Stack

Seniorenseminar

21.06.2013

Michael Pohlig (pohlig@kit.edu)

Übersicht

1. Axiomatik eins Kellers und seine Software-Realisierung
2. Bedeutung der Rekursion in der Mathematik
3. Rekursive Programmierung.
4. Beispiele
   a. Mathematische Funktionen
   b. Rekursives Sortieren (Quicksort)
   c. Rekursive Geometrie (Fraktale)

1. Axiomatik eines Kellers und Softwarerealisierung

Die Mathematik kennt keine Begriffe mit individuellen Eigenschaften, sie kennt nur Relationen zwischen Begriffen. Diese Relationen werden durch die Axiome festgelegt.

Gottfried Falk (Physik Zahl und Realität)

1. Axiomatik eines Kellers und Softwarerealisierung

Ein **Keller** (oder **Stack** oder **Stapel**) ist eine (abstrakte) Datenstruktur. k ist die Menge der Zustände eines Kellers.

Die Elemente eines Kellers kürzen wir mit t ab.

Auf k werden 4 Operationen mit der folgenden Signatur definiert.

$$
\begin{array}{ll}
createK: & \{\emptyset\} \to k \\
push: & (k, t) \to k \\
pop: & k \to k \\
top: & k \to t \\
empty: & k \to \{true, false\}
\end{array}
$$

Es gelten folgende Axiome:

K1: $empty(createK) = true$

K2: $empty\big(push(k, t)\big) = false$

K3: $pop\big(push(k, t)\big) = k$

K4: $top\big(push(k, t)\big) = t$

1. Axiomatik eines Kellers und Softwarerealisierung

Ein **Keller** (oder **Stack** oder **Stapel**) ist eine (abstrakte) Datenstruktur. k ist die Menge der Zustände eines Kellers.

Die Elemente eines Kellers kürzen wir mit t ab.

Auf k werden 4 Operationen mit der folgenden Signatur definiert.

$createK$:	$\{\emptyset\} \to k$	Erzeugt einen leeren Keller.
$push$:	$(k, t) \to k$	Legt Element auf den Keller (Stapel).
pop:	$k \to k$	Entfernt das oberste Element des Kellers.
top:	$k \to t$	Zeigt das zuletzt eingekellerte Element.
$empty$:	$k \to \{true, false\}$	Sagt ob der Keller leer ist oder nicht.

Es gelten folgende Axiome:

K1: $empty(createK) = true$

K2: $empty(push(k, t)) = false$

K3: $pop(push(k, t)) = k$

K4: $top(push(k, t)) = t$

Die Bedeutungen der Operationen erschließen sich aus den Axiomen, nicht aus den Namen der Operationen.
Die Namen der Operationen sind so gewählt, dass man die Bedeutung erkennt.

1. Axiomatik eines Kellers und Softwarerealisierung

Ein **Keller** (oder **Stack** oder **Stapel**) ist eine (abstrakte) Datenstruktur. k ist die Menge der Zustände eines Kellers.

Die Elemente eines Kellers kürzen wir mit t ab.

Auf k werden 4 Operationen mit der folgenden Signatur definiert.

$createK$:	$\{\emptyset\} \to k$	Erzeugt einen leeren Keller.
$push$:	$(k, t) \to k$	Legt Element auf den Keller (Stapel).
pop:	$k \to k$	Entfernt das oberste Element des Kellers.
top:	$k \to t$	Zeigt das zuletzt eingekellerte Element.
$empty$:	$k \to \{true, false\}$	Sagt ob der Keller leer ist oder nicht.

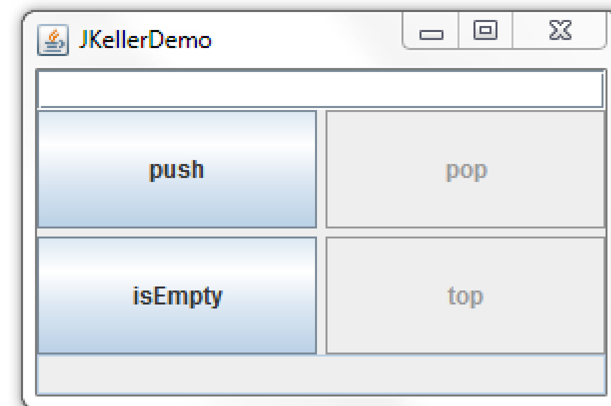
Es gelten folgende Axiome:

K1: $empty(createK) = true$

K2: $empty\big(push(k,t)\big) = false$

K3: $pop\big(push(k,t)\big) = k$

K4: $top\big(push(k,t)\big) = t$



2. Bedeutung der Rekursion in der Mathematik

Was ist $n!$?

$$1 \cdot 2 \cdot 3 \cdot \dots \cdot n$$

?

