



Listen, Stapel, Warteschlangen

Thomas Röfer

Abstrakte Datentypen

Reihung

Stapel

Warteschlange

Einfach und doppelt verkettete Liste



Rückblick „Prozeduren, Funktionen, Datenströme, JavaDoc“

Unterprogramme

Prozeduren ↔ Funktionen
Kopf (Signatur) ↔ Rumpf
Formale ↔ Aktuelle Parameter
Ein/Ausgabe & Transiente P.

Überladung

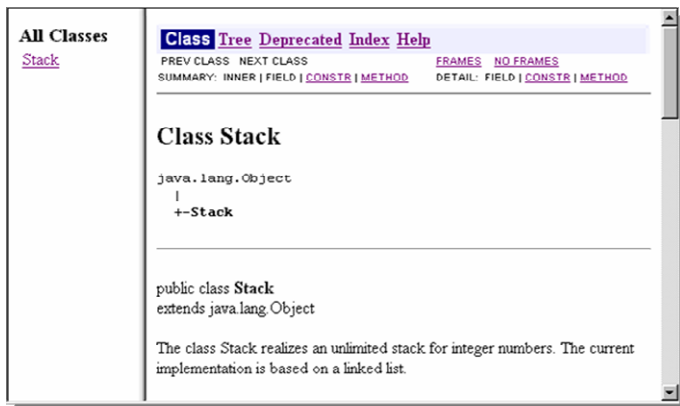
plus : int × long → int
 plus : int × long → long
 plus : long × long → long
 plus : long × int → int

Die Funktion main()

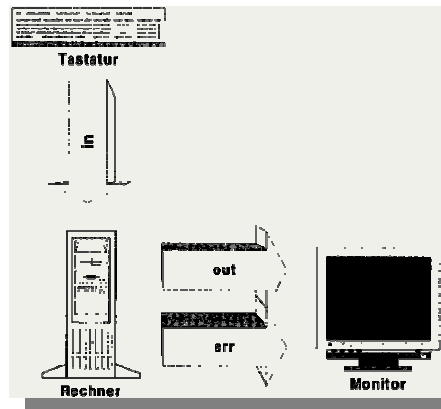
>java Klasse Hallo 21333 "dies und das"

String[] args	
args.length	3
args[0]	"Hallo"
args[1]	"21333"
args[2]	"dies und das"

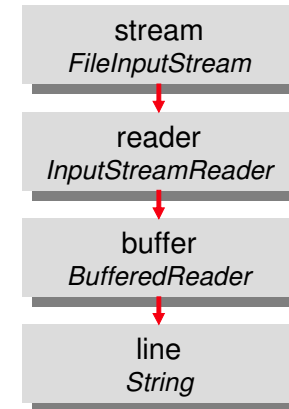
JavaDoc



Datenströme



Datenströme in Java





Abstrakte Datentypen

▶ Definition

- ▶ Ein abstrakter Datentyp besteht aus den Daten selbst (z.B. Objektzustand) und den auf den Daten auszuführenden Operationen (z.B. Methoden)
- ▶ Öffentliche Schnittstelle der Operationen
- ▶ Interne Abläufe werden versteckt (Geheimnisprinzip)
- ▶ In Java: Klassen

▶ Algebraische abstrakte Datentypen

- ▶ Neben der Schnittstelle der Operationen wird zusätzlich die Semantik der Operationen formal beschrieben, z.B. in Form von Axiomen
 - ▶ $plus : int \times int \rightarrow int$
 - ▶ $null : \rightarrow int$
 - ▶ $plus(a, b) = plus(b, a)$
 - ▶ $plus(a, null) = a$
 - ▶ $plus(a, plus(b, c)) = plus(plus(a, b), c)$

