RHEINISCH-
WESTFÄLISCHE
TECHNISCHE
HOCHSCHULE
AACHEN

Lehrstuhl für Hochleistungsrechnen
RWTH Aachen · D-52074 Aachen · GERMANY
http://programmierung.informatik.rwth-aachen.de

Prof. Christian Bischof, Ph.D.
Andre Vehreschild, Jakob T. Valvoda

Übung *Programmierung WS 06/07* – Blatt 7

Lösungen müssen bis zum **11. Dezember 2006, 17:00 Uhr** in den Kasten Ihrer Übungsgruppe eingeworfen werden. Sie finden die Kästen am Eingang Halifaxstr. des Informatikzentrums (Ahornstr. 55). Bitte vergessen Sie nicht, die **Nummer Ihrer Übungsgruppe**, sowie die **Namen** und **Matrikelnummern** der Mitglieder Ihrer 2er-Gruppe auf jedes Lösungsblatt Ihrer Abgabe zu schreiben. Bitte heften Sie ihre Blätter zusammen.

Alle erstellten Java-Programme sind sowohl in gedruckter Form abzugeben, als auch per E-Mail an den jeweiligen Tutor zu senden. Vergessen Sie auch bei der elektronischen Abgabe per E-Mail nicht die Angabe der **Nummer Ihrer Übungsgruppe**, sowie die jeweiligen **Namen** und **Matrikelnummern**.

Aufgabe 1 (4 + 1 = 5 Punkte)

In dieser Aufgabe sollen Sie einen rekursiven Algorithmus für das Sortieren von Arrays verwenden. Ein Ansatz besteht darin, das zu sortierende Feld in Bereiche aufzuteilen, welche mit einem rekursiven Aufruf der Sortiermethode bearbeitet werden. Nach Abarbeitung werden die jetzt sortierten Teile zusammengefügt. Dieses Prinzip wird allgemein „Teile und Herrsche“ bzw. „divide-and-conquer“ genannt.

a) Implementieren Sie einen rekursiven Algorithmus, der das obige Verfahren umsetzt. Als Eingabe soll Ihr Algorithmus ein `int`-Array $a_0, \ldots, a_{n-1}$ erwarten. Solange $n > 1$ ist, geht ihr Algorithmus wie folgt vor:

- Bestimme Teilungsindex $t = \lfloor \frac{n}{2} \rfloor$. Der Operator $\lfloor . \rfloor$ heisst untere Gauß-Klammer und rundet eine Zahl ab. Die untere Gauß-Klammer für eine Zahl $x \in \mathbb{R}$ ist definiert als $\lfloor x \rfloor = \max\{n \in \mathbb{N} \mid n \leq x\}$.
- Teile das Feld in zwei Bereiche $L = (a_0, \ldots, a_t)$ und $R = (a_{t+1}, \ldots, a_{n-1})$ auf.
- Wende den Algorithmus rekursiv auf beide Bereiche an. Dieser Schritt resultiert in zwei sortierten Feldern $L' = (a'_0, \ldots, a'_t)$ und $R' = (a'_{t+1}, \ldots, a'_{n-1})$,
- Füge beide Felder zusammen, so dass das Gesamtfeld ebenfalls sortiert ist. Dazu geht man beide Felder L' und R' von links durch und fügt jeweils das kleinere Element aus beiden Feldern in das Gesamtfeld ein. Das größere Element verbleibt in L' bzw. R' und wird in der nächten Iteration zum Vergleich herangezogen.