$0! = 1$?

$$1 \cdot 2 \cdot 3 \cdot 1 \cdot 2 \cdot 3 \cdot 4 \cdot \dots \cdot n$$

$$n! = n \cdot (n-1)! \ und \ 0! = 1$$

rekursiver Abstieg – Basisfall – rekursiver Aufstieg

$$4! = 4 \cdot 3! \qquad = 4 \cdot 6 = 24$$

$$3! = 3 \cdot 2! \qquad = 3 \cdot 2 = 6$$

$$2! = 2 \cdot 1! \qquad = 2 \cdot 1 = 2$$

$$1! = 1 \cdot 0! \qquad = 1 \cdot 1 = 1$$

$$0! = 1$$

Bei jedem Abstiegsschritt muss die Berechnung „einfacher“ werden

2. Bedeutung der Rekursion in der Mathematik

Was ist ***n*!**?

$$1 \cdot 2 \cdot 3 \cdot \dots \cdot n$$

? $0! = 1?$

$$1 \cdot 2 \cdot 3 \cdot 1 \cdot 2 \cdot 3 \cdot 4 \cdot \dots \cdot n$$

$$n! = n \cdot (n-1)! \; und \; 0! = 1$$

rekursiver Abstieg – Basisfall – rekursiver Aufstieg

$$4! = 4 \cdot 3! \qquad = 4 \cdot 6 = 24$$

$$3! = 3 \cdot 2! \qquad = 3 \cdot 2 = 6$$

$$2! = 2 \cdot 1! \qquad = 2 \cdot 1 = 2$$

$$1! = 1 \cdot 0! \qquad = 1 \cdot 1 = 1$$

$$0! = 1$$

Das Beispiel zeigt, dass das explizite, d.h. auf direktem Weg unzugängliche Unendliche, auf implizite (hier rekursive) Weise dennoch finit fassbar wird. (nach Falk).

3. Rekursive Programmierung

```
public class RekursionDemo{
    public static void main(String[] args){
       rekursion(5);
    }
    private static void rekursion(int a){
       a--;
       System.out.println(a);
       if (a!=0) rekursion(a);
       System.out.println(a);
   }
}
```

```
public class RekursionDemo{
    public static void main(String[] args){
       Aufruf rekursion(5);
    }
  rekursion(a){
       erniedrigt den Wert von a um 1
       gibt a aus
       wenn (a nicht 0) rekursion(a);
       gibt a aus
   }
}
```

Bei jedem Aufruf der Methode rekursion (Abstieg), wird eine Kopie dieser Methode mit ihren aktuellen Werten auf den Stack gelegt (***push***).
Beim rekursiven Aufstieg wird aktuelle auf dem Stack liegende (***top***) Methode zu Ende geführt. Danach wird sie vom Stack entfernt (***pop***).
Dies geschieht solange der Stack nicht leer ist (***empty*** liefert false) ist.