Reihungen

- ▶ **Eigenschaften**
 - ▶ Wahlfreier Zugriff auf alle Elemente in konstanter Zeit über Index
 - ▶ Reihungen haben normalerweise eine feste Größe
 - ▶ Wenn nicht, sind Vergrößern und Verkleinern teure Operationen (neue Reihung anlegen, alte Daten hinüberkopieren)
 - ▶ Einfügen und Löschen sind teure Operationen, da alle Elemente hinter dem eingefügten/gelöschten kopiert werden müssen
- ▶ **Operationen einer Reihung fester Größe**
 - ▶ Reihung erzeugen
 - ▶ $create : int \rightarrow Array$
 - ▶ Eintrag schreiben
 - ▶ $set : Array \times int \times Entry \rightarrow Array$
 - ▶ Eintrag auslesen
 - ▶ $get : Array \times int \rightarrow Entry$
 - ▶ Länge auslesen
 - ▶ $length : Array \rightarrow int$



Reihung mit Einfügen und Löschen

```
// int = Datentyp der Einträge

class Array
{
    int[] entries = new int[0];

    int get(int pos)
    {
        return entries[pos];
    }

    void set(int pos, int entry)
    {
        entries[pos] = entry;
    }

    boolean empty()
    {
        return entries.length == 0;
    }
}
```

```
void insertBefore(int pos, int entry)
{
    int[] temp = new int[entries.length + 1];

    for(int i = 0; i < pos; ++i)
        temp[i] = entries[i];
    temp[pos] = entry;
    for(int i = pos; i < entries.length; ++i)
        temp[i + 1] = entries[i];
    entries = temp;
}

void remove(int pos)
{
    int[] temp = new int[entries.length - 1];

    for(int i = 0; i < pos; ++i)
        temp[i] = entries[i];
    for(int i = pos + 1; i < entries.length; ++i)
        temp[i - 1] = entries[i];
    entries = temp;
}
}
```



Stapel (Stack)

▶ Zweck

- ▶ Ein Stapel dient zum Zwischenspeichern von Elementen
- ▶ Der Zugriff erfolgt nach dem *last-in, first-out* Prinzip (lifo), d.h. das zuletzt abgelegte Element wird zuerst zurückgeliefert

▶ Operationen

- ▶ Stapel erzeugen
 - ▶ $create : \rightarrow Stack$
- ▶ Eintrag ablegen
 - ▶ $push : Stack \times Entry \rightarrow Stack$
- ▶ Kopf auslesen
 - ▶ $top : Stack \rightarrow Entry$
- ▶ Eintrag entnehmen
 - ▶ $pop : Stack \rightarrow Stack$
- ▶ Auf Leere testen
 - ▶ $empty : Stack \rightarrow boolean$

▶ Axiome

- ▶ $top(push(s, e)) = e$
- ▶ $pop(push(s, e)) = s$
- ▶ $empty(create())$
- ▶ $\neg empty(push(s, e))$

▶ Alternative

- ▶ Eintrag entnehmen und zurückliefern
 - ▶ $pop : Stack \rightarrow Stack \times Entry$