Falls das Feld nur ein Element enthält, dann ist dieses bereits sortiert und das (einelementige) Feld wird zurückgegeben.

b) Untersuchen Sie das Laufzeitverhalten Ihres Algorithmus, indem Sie eine Klasse `SortTestBed` implementieren, welche mehrere Felder beliebiger Länge (z.B. $n_1 = 1000, n_2 = 5000, n_3 =$

10000, ...) aus Zufallszahlen erzeugt und die für das Sortieren benötigte Zeit misst und ausgibt. Geben Sie die gemessenen Zeiten in einer Tabelle an. Die aktuelle Zeit in Millisekunden kann mit der Methode `long System.currentTimeMillis()` ermittelt werden. Für die Berechnung von Zufallszahlen können Sie eine Instanz der Klasse `Random` aus dem Paket `java.util` erzeugen. Ein Aufruf der Methode `int nextInt( int n )` dieser Instanz liefert dann jeweils die nächste Zufallszahl im Bereich $0, \ldots, \mathtt{n} - 1$.

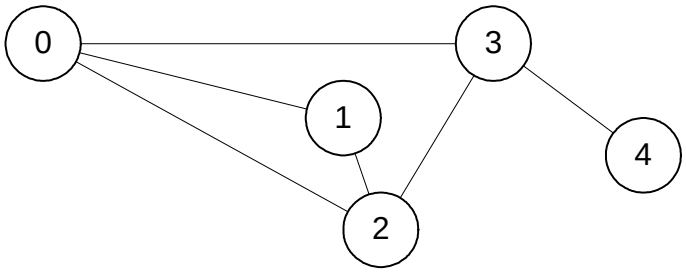
Hinweis: Sie können die Messung für große Arrays abbrechen, falls ihr Algorithmus länger als 30 Sekunden für das Sortieren benötigt. Sie können zusätzlich die Laufzeit Ihres rekursiven Algorithmus mit dem BubbleSort-Algorithmus vergleichen (dies fließt jedoch nicht in die Bewertung dieser Aufgabe ein).

Aufgabe 2 (4 + 4 + 2 = 10 Punkte)

Ein Graph $G = (V, E)$ besteht aus einer Menge Knoten V, welche durch Kanten $e \in E \subseteq V \times V$ verbunden werden. Zwei Knoten $v_i, v_j \in V$ sind demnach verbunden, falls $e_{v_i,v_j} = (v_i, v_j) \in E$. Graphen können durch Adjazenzmatrizen dargestellt werden. Hierbei repräsentiert jede Zeile und Spalte der Matrix A einen Knoten $v_i \in V$. Der Einfachheit halber wird davon ausgegangen, dass alle Knoten durchnummeriert sind. Die k-te Zeile und Spalte der Adjazenzmatrix wird dann mit dem Knoten $v_k \in V$ identifiziert. Sind zwei Knoten $v_i, v_j \in V$ verbunden (also Nachbarn), dann wird dies in der Matrix durch einen Eintrag gekennzeichnet:

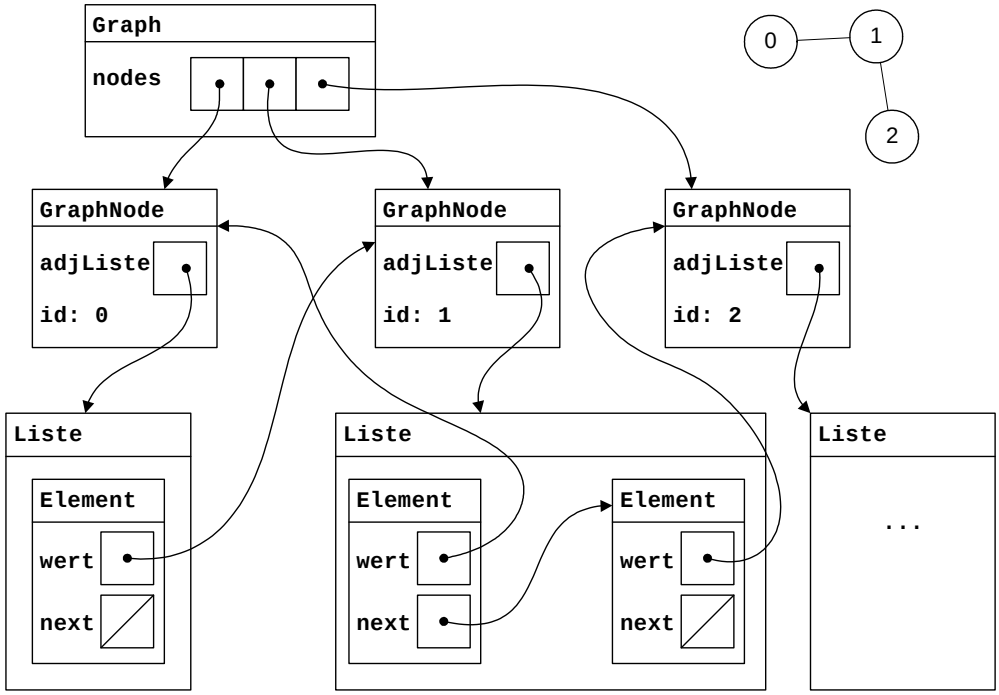
$$A = \left[a_{ij} \right] \qquad a_{ij} = \begin{cases} 1 & \text{falls} \quad e_{v_i,v_j} \in E \\ 0 & \text{sonst} \end{cases}$$

