



Abstrakte Datentypen

Tobias Scheffer

Praktische Informatik 1

- Aufbau und Funktion von Computern.
- Unix, Dateisysteme, Werkzeuge.
- Maschinenprogrammierung, Ausführung von Maschineninstruktionen.
- Objektorientierung.
- Konzepte von Programmiersprachen, primitive Datentypen.
- Strukturierte Datentypen, Arrays, Strings.
- Abstrakte Datentypen, Stacks, Queues, Listen.

Praktische Informatik 1

- Java-API, Interaktion.
- Rekursion und Induktion.
- Suchen.
- Datenstrukturen: Hashes, Bäume.
- Sortieralgorithmen: Merge-Sort, Quicksort.
- Software-Entwurf.
- Wrap-Up.

Call by Value / Call by Reference

```
public class ValueReference
{
    public int Array[] = {1, 2, 3, 4, 5};

    public void Ändern1 (int Wert)
    {
        Wert = 7;
    }

    public void Ändern2 (int A[])
    {
        int A2[] = {1, 2, 7, 4, 5};
        A = A2;
    }

    public void Ändern3 (int A[])
    {
        A[2] = 7;
    }

    public ValueReference ()
    {
        System.out.println("Array[2]= " + Array[2]);
        Ändern1 (Array[2]);
        System.out.println("Array[2]= " + Array[2]);
        Ändern2 (Array);
        System.out.println("Array[2]= " + Array[2]);
        Ändern3 (Array);
        System.out.println("Array[2]= " + Array[2]);
    }
}
```

Abstrakte Datentypen

- Komplexe Wechselwirkungen zwischen Objekten
 - ◆ Ein Objekt verändert die Attribute von Objekten anderer Klassen
- machen Software schwer verständlich und Fehlersuche schwierig.
- Idee: Jede Klasse soll anderen Klassen Methoden zur Verfügung stellen.
- Nur die Methoden der Klasse sollen die Attribute eines Objektes verändern dürfen.
- Dadurch saubere Kapselung, Programme einfacher zu verstehen und zu debuggen.

5

Tobias Scheffer

Abstrakte Datentypen

- Besteht aus:
 - ◆ Attributen,
 - ◆ Methoden, die auf diese Attribute angewendet werden können.
 - ◆ Axiomen: Mathematischen Aussagen über Objekte, die immer gelten sollen.
- Axiome beschreiben gewünschtes Verhalten der Objekte. Sollen immer gelten.
 - ◆ Unterschied zu Invarianten: Invarianten sind Positionen im Programmtext zugeordnet.
 - ◆ Werden als logische Formeln ausgedrückt.

6

Tobias Scheffer

Abstrakte Datentypen

- Axiome sind mathematische Aussagen über Speicherzustände.
- Ein Speicherzustand ist die Belegung aller Variablen zu einem Zeitpunkt.
- Ein Speicherzustand ist eine Abbildung von Bezeichnern auf Werte aus deren Datentyp.
- Notation für Speicherzustände:
 - ◆ Wenn x ein Bezeichner ist, dann schreibe ich
 - ◆ $[s](x)$
 - ◆ für den Wert der von x bezeichneten Variable im Zustand s_p

7

Tobias Scheffer

Klasse Stack

- Beispiel für einen abstrakten Datentypen.
- Ein Stack ist ein Stapelspeicher:
 - ◆ Man kann oben etwas auf den Stapel legen (push),
 - ◆ man kann das oberste Element vom Stapel nehmen (pop).
 - ◆ Ein Stapel wächst beliebig hoch.
- Unterschied Array:
 - ◆ Array hat feste Länge,
 - ◆ man kann auf jedes Element des Arrays zugreifen, nicht nur das oberste.

8

Tobias Scheffer

Stack

- Ein Stack ist eine unglaublich nützliche Klasse.
 - ◆ Immer wenn ich bei der Arbeit von etwas Dringendem unterbrochen werde, dann lege ich das was ich gerade tue auf den Stack (Schreibtisch) und fange mit der neuen Aufgabe an.
 - ◆ Wenn ich dabei von etwas Dringendem unterbrochen werde...
 - ◆ Wenn ich nichts zu tun habe, nehme ich das oberste Element von meinem Schreibtisch und arbeite dran.
- Ein Stack hilft beim Verstehen geschachtelter Relativsätze.
- LIFO: Last in, first out.