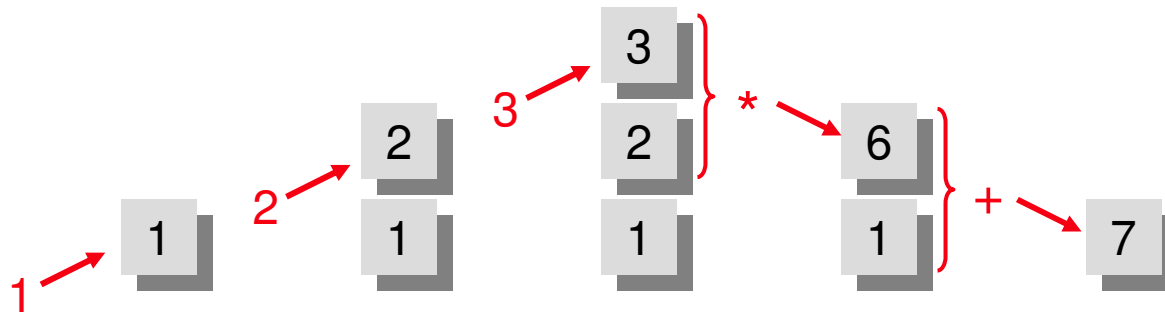
Beispiel: Ein UPN-Rechner

▶ Umgekehrte Polnische Notation

- ▶ Die Operanden kommen zuerst, dann kommt der Operator, z.B. $1\ 2\ +$
- ▶ Arbeitet als Stack-Maschine: Jeder Operator holt sich seine Operanden von der Spitze des Stapels und legt das Ergebnis wieder dort ab

▶ Beispiel

- ▶ Für $1 + 2 * 3$ schreibt man: $1\ 2\ 3\ *\ +$

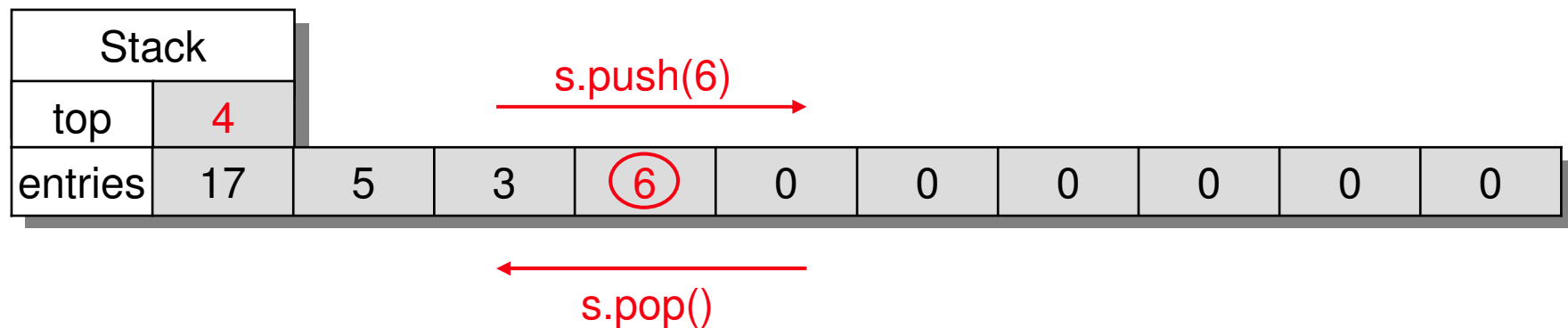




Beispiel: Ein UPN-Rechner in Java

```
class Eval
{
    static double eval(String[] expression)
    {
        Stack stack = new Stack();
        for(int i = 0; i < expression.length; ++i)
        {
            String s = expression[i];
            if(s.equals("+"))
                stack.push(stack.pop() + stack.pop());
            else if(s.equals("-"))
                stack.push(-stack.pop() + stack.pop());
            else if(s.equals("*"))
                stack.push(stack.pop() * stack.pop());
            else if(s.equals("/"))
                stack.push(1 / stack.pop() * stack.pop());
            else
                stack.push(Double.parseDouble(s));
        }
        return stack.pop();
    }
}
```

Beschränkter Stapel





Beschränkter Stapel

```
class Stack
{
    double[] entries;
    int top;

    Stack(int size)
    {
        entries = new double[size];
        top = 0;
    }

    void push(double entry)
    {
        assert top < entries.length;
        entries[top++] = entry;
    }
}
```