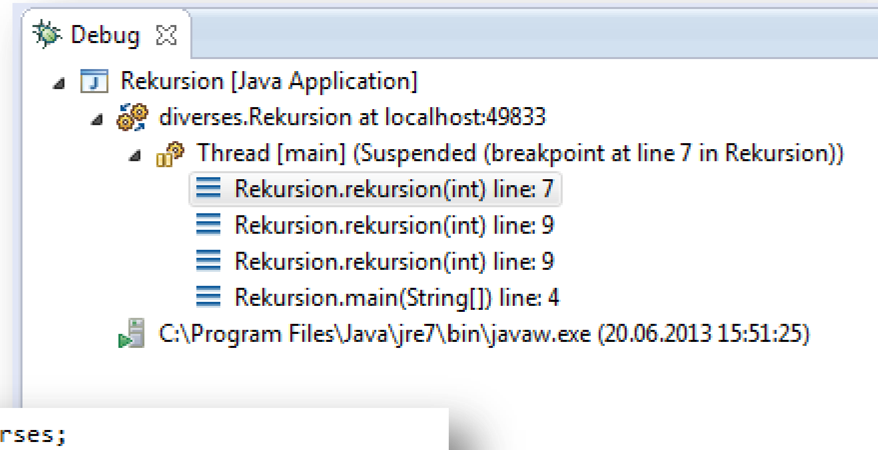
3. Rekursive Programmierung

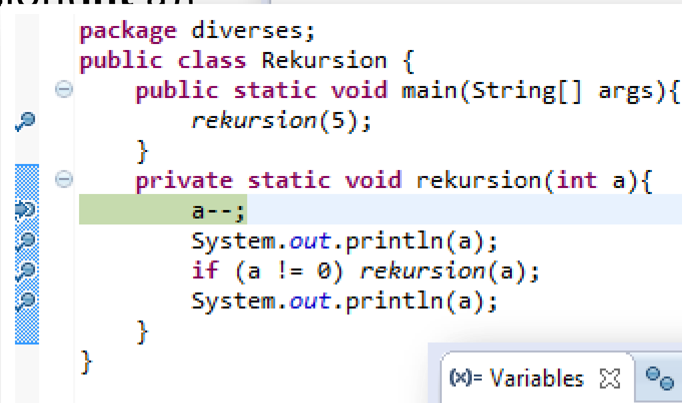
```
public class RekursionDemo{
    public static void main(String[] args){
       rekursion(5);
    }
    private static void rekursion(int a){
       a--;
       System.out.println(a);
       if (a!=0) rekursion(a);
       System.out.println(a);
   }
}
```

4
3
2
1
0
0
1
2
3
4

Bei jedem Aufruf der Methode rekursion (Abstieg), wird eine Kopie dieser Methode mit ihren aktuellen Werten auf den Stack gelegt (***push***).
Beim rekursiven Aufstieg wird aktuelle auf dem Stack liegende (***top***) Methode zu Ende geführt. Danach wird sie vom Stack entfernt (***pop***).
Dies geschieht solange der Stack nicht leer ist (***empty*** liefert false) ist.

```
public class RekursionDemo{
    public static void main(String[] args){
        rekursion(5);
    }
    private static void rekursion(int a){
        a--;
        System.out.println(a);
        if (a!=0) rekursion(a);
        System.out.println(a);
    }
}
```





Bei jedem Aufruf der Methode rekursion (Abstieg ... de
mit ihren aktuellen Werten auf den Stack gelegt (...
Beim rekursiven Aufstieg wird aktuelle auf dem S...
Ende geführt. Danach wird sie vom Stack entfern...
Dies geschieht solange der Stack nicht leer ist (*empty* liefert false) ist.

4 Beispiele a) Mathematische Funktionen

Brute-force Algorithmus

```
public class ggTit {
      public static void main(String[] args) {
            System.out.println(ggT(1620197464, 1157283932));
      }

      private static int ggT(int a, int b) {
            int ggT = 1;
            for (int i = 1; i <= Math.min(a, b); i++) {
               if ((a % i == 0) && (b % i == 0))
               ggT = i;
            }
            return ggT;
      }
}
```

4 Beispiele a) Mathematische Funktionen

ggT(792,75) ?

ggT(a,b) = ggT(b,a mod b)

$$792 = 10 \cdot 75 + 42$$
$$75 = 1 \cdot 42 + 33$$
$$42 = 1 \cdot 33 + 9$$
$$33 = 3 \cdot 9 + 6$$
$$9 = 1 \cdot 6 + 3$$
$$6 = 2 \cdot \mathbf{3} + 0$$

```
public class ggTrek {
    public static void main(String[] args){
        System.out.println(ggT(1620197464, 1157283932));
    }
    private static int ggT(int a, int b){
        if(a==b||b==0) return a;
        else return ggT(b,a%b);
    }
}
```

4 Beispiele a) Mathematische Funktionen

Fibonacci-Zahlen

$$1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

$$fib(1) = fib(2) = 1$$
$$\forall n \geq 3: fib(n) = fib(n-1) + fib(n-2)$$

```
private static int fibonacci(int a){
  if (a==1||a==2) return 1;
  else return fibonacci(a-1)+fibonacci(a-2);
}
```