9

Tobias Scheffer

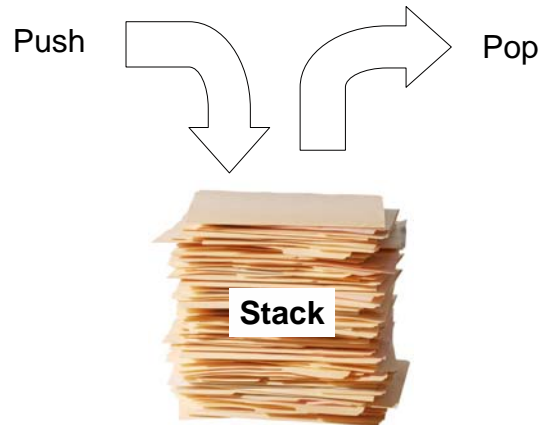
Stack

- Der Java-Interpreter benutzt einen Stack um Methodenaufrufe zu verarbeiten.
 - ◆ Der Speicher für lokale Variablen liegt auf einem Stack.
 - ◆ Wenn eine Methode aufgerufen wird, dann werden die lokalen Variablen dieser Methode darüber gestapelt, die Variablen der aufrufenden Methode werden davon verdeckt.
 - ◆ Wenn die aufgerufene Methode fertig ist, werden ihre lokalen Variablen vom Stack geworfen, die lokalen Variablen der aufrufenden Methode erscheinen wieder oben.

10

Tobias Scheffer

Stack



11

Stack

- Attribute:
 - ◆ OberstesElement.
 - ◆ [Irgendwo müssen auch die Elemente darunter gespeichert werden].
- Methoden:
 - ◆ **Stack.Push(Element)** → verändert Stack. Neues Element auf den Stack packen.
 - ◆ **Stack.Pop** → liefert Element, verändert Stack. Element vom Stack entfernen und zurück liefern.
 - ◆ **Stack.IsEmpty** → liefert Boolean. Test, ob der Stack leer ist; liefert true oder false.

12

Stack: Axiome

Zustand s_0 .

- `Stack.Push(e1); e2=Stack.Pop ()`

Zustand s_1 .

Hier gilt: $[s_0](\text{Stack}) = [s_1](\text{Stack})$,
 $[s_1](e2) = [s_1](e1)$

13

Stack: Axiome

Zustand s_0 .

- `e1=Stack.Pop ();`

Zustand s_1 .

Wenn $[s_0](\text{Stack.IsEmpty}()) = \text{true}$, dann gilt hier
 $[s_1](e1) = \text{null}$

14

Stack: Axiome

Zustand s_0 .

- `e1=Stack.Pop (); Stack.Push(e1);`

Zustand s_1 .

Wenn $[s_0](\text{Stack.IsEmpty}()) = \text{false}$, dann gilt hier
 $[s_1](\text{Stack}) = [s_0](\text{Stack})$

15

Stack: Axiome

Zustand s_0 .

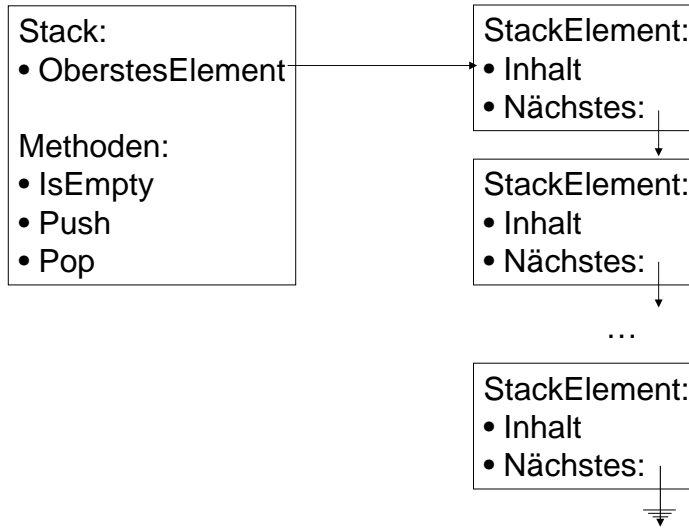
- `Stack.Push(e1);`

Zustand s_1 .