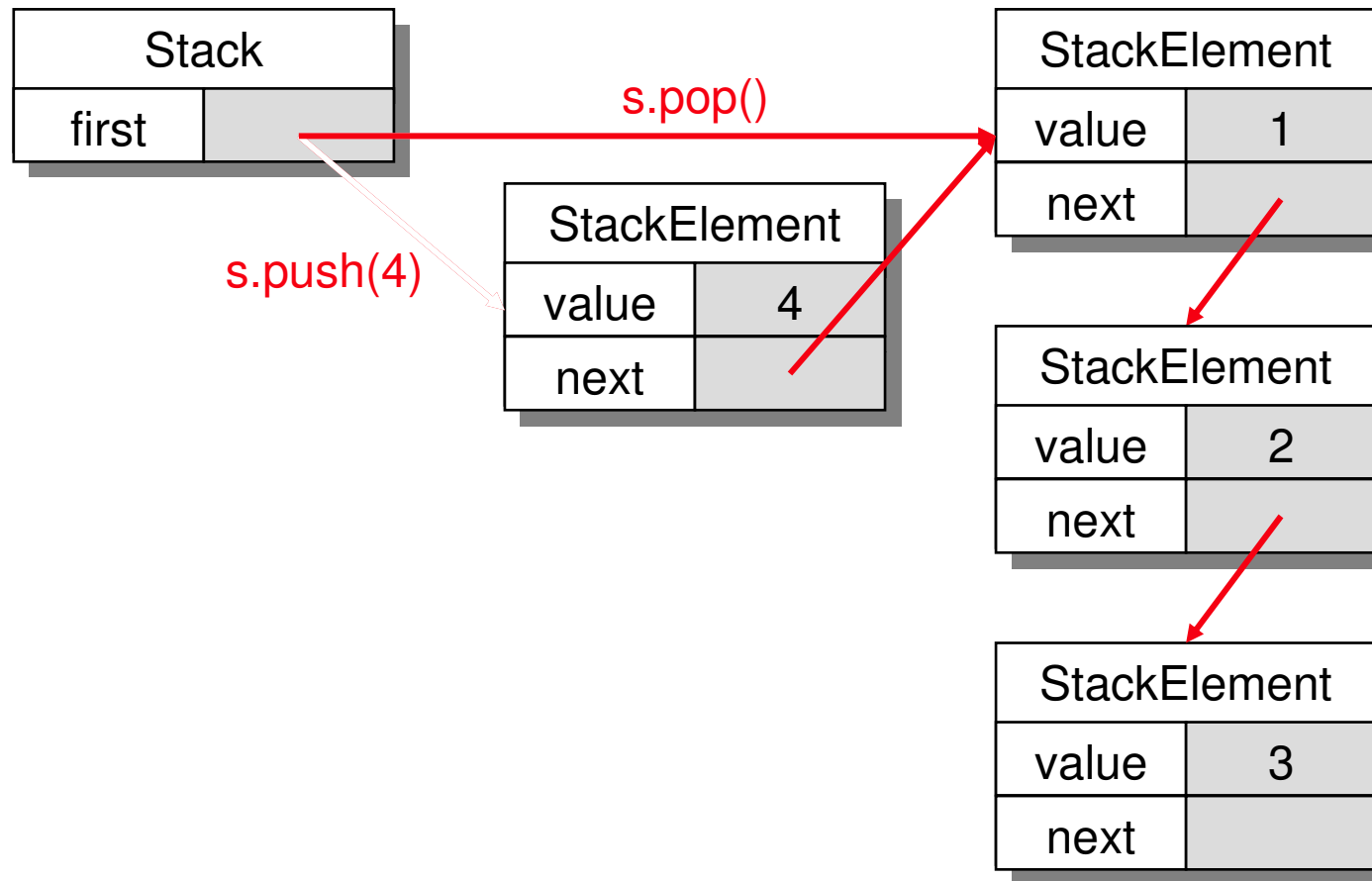
```
double pop()
{
    assert !empty();
    return entries[--top];
}

boolean empty()
{
    return top == 0;
}

// double = Datentyp der Einträge
```