4 Beispiele a) Mathematische Funktionen

<table>
<tr><td colspan="13">fib(6)</td></tr>
<tr><td colspan="8">fib(5)</td><td colspan="5">fib(4)</td></tr>
<tr><td colspan="5">fib(4)</td><td colspan="3">fib(3)</td><td colspan="3">fib(3)</td><td colspan="2">fib(2)</td></tr>
<tr><td colspan="3">fib(3)</td><td colspan="2">fib(2)</td><td colspan="2">fib(2)</td><td>fib(1)=1</td><td colspan="2">fib(2)</td><td>fib(1)=1</td><td>fib(1)=1</td><td>fib(0)=1</td></tr>
<tr><td colspan="2">fib(2)</td><td>fib(1)=1</td><td>fib(1)=1</td><td>fib(0)=1</td><td>fib(1)=1</td><td>fib(0)=1</td><td></td><td>fib(1)=1</td><td>fib(0)=1</td><td></td><td></td><td></td></tr>
<tr><td>fib(1)=1</td><td>fib(0)=1</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

					1	**0**																				
				2	2	2	1		1	0			1	0					1	0						
			3	3	3	3	3	2	2	2		2	2	2	1			2	2	2	1			1	0	
		4	4	4	4	4	4	4	4	4	3	3	3	3	3		3	3	3	3	3		2	2	2	
	5	5	5	5	5	5	5	5	5	5	5	5	5	5	**5**	4	4	4	4	4	4	4	4	4	**4**	
6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	**6**

4 Beispiele a) Mathematische Funktionen

```
for (long fib1 = 1, fib2 = 1, i=3; i <= a; i++){
      fib =  fib1 + fib2;
      fib1 = fib2;
      fib2 = fib;
}
```

$1, 1, 2, \mathbf{3}, \mathbf{5}, \mathbf{8}, 13, 21, 34, \ldots$

$1, 1, 2, 3, \mathbf{5}, \mathbf{8}, \mathbf{13}, 21, 34, \ldots$

fib1 (Speicher)	fib2 (Speicher2)	fib (Speicher3)
3	5	3+5 = 8
5	8	13

Geht sicher schneller

4 Beispiele a) Mathematische Funktionen

```
for (long fib1 = 1, fib2 = 1, i=3; i <= a; i++){
      fib =  fib1 + fib2;
      fib1 = fib2;
      fib2 = fib;
}
```

$1, 1, 2, \mathbf{3}, \mathbf{5}, \mathbf{8}, 13, 21, 34, \ldots$

$1, 1, 2, 3, \mathbf{5}, \mathbf{8}, \mathbf{13}, 21, 34, \ldots$

fib1 (Speicher)	fib2 (Speicher2)	fib (Speicher3)
3	5	3+5 = 8
5	8	13

$$fib(n) = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}}$$

Noch schneller

4 Beispiele b) Rekursives Sortieren (Quicksort)

76	7	58	88	60	41	82	77	49	86
49	7	58	41	60	88	82	77	76	86
7	49	58	41	60	76	77	82	88	86

4 Beispiele b) Rekursives Sortieren (Quicksort)

```
private static void quickSort(int[] liste, int untereGrenze, int obereGrenze) {
    int links = untereGrenze;
    int rechts = obereGrenze;
    int pivot = liste[((untereGrenze + obereGrenze) / 2)];
    do {
        while (liste[links] < pivot) {
            links++;
        }
        while (pivot < liste[rechts]) {
            rechts--;
        }
        if (links <= rechts) {
            int tmp = liste[links];
            liste[links] = liste[rechts];
            liste[rechts] = tmp;
            links++;
            rechts--;
        }
    } while (links <= rechts);
    if (untereGrenze < rechts) liste = quickSort(liste, untereGrenze, rechts);
    if (links < obereGrenze) liste = quickSort(liste, links, obereGrenze);
}
```