Beispiel: Der Graph $G = (V, E)$ mit $V = \{v_0, \ldots, v_4\}$ und die zugehörige Adjazenzmatrix.



	0	1	2	3	4
0	0	1	1	1	0
1	1	0	1	0	0
2	1	1	0	1	0
3	1	0	1	0	1
4	0	0	0	1	0

Ihre Aufgabe ist es, eine rekursive Datenstruktur für die Repräsentation eines Graphen $G = (V, E)$ zu implementieren. Hierzu müssen die Klassen `Graph` und `GraphNode` neu erstellt werden. Die Klassen `Liste` und `Element` aus der Vorlesung sind leicht zu modifizieren. In der folgenden Abbildung ist das Zusammenspiel der Klassen zur Speicherung des einfachen Graphen $G = (\{v_0, v_1, v_2\}, \{(v_0, v_1), (v_1, v_0), (v_1, v_2), (v_2, v_1)\})$ gegeben:



Ein Objekt der Klasse `Graph` speichert in einem Array alle `GraphNode`s. Dies entspricht der Menge der Knoten, in diesem Beispiel $V = \{v_0, v_1, v_2\}$. Die Klasse `GraphNode` enthält eine Liste `adjListe`, welche die Adjazenzbeziehung wiedergibt. Die Liste `adjListe` enthält z.B. für den obigen `GraphNode` v_1 zwei Elemente, welche Referenzen auf die Nachbarknoten v_0 und v_2 enthalten. Modifizieren Sie die in der Vorlesung vorgestellte Listen-Datenstruktur bestehend aus den Klassen `Element` und `Liste` dermaßen, dass Graphknoten anstatt Zahlen als Werte verarbeitet werden. Fügen Sie der Klasse `Liste` die Methode `Element getFirst()` hinzu, welche den Kopf der Liste zurückliefert.

a) Implementieren Sie die Klasse `GraphNode` und ihre Methoden

- `int getId()`, welche die Identifikationsnummer des Knoten liefert,
- `connectToNode( GraphNode k )`, welche den Knoten `k` in die Adjazenzliste der Knoten-Instanz einträgt,
- `GraphNode getFirstNeighbor()`, welche den ersten Nachbarn in der Adjazenzliste zurückliefert und
- `GraphNode getNextNeighbor()`, welche jeweils den nächsten Nachbarn liefert. Mit iterativen Aufrufen von `getNextNeighbor()` kann die Adjazenzliste durchlaufen werden, bis `getNextNeighbor()` den Wert `null` zurückliefert. Mit einem Aufruf von `getFirstNeighbor()` kann immer der Durchlauf wieder am Anfang begonnen werden.

  Hinweis: Fügen Sie der Klasse `GraphNode` ein Attribut hinzu, welches eine Referenz auf das zuletzt angeforderte Objekt speichert. Beim Aufruf von `getFirstNeigbor()` wird dieses auf das erste Element der Liste (kann mittels `getFirst()` geholt werden) gesetzt. Jeder Aufruf von `getNextNeigbor()` setzt den Wert entsprechend neu.