Stack als einfach verkettete Liste





Stack als einfach verkettete Liste

```
class StackElement
{
    StackElement next;
    double value;
}
```

```
class Stack
{
    StackElement first;

    Stack()
    {
        first = null;
    }

    void push(double entry)
    {
        StackElement s =
            new StackElement();
        s.value = entry;
        s.next = first;
        first = s;
    }
}
```

```
double pop()
{
    double d = first.value;
    first = first.next;
    return d;
}

boolean empty()
{
    return first == null;
}
```



Warteschlange (Queue)

▶ Zweck

- ▶ Eine Warteschlange dient zum Zwischenspeichern von Elementen
- ▶ Der Zugriff erfolgt nach dem *first-in, first-out* Prinzip (lilo), d.h. das zuerst abgelegte Element wird auch zuerst zurückgeliefert

▶ Operationen

- ▶ Warteschlange erzeugen
 - ▶ $create : \rightarrow Queue$
- ▶ Eintrag ablegen
 - ▶ $push : Queue \times Entry \rightarrow Queue$
- ▶ Kopf auslesen
 - ▶ $top : Queue \rightarrow Entry$
- ▶ Eintrag entnehmen
 - ▶ $pop : Queue \rightarrow Queue$
- ▶ Auf Leere testen
 - ▶ $empty : Queue \rightarrow boolean$

▶ Axiome

- ▶ $top(push(push(create(), e), f)) = e$
- ▶ $pop(push(push(create(), e), f)) = push(create(), f)$
- ▶ $empty(create())$
- ▶ $\neg empty(push(q, e))$

▶ Alternative

- ▶ Eintrag entnehmen und zurückliefern
 - ▶ $pop : Queue \rightarrow Queue \times Entry$



Beispiel: UPN-Rechner mit Puffer

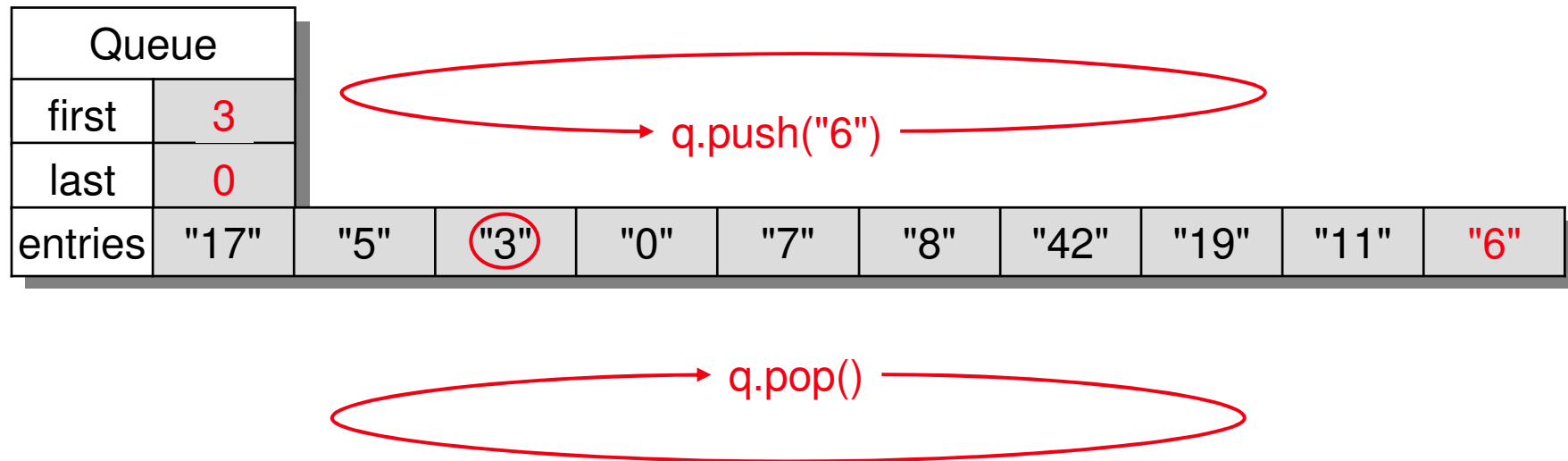
```
class Eval
{
    static double eval(String expression)
    {
        return eval(split(expression));
    }

    static Queue split(String expression)
    {
        Queue queue = new Queue();
        expression = expression.trim();
        while(!expression.equals(""))
        {
            int index = (expression + " ").indexOf(" ");
            queue.push(expression.substring(0, index));
            expression = expression.substring(index).trim();
        }
        return queue;
    }

    static double eval(Queue queue)
    {
        Stack stack = new Stack(10);
        while(!queue.empty())
        {
            String s = queue.pop();
        }
    }
}
```



