Übungen zur Vorlesung
Praktische Informatik II
Sommersemester 2001

Serie 3

AVL–Bäume und Stapel

Aufgabe 1: Akrobatik: Balancierte Bäume (AVL-Bäume) (50%)

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–Velskii 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

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

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

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

Der von Adelson–Velskii 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:

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

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:

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

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:

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

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

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

$$rebal(t) = \begin{cases} shiftr(t) & \text{if } slope(t) = 2 \\ shiftl(t) & \text{if } slope(t) = -2 \\ t & \text{otherwise} \end{cases}$$

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:

$$insert(x, \varepsilon) = (\varepsilon, x, \varepsilon)$$

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

Beachten Sie, daß bei dieser Version der AVL–Bäumen keine Elemente doppelt innerhalb der Baumstruktur gespeichert werden.

Implementieren Sie eine Klasse `AVLTree` in Java, welche intern AVL–Bäume zur sortierten Speicherung von Integer–Zahlen verwendet. Stellen Sie in Ihrer Klasse mindestens die folgenden drei Methoden zur Verfügung:

void insert(int x): Fügt den Wert `x` 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 contains(int x): Gibt genau dann *True* zurück, wenn der Wert x in dem sortierten Baum vorkommt.

void printStructured(): Gibt die Struktur des AVL–Baumes in geklammerter Form auf dem Bildschirm aus, d. h. zum Beispiel ein Baum mit Inhalten 3 und 5 führt zu der Ausgabe `((-, 3, -), 5, -)`, falls die 5 in der Wurzel gespeichert ist (`-` steht für den leeren Baum).

Die interne Datenstruktur Ihrer 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 `insert`–Methode objektorientiert mit Modifikation des internen Objektzustands umsetzen zu können, empfiehlt es sich, die AVLTree–Klasse von einer internen Node–Klasse zur tatsächlichen Speicherung der Baumstruktur zu trennen (vgl. hierfür auch Balzert, S. 616ff). Weiterhin ist es bei dieser Aufgabe erlaubt, eine derartige Node–Klasse mit Methoden zu versehen, die explizit die Verkettung bestehender Knoten verändert (also z.B. `setLeft, setRight`–Methoden). Damit muß nicht mehr, wie bei Serie 2, jedesmal ein teilweise neuer Baum erzeugt werden, wenn sich die Struktur ändert (wie z.B. beim Rotieren) oder ein Element angefügt wird.
Test: Geben Sie für die beim schrittweisen Einfügen der Zahlenfolge `4,5,7,2,1,3,6` entstehenden AVL–Bäume nach jedem `insert` Aufruf die Baumstruktur mit Hilfe der `printStructured`–Methode wieder. Testen Sie weiterhin wie üblich Randfälle/Sonderfälle.

Aufgabe 2: Stapel (Stacks) ... und Roboter! (50%)

Die Idee bei der Speicherung von Daten in einem Stapel (engl. Stack) wird aus dem täglich Leben abgeleitet: Es ist möglich, oben auf einem Stapel neue Objekte zu plazieren, die dann den Zugriff auf bereits vorher vorhandene Objekte blockieren. Der Zugriff auf die Elemente eines Stapels ist immer nur von oben nach unten möglich, d.h. das zuletzt oben aufgelegte Objekte ist das, welches zuerst wieder entnommen werden muß, wenn man sich weiter nach unten voranarbeiten will. Bei einem Stapel handelt es sich damit um einen Datentyp, der eine Last–In–First–Out (LIFO) Strategie umsetzt. Die in der Informatikliteratur üblichen Bezeichnungen für das Hinzufügen und Entnehmen eines Objekts sind *push* und *pop*. Weiterhin ist es mit Hilfe einer Operation *top* möglich, das oben aufliegende Objekte zu inspizieren, ohne es vom Stapel herunterzunehmen.

Aufgabe 2a: Definieren Sie als Basisklasse für Objekte eines aufzubauenden Stacks eine abstrakte Klasse `StackItem` die eine abstrakte Methode `isGreaterThan` zum Vergleich von zwei `StackItem` Objekten zur Verfügung stellt:

boolean isGreaterThan(StackItem): liefert *True*, falls der Wert des aktuellen Objekts größer ist als der des übergebenen (hier nur abstrakte Methode, keine Implementierung).

Leiten Sie von `StackItem` mit Hilfe der Vererbung zwei Klassen `StringItem` und `IntItem` zur Speicherung von einer Zeichenkette bzw. einem Integer–Wert ab. Die `StackItem`–Klasse als Oberklasse soll es später über die Polymorphieeigenschaften von Java ermöglichen, Stacks über alle von `StackItem` abgeleiteten Klassen aufbauen zu können, ohne die Implementierung der eigentlichen Stackfunktionalität zu modifizieren. Um die Attribute der abgeleiteten Klassen zu setzen und auszulesen sind geeignete `set`– und `get`–Methoden sowie ein Konstruktor zu integrieren. Zusätzlich muß eine Redefinition der `isGreaterThan`–Methode in beiden abgeleiteten Klassen erfolgen.

