FAU, Informatik 2, AUD-Team
aud@i2.cs.fau.de

6. Übung

Abgabe bis 05.12.2016, 10:00 Uhr

Einzelaufgabe 6.1: Verbunddatentypen — 29 EP

In dieser Aufgabe sollen Sie eine n-dimensionale Vektorklasse `VectorND` um die notwendigen Methoden für typische Vektor-Arithmetik erweitern. Laden Sie hierzu die Datei `VectorND.java` von der Übungsseite herunter. Sie können davon ausgehen, dass alle übergebenen Vektoren und Arrays *nicht* `null` sind. Deklarieren Sie keine eigenen Klassen- und Instanzvariablen.
Weiterführende Information zur Definition und Benutzung von Vektoren finden Sie unter `http://de.wikipedia.org/wiki/Vektor`
Implementieren Sie die folgenden Methoden:

`VectorND(int dim).` Implementieren Sie zunächst den Konstruktor der Klasse `VectorND`. Als Parameter wird die Anzahl der Dimensionen `dim` übergeben. Stellen Sie hier sicher, dass Sie den notwendigen Speicherbereich für die Dimensionen bereit stellen. Sie können annehmen, dass `dim` ausschließlich positive Werte (>0) annimmt.

`VectorND(double[] initData)` ist eine weitere Variante des Konstruktors, mit der zusätzlich ein Array aus Gleitkommazahlen dazu verwendet wird, die Daten im Vektor zu initialisieren. Beachten Sie, dass der Aufrufer das Array `initData` später ändern können soll, ohne dass dadurch der Vektor verändert wird.

`int getDimension()` gibt die Anzahl der Dimensionen des Vektors zurück.

`void init(double[] initData)` initialisiert den Vektor mit den übergebenen Daten im Array. Es soll keine Initialisierung durchgeführt werden, falls die Größe des Arrays nicht der Dimensionsgröße entspricht. Beachten Sie, dass der Aufrufer das Array `initData` später ändern können soll, ohne dass dadurch der Vektor verändert wird.

`void multiply(double m)` multipliziert den Vektor mit einem skalaren Wert `m`. Bei der skalaren Multiplikation eines Vektors wird jedes Element (jeder Dimension) mit dem skalaren Wert multipliziert.

`double dot(VectorND vec)` berechnet das Skalarprodukt zwischen dem Vektor und einem weiteren Vektor `vec`, und gibt dieses zurück. Das Skalarprodukt s zweier Vektoren $\vec{v_1}$ und $\vec{v_2}$ ist definiert durch

$$s = \sum_{i \in dim} \vec{v_1}[i] \cdot \vec{v_2}[i].$$

Bei falschen Dimensionen soll `NaN` zurueckgegeben werden.

`void add(VectorND vec)` addiert `vec` zum aktuellen Vektor. Der Summenvektor zweier Vektoren ergibt sich aus den Summen der Elemente der einzelnen Dimensionen:

$$\begin{bmatrix} \vec{v_1}[0] + \vec{v_2}[0] \\ \vec{v_1}[1] + \vec{v_2}[1] \\ \cdots \end{bmatrix}.$$

Bei falschen Dimensionen soll nichts gemacht werden.

`double norm()` berechnet die euklidsche Norm $\|\vec{v}\|$ (*Länge*) des Vektors $\vec{v}$ und gibt diese zurück. Sie kann mit Hilfe des Skalarprodukts bestimmt werden:

$$|\vec{v}| = \sqrt{\vec{v} \cdot \vec{v}}.$$

`void normalize()` nutzt die euklidsche Norm um die Länge des Vektors auf 1 zu normieren. Eine Normierung kann durchgeführt werden, indem jedes Element (jeder Dimension) durch die euklidsche Norm des Vektors geteilt wird.

Geben Sie Ihre Implementierung in der Datei `VectorND.java` über EST ab.
Beim Starten von `VectorND` wird folgende Ausgabe erzeugt:

```
# Vectors
-------------------------------------
VectorND with 2 dimensions: ( 1.000 , 2.000)
VectorND with 3 dimensions: ( 3.000 , 3.000 , 3.000)
Multiply v1 by 3.
VectorND with 2 dimensions: ( 3.000 , 6.000)
VectorND with 2 dimensions: ( 4.000 , 8.000)
Scalar product between v1 and v2: NaN
Scalar product between v1 and v3: 60.0
Scalar product between v3 and v1: 60.0
Normalize v2.
VectorND with 3 dimensions: ( 0.577 , 0.577 , 0.577)

-------------------------------------
# Finished Vectors
```