Beschränkte Warteschlange (Ringpuffer)





Beschränkte Warteschlange (Ringpuffer)

```
class Queue
{
    String[] entries;
    int first,
        last;

    Queue(int size)
    {
        entries = new String[size + 1];
        first = last = 0;
    }

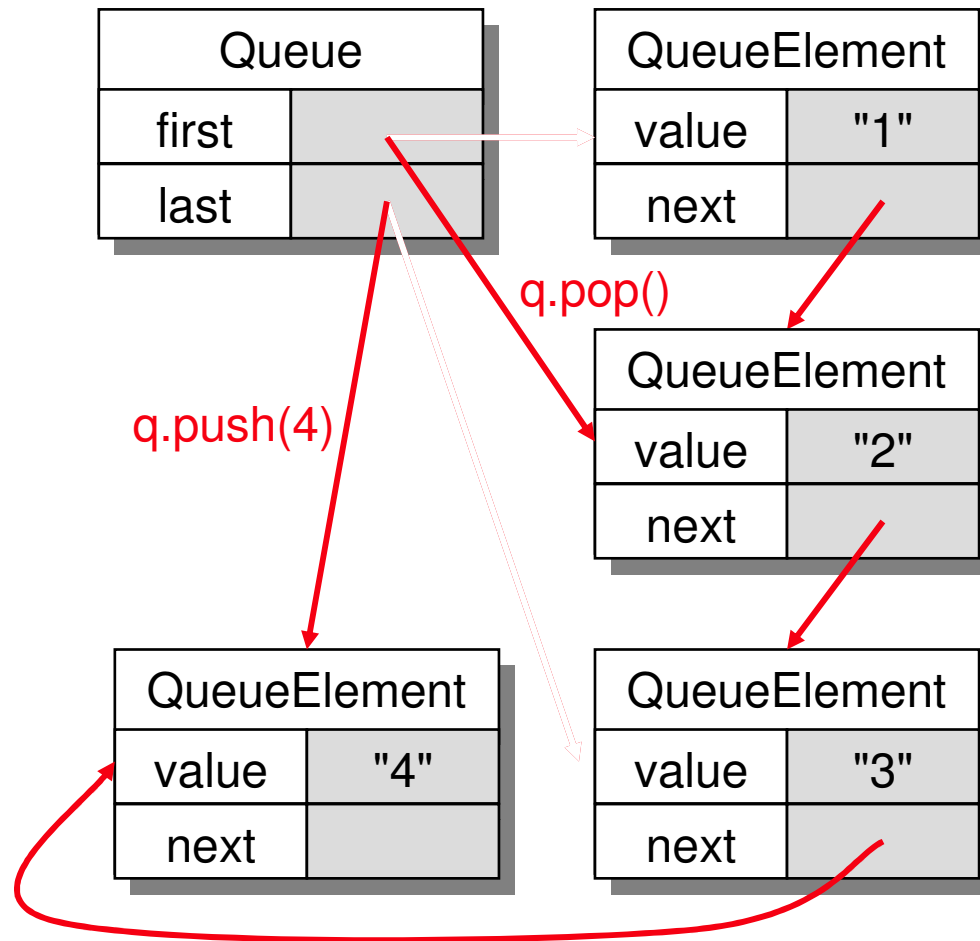
    void push(String entry)
    {
        entries[last++] = entry;
        last %= entries.length;
        assert !empty();
    }
}
```

```
String pop()
{
    assert !empty();
    String entry = entries[first++];
    first %= entries.length;
    return entry;
}

boolean empty()
{
    return first == last;
}

// String = Datentyp der Einträge
```