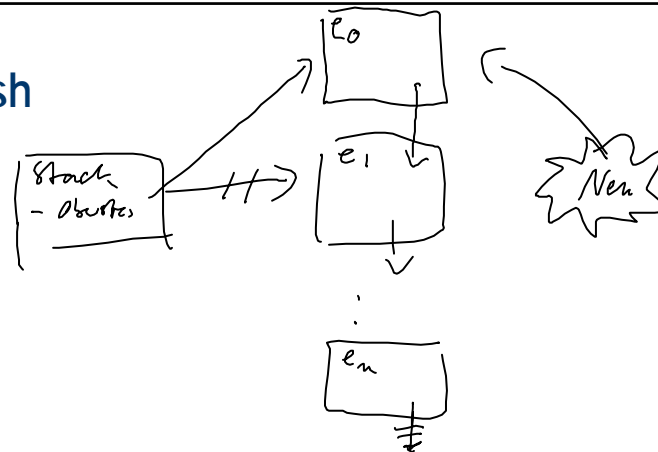
Hier gilt: $[s_1](\text{Stack.IsEmpty}()) = \text{false}$

16

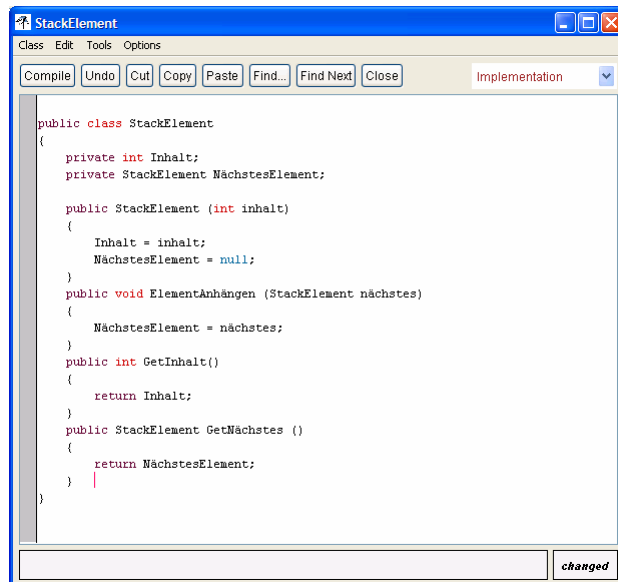
Stack



Push



StackElement



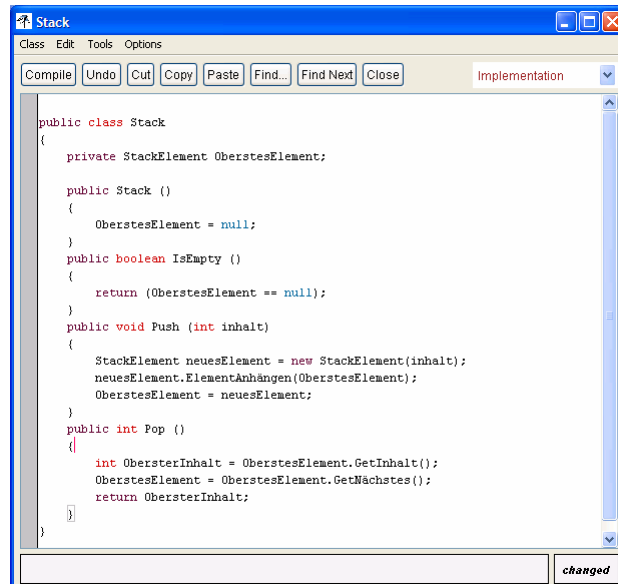
```
public class StackElement
{
    private int Inhalt;
    private StackElement NächstesElement;

    public StackElement (int inhalt)
    {
        Inhalt = inhalt;
        NächstesElement = null;
    }
    public void ElementAnhängen (StackElement nächstes)
    {
        NächstesElement = nächstes;
    }
    public int GetInhalt()
    {
        return Inhalt;
    }
    public StackElement GetNächstes ()
    {
        return NächstesElement;
    }
}
```