  Beispiel: Die Adjazenzliste enhält die Knoten v_0, v_5 und v_6. Die Aufrufe liefern

- `getFirstNeigbor()` $\rightarrow v_0$
- `getNextNeigbor()` $\rightarrow v_5$
- `getFirstNeigbor()` $\rightarrow v_0$

b) Implementieren Sie die Klasse `Graph` welche die Verwaltung des Graphen übernimmt. `Graph` soll ein Array aller Knoten des Graphen enthalten. Dieses Array soll am Anfang eine vordefinierte Größe besitzen und bei Bedarf vergrößert werden (vgl. die Stack-Implementierung in Übung 6, Aufgabe 4).

Implementieren Sie zunächst drei Konstruktoren:

- Den Standardkonstruktor `Graph()` der einen leeren Graphen initialisiert und die Größe des Knoten-Arrays auf 100 setzt,
- den Konstruktor `Graph( int )` der einen leeren Graphen initialisiert und die Größe des Knoten-Arrays auf den übergebenen Wert setzt, sowie
- den Konstruktor `Graph( boolean[][] )`, welcher als Parameter eine boolsche Adjazenzmatrix erwartet und die rekursive Datenstruktur erzeugt.

Die Größe des Arrays soll bei Bedarf jeweils verdoppelt werden. Das Array wird nicht verkleinert.

Implementieren Sie die folgenden Methoden der Klasse `Graph`

- `int newNode()` – erzeugt einen neuen Knoten und liefert seine Nummer zurück,
- `newEdge( int, int )` – erzeugt eine Kante zwischen den spezifizierten Knoten v_i und v_j. Dabei ist zu beachten, dass sowohl in der Adjazenzliste von v_i als auch in der Adjazenzliste von v_j die Kante eingetragen wird!
- `GraphNode getNode( int )` – liefert den angegebenen Knoten aus dem Knoten-Array.

c) Implementieren Sie in der Klasse `Graph` eine Methode `besuche( int )`, welche ausgehend von einem durch seinen Index spezifizierten Knoten v_i rekursiv den Graphen durchläuft und dabei jeweils die Identifikationsnummer des besuchten Knotens ausgibt.

Hierzu verfährt man wie folgt: Alle Knoten erhalten eine Markierung, welche initial nicht gesetzt ist. Wird ein noch nicht markierter Knoten besucht, dann wird er zuerst markiert. Danach werden rekursiv alle verbundenen Knoten in der Adjazenzliste besucht. Wird ein bereits markierter Knoten besucht, dann wird dessen Adjazenzliste nicht mehr durchlaufen.

Erweitern Sie hierzu Ihre Klasse `GraphNode` um ein Flag, d.h. eine boolsche Variable, welches anzeigt, ob ein Knoten bereits besucht wurde. Fügen Sie der Klasse `GraphNode` die Methoden `setVisitFlag( boolean )` und `boolean getVisitFlag()` hinzu, welche den Flag manipulieren. Implementieren Sie ebenfalls in der Klasse `Graph` die Methode `clearVisitFlag()`, welche das besagte Flag bei allen Knoten des Graphen auf den Wert `false` setzt.

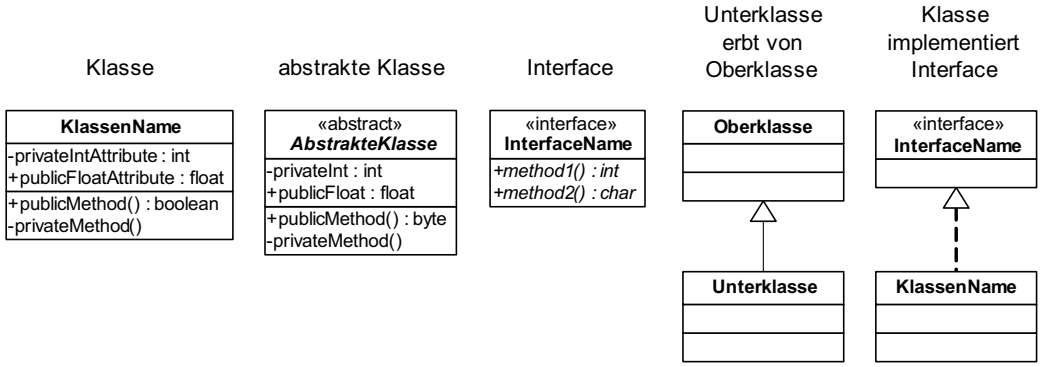
Beachten Sie, dass die Methode `clearVisitFlag()` das Knoten-Array nutzen kann (und muss). Die Methode `besuche( int )` hingegen greift auf das Array nur einmal zu, um