Implementieren Sie darauf aufbauend eine Klasse `MyStack`, die folgende Operationen zur Verfügung stellt:

void push(StackItem x): Fügt dem Stack obenauf ein neues Objekt hinzu. Dabei soll nur der interne Zustand des Stacks modifiziert werden. Zusätzlich soll dafür gesorgt werden, daß der Stack Sortenrein bleibt, d.h. das Hinzufügen des neuen Elements x wird nur dann akzeptiert, wenn es ein Objekt der gleichen Klasse ist wie alle bisher auf dem Stack abgelegten Objekte. Bei einem leeren Stack wird mit dem ersten `push` der Typ der gestapelten Objekte definiert.

StackItem pop(): Entfernt das oberste Element des Stacks (interne Zustandsänderung) und liefert es als Ergebnis zurück.

StackItem top(): Liefert das oberste Element des Stacks zurück, ohne es zu entfernen.

boolean isEmpty(): Liefert einen booleschen Wert zurück, der genau dann `true` ist, wenn der Stapel leer ist.

Überlegen Sie sich dabei eine geeignete interne Datenstruktur zur Speicherung aller dem Stack hinzugefügten Daten.

Entwickeln Sie außerdem in Ihrem Programm eine statische Methode, um ein Textfile einzulesen und daraus zeilenweise passende StringItems oder IntItems zu generieren (je nach Inhalt des Textfiles), die einem gegebenen Stapel hinzugefügt werden. (Tip: Verwenden Sie hierbei `Integer.parseInt(String)` um einen String in eine Zahl umzuwandeln).

Hinweis: Zur Identifizierung, von welcher Klasse ein Objekt ist, ist das Prädikat `instanceof` oder auch die Methode `getClass()` in Kombination mit dem Attribut `class` hilfreich.

Aufgabe 2b: Betrachten Sie das folgende Szenario: In einem kleinen Unternehmen werden aus Kostengründen (Regale sind teuer ...) alle Lagerartikel einfach auf große Stapel gelegt. Um dennoch in der Lage zu sein, einen Artikel halbwegs leicht wiederzufinden, sind diese Stapel sortiert. Dabei sollen die größten Objekte immer unten angeordnet werden (sonst kippt der Stapel zu leicht um). Leider werden im Wareneingang neue Lieferungen immer in unsortierten Stapeln angeliefert. Glücklicherweise hat sich aber in der letzten Woche ein manisch–depressiver Roboter vorgestellt, der eigentlich nur auf das Ende der Welt wartet (nennen wir ihn Marvin). Er hat sich bereit erklärt, nebenbei auch noch (kostenlos) immer die Artikel der unsortierten Eingangsstapel unter Zuhilfenahme eines Hilfsstapels im Lager jeweils auf einem neuen Stapel sortiert aufzuschichten.

Schreiben Sie für Marvin ein Java–Programm, welches ihm sagt, wie seine Sortieraufgabe für einen gegebenen unsortierten Stapel durch hin– und herstapeln erledigt werden kann. Die grundlegende Idee besteht darin, daß der Eingangsstapel schrittweise abgebaut wird und auf den Hilfsstapel umgeschichtet werden kann, wobei man sich in einer internen Variablen (eine `StackItem` Referenz) merkt, welches bislang das größte Objekt ist. Wenn der Stapel komplett übertragen ist, so kennt man das momentane Maximum und kann es beim Zurückschichten des Hilfsstapels in den Eingangsbereich auf den sortierten Stapel legen. Dies wiederholt man so lange, bis der Eingangsstapel leer ist, der Lagerstapel ist dann sortiert. (Das Verfahren ist nicht das effizienteste, aber Marvin hat genug Zeit ...)

Die Wareneingangsstapel sollen aus einer Datei in einen `MyStack` eingelesen werden (siehe Aufgabe 2a). Der Algorithmus soll die Polymorphieeigenschaften von Java ausnutzen indem zum Vergleichen der Größe von Objekten die Methode `isGreaterThan` herangezogen werden soll, welche sowohl auf StringItems als auch auf IntItems gleichermaßen funktioniert. Der Sortieralgorithmus soll auf dem Bildschirm Anweisungen für Marvin ausgeben, was er tun soll, also z.B. „Objekt x: Eingangsstapel `->` Hilfsstapel“ (Die Angabe des Objekts ist nur eine Hilfsinformation zum Nachvollziehen des Algorithmus; durch die Struktur des Stapels ist dieses Objekt bei einer konkreten Stackbelegung immer eindeutig definiert).

Testen Sie Ihren Sortieralgorithmus sowohl mit Stapeln über Integer–Werten (IntItems) als auch mit Zeichenketten–Objekten (StringItems).

Abgabe: 30. Mai bis 1. Juni nach den jeweiligen Praktika. Die Abgabe soll sowohl elektronisch (Programm-Quellcode) als auch in gedruckter Form (mit LaTeX gesetzter kommentierter und erläuterter Quellcode) erfolgen!