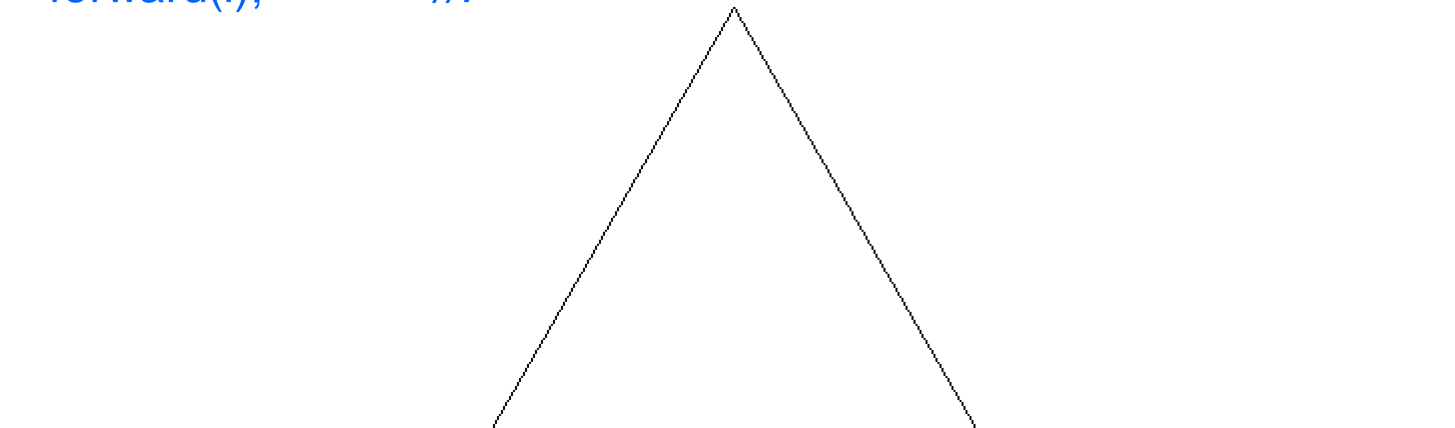
4 Beispiele c) Rekursives Geometrie (Fraktale)

Koch-Kurve

Strecke dritteln und Dreieck darüber zeichnen mit der Vorschrift:

```
forward(l);     //F
left(60);       //+
forward(l);     //F
right(120);     //--
forward(l);     //F
left(60);       //+
forward(l);     //F
```

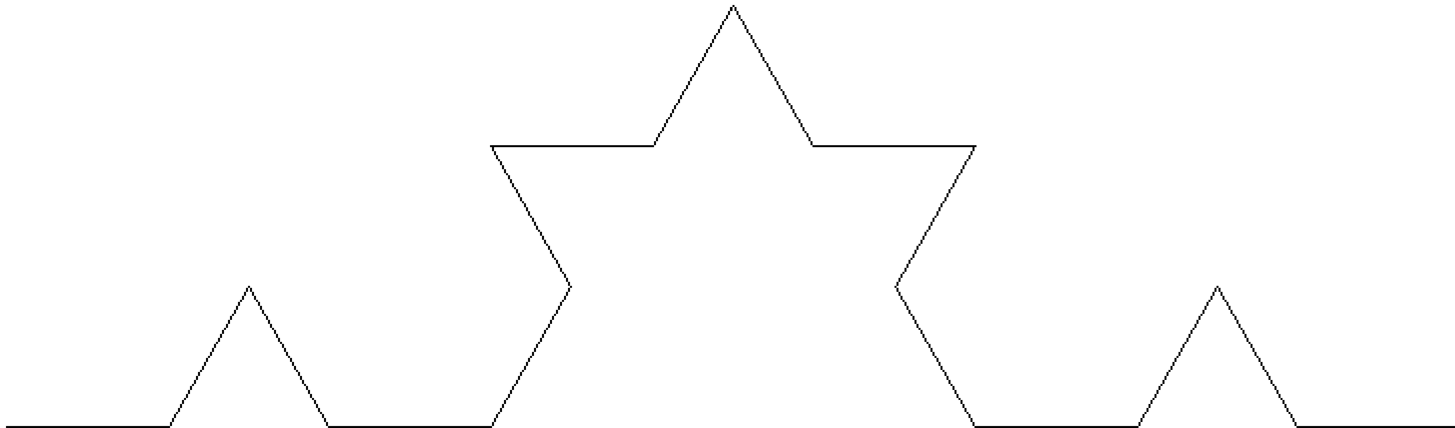
L:= F - F + + F - F



Koch-Kurve

Streckenlänge dritteln

L - L + + L - L mit L:= F - F + + F - F



Weiter rekursiv (Strecke immer dritteln)

4 Beispiele c) Rekursives Geometrie (Fraktale)

Koch-Kurve

```
forward(l);        //F
left(60);          //+
forward(l);        //F
right(120);        //--
forward(l);        //F
left(60);          //+
forward(l);        //F
```

```
public void zeichneKochkurve(double l){
    if(l<2) forward(l);
    else {
        zeichneKochkurve(l/3);   //L
        left(60);                //+
        zeichneKochkurve(l/3);   //L
        right(120);              //--
        zeichneKochkurve(l/3);   //L
        left(60);                //+
        zeichneKochkurve(l/3);   //L
    }
}
```