Gruppenaufgabe 6.2: $\mathcal{O}$-Kalkül für Methoden — 11 GP

Geben Sie zu jeder der folgenden 6 Methoden die *kleinste* obere Schranke im $\mathcal{O}$-Kalkül für die Laufzeit so an, dass sich das Ergebnis nicht mehr weiter vereinfachen lässt. Betrachten Sie dabei die Methodenargumente als „Problemgröße“ und nehmen Sie vereinfachend an, dass der Datentyp der verwendeten Variablen unbeschränkt ist sowie grundsätzlich keine Überläufe auftreten können. Geben Sie *jeweils* eine kurze *Begründung* für Ihre Einschätzung an.

a) `a(x)`


```
long a(int x) {
    long s = 0;
    for (int i = 1; i <= x; i++) {
        for (int j = 1; j <= x; j = j * 2) {
            s++;
        }
    }
    return s;
}
```

---

b) `b(x)`


```
long b(int x) {
    long s = 1, i = 1;
    do {
        s = s * 3;
        i++;
    } while (i <= x);
    while (s > 0) {
        i++;
        s--;
    }
    return 42;
}
```

---

c) `c(x)`

```
long c(int x) {
    long s = -x * x;
    while (s <= x * x * x) {
        s++;
    }
    for (long i = s * x; i > 0; i--) {
        s--;
    }
    return s;
}
```

X^2+x^3

(X^2+x^3)*x

X^2+x^3+(x^2+x^3)*x
=X^2+x^3+x^3+x^4
=X^4+2*x^3+x^4
=>x^4+x^3

---

d) `d(x,y)`

```
long d(int x, int y) {
    if (x <= 0) {
        return y;
    } else {
        return d(x - 1, 2 * y);
    }
}
```

O(x), x zählt einfach nur rekursiv herunter oder O(x^2) da es rekursiv passiert? keine Abhängigkeit vom Wert von y Vorschlag Patrick: O(y^2x)

---

e) `e(x,y)` mit `d(x,y)` aus Teilaufgabe d)

```
double e(int x, int y) {
    double s = 0.0D;
    for (long i = d(x, y); i >= 0; i--) {
        s++;
    }
    return s;
}
```

Ergebnis aus d) da d) aufgerufen wird, innerhalb von e) alleine wird i mal durchlaufen, also zusätzlich zum Ergebnis von d) noch *i

---

f) `f(x)`

```
char f(int x) {
    char h = 'h';
    for (long i = h; i-- > ++i; x--) {
        h++;
    }
    return h;
}
```

Wird in eclipse genau einmal durchlaufen und gibt h zurück ohne es zu erhöhen. O(1)

Geben Sie Ihre Lösung als `OKMeth.pdf` über EST ab.

Gruppenaufgabe 6.3: Referenzvariablen — 10 GP

Führen Sie die `main`-Methode der folgenden Klasse `Objekt` in einem Schreibtischlauf aus:

```
public class Objekt {
    private char x;
    private int y;

    Objekt(char x, int y) {
        this.x = x;
        this.y = y;
    }

    public Objekt vermenge(Objekt x) {
        this.x = x.x;
        this.y += x.y;
        return this;
    }

    public Objekt vermische(Objekt y) {
        return new Objekt(x, y.y);
    }

    public void kopiere(Objekt z) {
        z = this;
    }

    public void aendere(char x) {
        this.x = x;
    }

    public static void main(String[] args) {
        Objekt x, y = null, z = null;
        x = new Objekt('A', 1);
        /** 0 **/
        y = new Objekt('B', 13);
        /** 1 **/
        z = x.vermenge(y);
        /** 2 **/
        x = new Objekt('C', 42);
        /** 3 **/
        y = z;
        /** 4 **/
        z = z.vermische(x);
        /** 5 **/
        y = (new Objekt('D', 666)).vermenge(y);
        /** 6 **/
        x.kopiere(y);
        /** 7 **/
        z = x = y.vermische(z);
        /** 8 **/
        z.aendere('X');
        /** 9 **/
        y = x.vermische(y.vermenge(z.vermische(y)));
        /** 10 **/
    }
}
```

Beim Ausführen des Programms werden insgesamt acht Instanzen $O_0 - O_7$ der Klasse `Objekt` erzeugt. Geben Sie für jede der mit „`/** n **/`“ markierten Stellen an, welche Instanzen ($O_0 - O_7$) dort bekannt sind, welche Werte ihre Attribute `(x, y)` jeweils besitzen und auf welche Instanzen die Referenzvariablen `x`, `y` und `z` in der `main`-Methode jeweils verweisen. Nehmen Sie dabei an, dass es während der Ausführung *keine* Speicherbereinigung gibt. Verwenden Sie dazu eine Tabelle der folgenden Art, deren Anfang bereits exemplarisch vorgegeben ist:

`** n **`	$x_{\texttt{main}}$	$y_{\texttt{main}}$	$z_{\texttt{main}}$	O_0	O_1	O_2	O_3	O_4	O_5	O_6	O_7
`** 0 **`	O_0	`null`	`null`	(A,1)	–	–	–	–	–	–	–
...	...	...	...	...	...	...	...	...	...	...	...

