlichen Teilbäume werden unverändert umarrangiert, so daß die Sortierung innerhalb des Baumes beibehalten wird und keine Daten verloren gehen. Die beiden Rotationsvorschriften sind:

$$\begin{aligned}
\text{rotr}(((t_1, y, t_2), x, t_3)) &= (t_1, y, (t_2, x, t_3)) \\
\text{rotl}((t_1, x, (t_2, y, t_3))) &= ((t_1, x, t_2), y, t_3)
\end{aligned}$$

Je nach entstehendem Ungleichgewicht in einem Baum nach dem Einfügen eines neuen Elements reicht es nicht aus, nur einfach eine der beiden Rotationen anzuwenden, um wieder einen sortierten, balancierten Baum zu erhalten. Um mit den verschiedenen Situationen umgehen zu können, definiert man als erstes die *Neigung* (engl. *slope*) eines Knotens, die das lokale Ungleichgewicht bezeichnet:

$$\begin{aligned}
\text{slope}(\varepsilon) &= 0 \\
\text{slope}((t_1, x, t_2)) &= \text{depth}(t_1) - \text{depth}(t_2)
\end{aligned}$$

Dies bedeutet, daß eine positive Neigung einer größeren Höhe auf der linken Seite entspricht. Da bei jedem Einfügen eines Elements in einen balancierten Baum die Höhen der Teilbäume nur maximal um den Wert 1 erhöht werden können, markiert ein slope-Wert von 2 oder -2 eine Situation, in der ein nicht-balancierter Baum entstanden ist, der durch entsprechende Rotation(en) wieder „repariert“ werden muß. Eine einzelne Rotation ist hierfür nicht immer ausreichend, da bei ungünstiger Struktur des Ausgangsbaums nur ein neuer nicht-balancierter Baum generiert wird. Die Ungleichgewichte müssen in allen Unterbäumen erst auf der richtigen Seite „aufgesammelt“ werden, damit eine Rotation auf nächsthöherer Ebene nicht die zu langen Ketten auf die jeweils andere Seite des Baumes kopiert und dort ein Ungleichgewicht erzeugt (siehe Fragestellung unten). Die folgenden Definitionen berücksichtigen dieses Vorgehen:

$$\text{shiftr}((t_1, x, t_2)) = \begin{cases} \text{rotr}((\text{rotl}(t_1), x, t_2)) & \text{falls slope}(t_1) = -1 \\ \text{rotr}((t_1, x, t_2)) & \text{sonst} \end{cases}$$

$$\text{shiftl}((t_1, x, t_2)) = \begin{cases} \text{rotl}((t_1, x, \text{rotr}(t_2))) & \text{falls } \mathit{slope}(t_2) = 1 \\ \text{rotl}((t_1, x, t_2)) & \text{sonst} \end{cases}$$

Das lokale Rebalancieren eines durch Einfügen modifizierten AVL-Baumknotens ist dann definiert als:

$$\text{rebal}(t) \;=\; \begin{cases} \text{shiftr}(t) & \text{falls } \mathit{slope}(t) = 2 \\ \text{shiftl}(t) & \text{falls } \mathit{slope}(t) = -2 \\ t & \text{sonst} \end{cases}$$

Wird ein neuer Wert in einen AVL-Baum eingefügt, so muß der resultierende Baum von unten nach oben entlang des Suchpfades für die Einfügeposition des neuen Elements rebalanciert werden, da auf allen Ebenen möglicherweise Modifikationen notwendig sind. Das Einfügen eines neuen Wertes x in einen sortierten AVL-Baum ist spezifiziert durch die folgende Definition:

$$\text{insert}(x, \varepsilon) \;=\; (\varepsilon, x, \varepsilon)$$

$$\text{insert}(x, (t_1, y, t_2)) \;=\; \begin{cases} \text{rebal}((\text{insert}(x, t_1), y, t_2)) & \text{falls } x < y \\ (t_1, y, t_2) & \text{falls } x = y \\ \text{rebal}((t_1, y, \text{insert}(x, t_2))) & \text{falls } x > y \end{cases}$$

Beachtet, daß bei dieser Version der AVL-Bäume keine Elemente doppelt innerhalb der Baumstruktur gespeichert werden.

Implementiert eine Klasse `AvlBaum` in Java, welche intern AVL-Bäume zur sortierten Speicherung von Strings verwendet. Die Strings sollen lexikographisch sortiert sein. Verwendet zum Vergleich zweier Strings die Methode `compareTo(String s)` der Klasse `String`.
Stellt in Eurer Klasse mindestens die folgenden drei Methoden zur Verfügung:

void einfuegen(String s): Fügt den String `s` sortiert in den AVL-Baum ein und rebalanciert ihn dabei wieder. Hier soll der interne Zustand des Objekts modifiziert werden, es wird *kein* modifizierter Baum zurückgeliefert.

boolean enthaelt(String s): Gibt genau dann `true` zurück, wenn der String `s` in dem sortierten Baum vorkommt.

void druckeStrukturiert(): Gibt die Struktur des AVL-Baumes in geklammerter Form auf dem Bildschirm aus, d.h. zum Beispiel ein Baum mit Inhalten „`Bier`“ und „`Cola`“ führt zu der Ausgabe `((-, Bier, -), Cola, -)`, falls die „`Cola`“ in der Wurzel gespeichert ist („`-`“ steht für den leeren Baum).

Die interne Datenstruktur Eurer AVL-Bäume soll dabei nach außen nicht sichtbar, d.h. vollständig gekapselt sein.
Hinweis: Um sowohl einfach auf den Baumstrukturen operieren zu können als auch die `einfuegen`-Methode objektorientiert mit Modifikation des internen Objektzustands umsetzen zu können, empfiehlt es sich, die `AvlBaum`-Klasse von einer internen `Knoten`-Klasse zur tatsächlichen Speicherung der Baumstruktur zu trennen.
Test: Gebt für die beim schrittweisen Einfügen der Strings „`Dill`“, „`Ei`“, „`Gans`“, „`Bier`“, „`Anis`“, „`Cola`“ und „`Feige`“ entstehenden AVL-Bäume nach jedem `einfuegen`-Aufruf die Baumstruktur mit Hilfe der `druckeStrukturiert`-Methode aus. Testet weiterhin wie üblich Randfälle und Sonderfälle.

Abgabe: 7.–10. Juli 2003 in den Übungen

Die Abgabe soll sowohl elektronisch (Programm-Quellcode) als auch in gedruckter Form (mit LaTeX gesetzter kommentierter und erläuterter Quellcode) erfolgen. Dabei ist auch auf geeignete Testfälle und deren Dokumentation zu achten!