19

Tobias Scheffer

Stack



```
public class Stack
{
    private StackElement OberstesElement;

    public Stack ()
    {
        OberstesElement = null;
    }
    public boolean IsEmpty ()
    {
        return (OberstesElement == null);
    }
    public void Push (int inhalt)
    {
        StackElement neuesElement = new StackElement(inhalt);
        neuesElement.ElementAnhängen(OberstesElement);
        OberstesElement = neuesElement;
    }
    public int Pop ()
    {
        int ObersterInhalt = OberstesElement.GetInhalt();
        OberstesElement = OberstesElement.GetNächstes();
        return ObersterInhalt;
    }
}
```

20

Tobias Scheffer

Stack

- Eingabe: ein String, der einen mathematischen Ausdruck enthält.
- Ausgabe. „ja“, wenn die Klammern des Ausdrucks balanciert sind, sonst „nein“.
- Beispiel:
 - ◆ $(2 + [3 * x]) \rightarrow \text{ja.}$
 - ◆ $((f(x))) \rightarrow \text{nein.}$
 - ◆ $[((f(x)^2)+1)] \rightarrow \text{ja.}$

21

Stack: Klammern balanciert?

```
public class Parser
{
    public boolean Parse (String ausdruck)
    {
        StackInteraktiv KlammerStack = new StackInteraktiv ();
        int index;

        for (index = 0; index < ausdruck.length(); index++) {
            if (ausdruck.charAt(index) == '(' ||
                ausdruck.charAt(index) == '[' ||
                ausdruck.charAt(index) == '{')
                KlammerStack.Push(ausdruck.charAt(index));
            else if (ausdruck.charAt(index) == ')' ||
                    ausdruck.charAt(index) == ']' ||
                    ausdruck.charAt(index) == '}') {
                char schließend = ausdruck.charAt(index);
                if (KlammerStack.IsEmpty()) return false;
                char öffnend = KlammerStack.Pop();
                if ((öffnend == '(' && schließend != ')')
                    || (öffnend == '[' && schließend != ']')
                    || (öffnend == '{' && schließend != '}'))
                    return false;
            }
        }
        return KlammerStack.IsEmpty();
    }
}
```

22

Klasse Queue

- [Sprich: „kjuh“]. Warteschlange.
- Elemente können
 - ◆ eingereicht,
 - ◆ am anderen Ende herausgenommen werden.
- Sehr nützliche Klasse:
 - ◆ Prozesse warten in Queue darauf, den Prozessor benutzen zu können.
 - ◆ Supermarktkunden warten in Queue vor der Käsetheke.
 - ◆ Schlangen vor Clubs sind keine Queues. (Mädchen werden rausgezogen und dürfen rein).
- FIFO: First in, first out.

23

Klasse Queue



24

Klasse Queue

- Attribute:
 - ◆ Anfang,
 - ◆ Ende.
- Methoden:
 - ◆ **Queue.Enqueue(Element)**: Verändert Queue.
 - ◆ **Queue.Dequeue()**: → Element, verändert Queue.
 - ◆ **Queue.IsEmpty()**: → Boolean.