den Startknoten zu ermitteln. Danach nutzt die Methode ausschliesslich nur die jeweiligen Adjazenzlisten.

Aufgabe 3 (6 Punkte)

Ein System zur Überwachung von Fahrzeugen unterscheidet Flugzeuge, Helikopter, Züge, Straßenbahnen, Überlandbusse, Stadtbusse und Autos. Jedes Fahrzeug hat eine Identifikationsnummer und kann anhand der zuletzt übertragenen GPS-Koordinaten lokalisiert werden. Im Gegensatz zu Strassenfahrzeugen (Überlandbus, Stadtbus, Auto) wird die Fahrtroute von Luftfahrzeugen (Flugzeug, Helikopter) zusätzlich durch ihre Start- und Landeplätze (so z.B. Flughäfen oder Landeplätze bei Krankenhäusern) bestimmt. Schienenfahrzeuge (Zug, Straßenbahn) enhalten als Zusatzinformation die aktuelle Schienenstrecke. Diese Zusatzinformationen werden in `String`s gespeichert. Alle Strassenfahrzeuge enthalten die Information über ihre Höchstgeschwindigkeit und Busse zusätzlich die Angabe über die Passagierkapazität. Überlandbusse unterscheiden sich von Stadtbussen durch Bauart und sind für lange Fahrtrouten ausgelegt. Die Bauart wird durch eine Anzahl von Sternen als Komfort-Kategorie angegeben.

a) Ihre Aufgabe ist es eine Klassenhierarchie der oben beschriebenen Fahrzeuge zu entwerfen, welche in dem Überwachungssystem eingesetzt werden kann. Achten Sie darauf, gemeinsame Merkmale in (evtl. abstrakten) Oberklassen zusammenzufassen. Notieren Sie Ihren Entwurf als UML-Diagramm und verwenden Sie hierzu die folgende Notation (s. auch Aufgabe 4):