Geben Sie Ihre Lösung als `Referenzvariablen.pdf` über EST ab.

Gruppenaufgabe 6.4: Speicherbelegung — 10 GP

Gegeben sei folgende Java-Klasse:

```
1  public class Klasse {
2      private int a;
3
4      public Klasse(int a) {
5          this.a = a;
6      }
7
8      public Klasse foo(Klasse i) {
9          return new Klasse(a - i.a);
10     }
11
12     public Klasse bar(Klasse i) {
13         a -= i.a + 7;
14         return this;
15     }
```

```
17     public Klasse baz(Klasse i, Klasse j) {
18         int k = 42;
19         Klasse t = new Klasse(k);
20         i = i.foo(t);
21         j = j.bar(i);
22         return j;
23     }
24
25     public void qux() {
26         int k = 666;
27         Klasse i = new Klasse(k);
28         Klasse j = new Klasse(4711);
29         i = baz(i, j);
30     }
31 }
```

Führen Sie die Methode `qux` in einem Schreibtischlauf aus und verfolgen Sie dabei die Programmstapel- und Speicherbelegung. Der Zustand beider Speicherarten unmittelbar *vor* dem Aufruf von `baz` in Zeile 29 sei wie in Abbildung 1 dargestellt. Im Stapel sind die Variablennamen sowie – je nach Variablentyp – deren Wert („#“) bzw. die Speicheradresse („▶“) des referenzierten Objekts angegeben. Im Speicher sind für alle Attribute der gespeicherten Objekte jeweils die aktuellen Werte aufgeführt (Referenzen sind in diesem Beispiel nicht möglich).
Zeichnen Sie *jeweils* den Zustand unmittelbar nach dem Ausführen der Zeilen 21 und 29 in der gleichen Darstellung wie in Abbildung 1. Nehmen Sie dabei an, dass der Garbage-Collector zu keinem Zeitpunkt aktiv ist. Der tatsächliche Speicherverbrauch soll ignoriert werden, d.h. jeder Typ verbraucht genau eine Speicherstelle. Die Speicherstellen sollen aufsteigend vergeben werden.

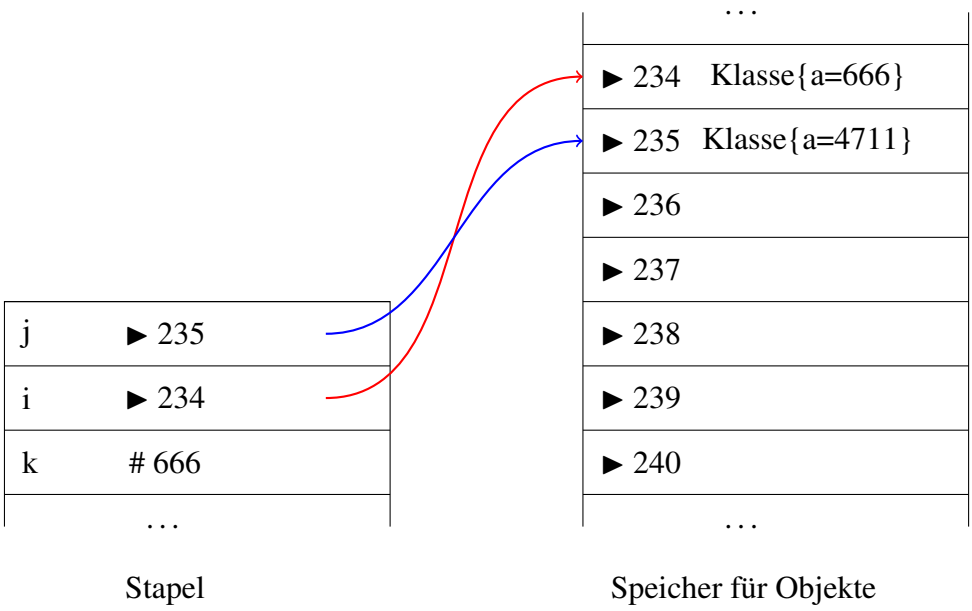

Abbildung 1: Belegung vor dem Aufruf der Methode `meth`.

Geben Sie Ihre Lösung als `Speicherbelegung.pdf` über EST ab.

29 EP + 31 GP = 60 Punkte