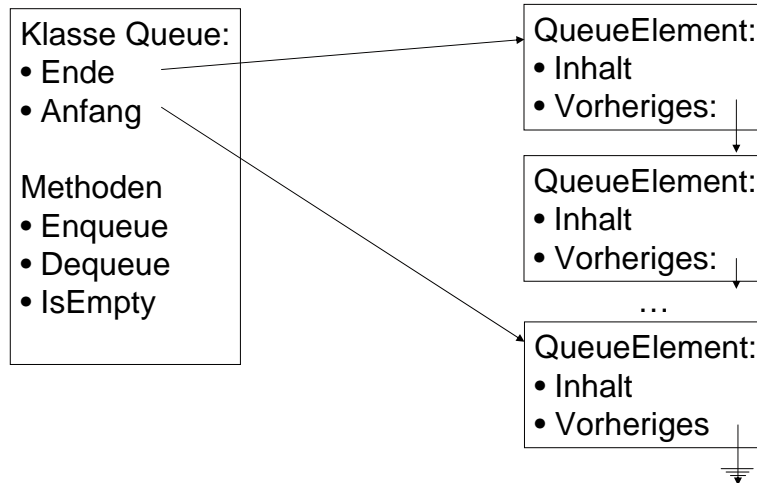
25

Queue: Axiome

- Sehr knifflig, das gewünschte Verhalten eines Queues mit mathematischen Aussagen exakt zu spezifizieren.
- Noch kniffliger, zu beweisen, dass eine Implementierung der Klasse Queue diese Spezifikation erfüllt.
- Philosophie heute („model checking“):
 - ◆ Einige wichtige, nicht zu komplizierte Eigenschaften mathematisch spezifizieren.
 - ◆ Mit automatischen Beweisverfahren zeigen, dass Eigenschaften für Softwaresystem gelten.

26

Klasse Queue



27

Klasse QueueElement

```
public class QueueElement
{
    private char Inhalt;
    private QueueElement Vorheriges;

    public QueueElement (char inh, QueueElement V)
    {
        Inhalt = inh;
        Vorheriges = V;
    }
    public char GetInhalt ()
    {
        return Inhalt;
    }
    public QueueElement GetVorheriges ()
    {
        return Vorheriges;
    }
    public void LöscheVorheriges ()
    {
        Vorheriges = null;
    }
}
```

28

Klasse Queue

```
public class Queue
{
    private QueueElement Anfang;
    private QueueElement Ende;

    public boolean isEmpty()
    {
        return Anfang == null;
    }

    public void Enqueue (char inhalt)
    {
        boolean erstesMal = isEmpty();
        Ende = new QueueElement (inhalt, Ende);
        if (erstesMal) Anfang = Ende;
    }

    public char Dequeue ()
    {
        char inhaltAnfang = Anfang.GetInhalt();
        QueueElement zweites = Ende;
        while (zweites.GetVorheriges() != Anfang) zweites = zweites.GetVorheriges();
        Anfang = zweites;
        zweites.LöscheVorheriges();
        return inhaltAnfang;
    }
}
```

29

Klasse Queue

- Ein Queue lässt sich auch mit zwei Stacks implementieren.
- Wie?

30

Klasse Liste

- Geordnete Menge.
- Elemente können
 - ◆ an jeder Stelle eingefügt,
 - ◆ verändert,
 - ◆ gelöscht werden.
- Listen sind auch praktisch:
 - ◆ Z.B. Liste alphabetisch sortierter Namen.
 - ◆ Kontakte einfügen / löschen möglich.
- Viele Varianten der Definition üblich.

31

Klasse Liste

- Attribute:
 - ◆ Erstes Listenelement.
 - ◆ Aktuelles Listenelement.
- Methoden:
 - ◆ **Liste.ZumAnfang()**: Verändert Liste.
 - ◆ **Liste.Lies()**: → liefert Element.
 - ◆ **Liste.Nächstes()**: Verändert Liste.
 - ◆ **Liste.Lösche()**: Verändert Liste.
 - ◆ **Liste.EinfügenVor(Element)**: Verändert Liste.
 - ◆ **Liste.AktuellesLeer ()**: → Boolean.

32

Liste: Axiome

- Sehr knifflig, das gewünschte Verhalten einer Liste mit mathematischen Aussagen exakt zu spezifizieren.
- Noch kniffliger, zu beweisen, dass eine Implementierung der Klasse Liste diese Spezifikation erfüllt.
- Siehe Kommentare zu Axiome Queue.

33

Liste: Axiome

Zustand s_0 .

- Liste.EinfügenVor(e1);
- Liste.Lösche();

Zustand $s_1 = s_0$.

34

Liste: Axiome

Zustand s_0 .

- `e1 = Liste.Lies();`
- `Liste.Lösche();`
- `Liste.EinfügenVor(e1);`

Zustand s_1 .

Wenn $[s_0](\text{Liste.AktuellesLeer()}) = \text{false}$
dann gilt: $s_1 = s_0$.