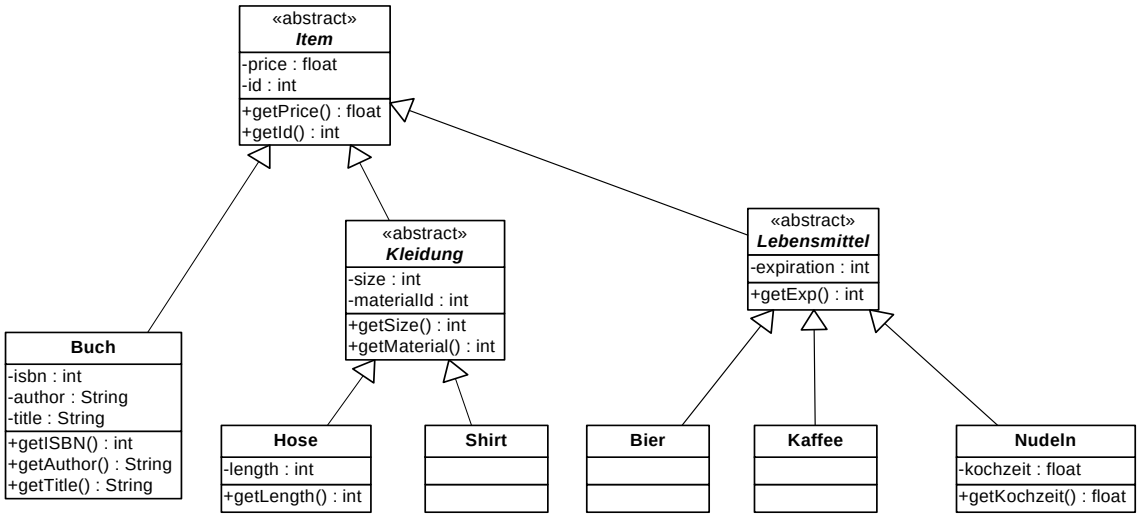


Geben Sie (wie in der obigen Vorlage) für jede Klasse ihren Namen und die Namen und Typen ihrer Attribute sowie die Signaturen der Methoden an. Gehen Sie davon aus, dass alle Informationen durch Basisdatentypen und den Typ `String` repräsentiert werden. Notieren Sie für jedes Attribut den zugehörigen Selektor.

b) Zu der obigen Klassifizierungen der Fahrzeuge werden die Informationen von lokalen Verkehrsunternehmen hinzugefügt. Diese Unternehmen betreiben Straßenbahnen und Stadtbusse. Die Fahrzeuge der Verkehrsunternehmens verfügen über eine genaue Auflistung der Haltestellen (als Array von Zeichenketten), welche von ihnen angefahren werden. Integrieren Sie diese Zusatzspezifikation in Ihre Lösung.

Aufgabe 4 (3 + 1 = 4 Punkte)

Eine Weiterentwicklung des von Ihnen in Übung 6, Aufgabe 3 implementierten Kassensystems nutzt eine Vererbungshierarchie für die Verwaltung von einzelnen Gegenständen. Die Klassenhierarchie ist im folgenden UML-Diagramm gegeben:



a) Implementieren Sie die Methode `float berechneSumme( Item [], float )` welche im ersten Parameter ein Array von gekauften Gegenständen und im zweiten Paramter den kundenspezifischen Rabatt übergeben bekommt und daraus die Brutto-Gesamtsumme berechnet. Verwenden Sie hierzu die Methode `float getPrice()` der einzelnen Gegenstände, welche den Netto-Preis zurückliefert. Berechnen Sie den Steuerzuschlag auf die Gegenstände und berücksichtigen Sie jeweils den Rabatt-Betrag des Kunden. Der Rabatt wird in Prozent angegeben und auf den Netto-Preis angewendet. Zusätzlich gilt, dass Bücher aufgrund der Buchpreisbindung grundsätzlich vom Rabatt ausgeschlossen sind. Die Steuerabgaben werden nach den folgenden Regeln berechnet:

- Auf Bücher und Lebensmittel (bis auf Bier und Kaffee) wird der reduzierte Mehrwertsteuersatz von 10% angewendet.
- Für Kleidung, Bier und Kaffee wird der vollen Steuersatz von 20% berechnet.
- Auf Kaffee wird zusätzlich noch ein Zuschlag von 5% auf den Endpreis erhoben.

Beispiel: Eine Kundin kauft ein Buch für 10 Euro, 4 Pakete Kaffee für insgesamt 10 Euro, eine Flasche Bier und eine Packung Nudeln für jeweils 1 Euro (alle Preise sind Netto). Die Kundin hat einen Rabatt von 10%. Sie zahlt für das Buch 11 Euro, für den Kaffee 11,34 Euro (10 Euro abzgl. Kundenrabatt von 10% + 20% Steuer und schliesslich zzgl. 5% Kaffee-Zuschlag), 0,99 Euro für die Nudeln und 1.08 für das Bier, also insgesamt 24,41 Euro.

Hinweis: Verwenden Sie für die Identifizierung der Klassen den Ausdruck `instanceof`.

b) Wie könnte die Berechnung der Gesamtsumme dahingehend vereinfacht werden, dass keine Aufrufe von `instanceof` benötigt werden? Diskutieren Sie Ihren Vorschlag.