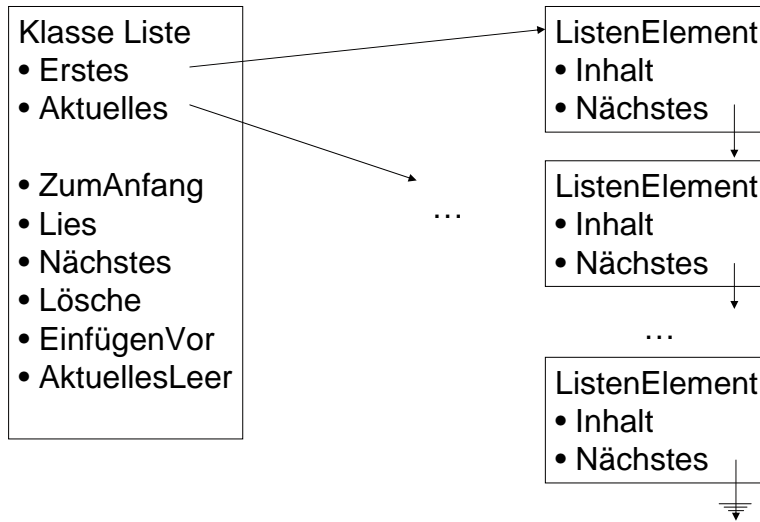
35

Liste: Axiome

- ...
- überlegen Sie sich noch ein paar.

36

Liste



37

Klasse ListElement

38

Klasse Liste

Tobias Scheffer

39

Liste invertieren

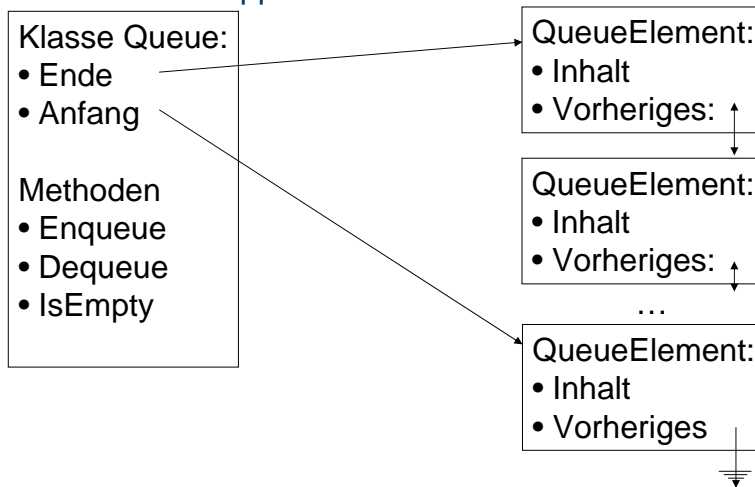
Tobias Scheffer

40

Minimum in Liste finden

Doppelt verkettete Liste.

- Queue mit doppelt verketteter Liste



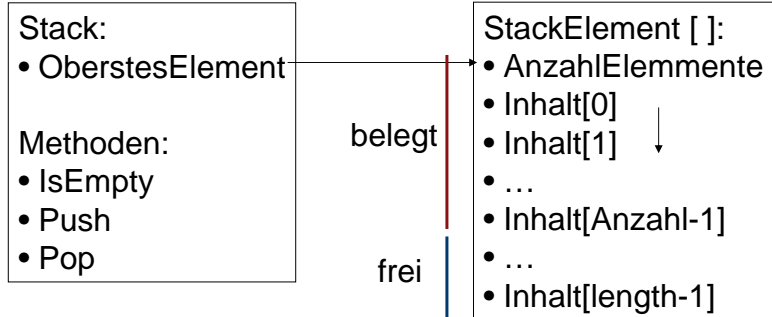
Verkettete Listen

- Wie viele Operationen braucht der Interpreter, um auf das n-te Element zuzugreifen?
- Einfügen eines Elementes: „Teuer“ oder „billig“?

Listen mit Arrays implementiert

- Listen, Stacks, Queues.
- Statt verketteter Liste: Array von Elementen.
- Wenn Array voll → größeres Array anlegen, Elemente umkopieren.
- Element einfügen (außer am Ende): alle folgenden Elemente eine Position hochkopieren, dann in freie Position einfügen.

Stack mit Array implementiert



Stack mit Array implementiert