Warteschlange als einfach verkettete Liste





Warteschlange als einfach verkettete Liste

```
class QueueElement
{
    QueueElement next;
    String value;

    QueueElement(String s)
    {
        value = s;
        next = null;
    }
}

class Queue
{
    QueueElement first,
                last;

    Queue()
    {
        first = last = null;
    }
}
```

```
void push(String s)
{
    if(empty())
        first = last = new QueueElement(s);
    else
    {
        last.next = new QueueElement(s);
        last = last.next;
    }
}

String pop()
{
    assert !empty();
    String s = first.value;
    first = first.next;
    return s;
}

boolean empty()
{
    return first == null;
}
}
```

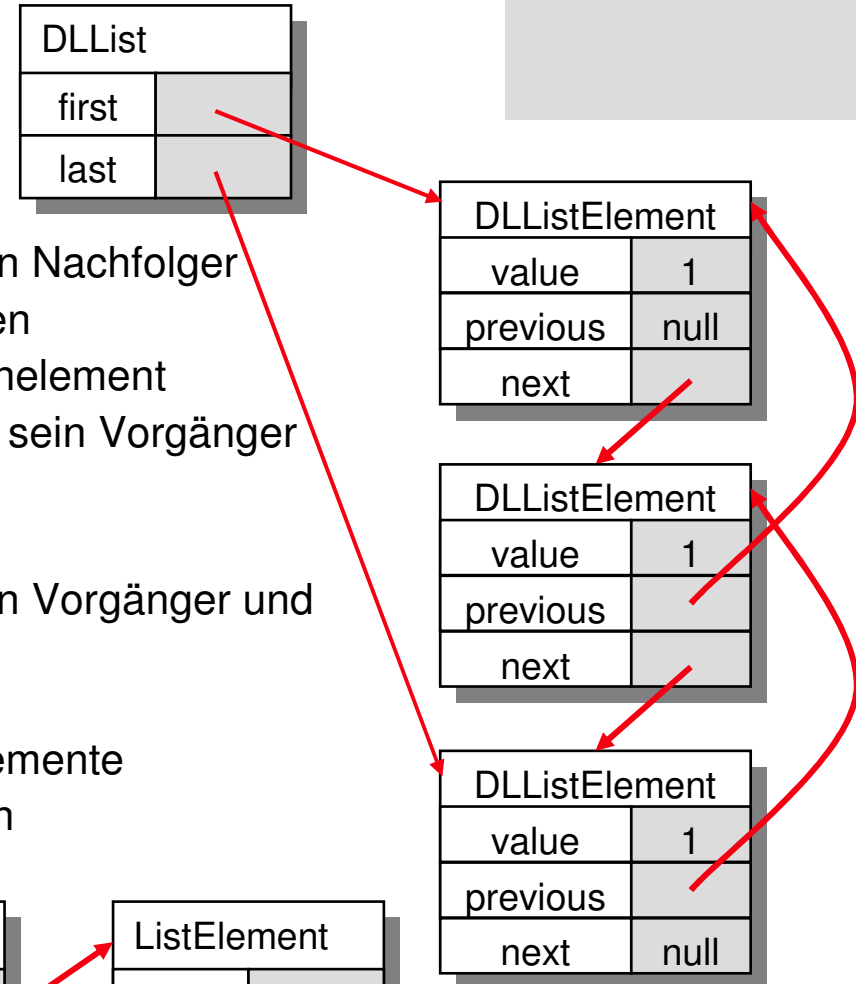
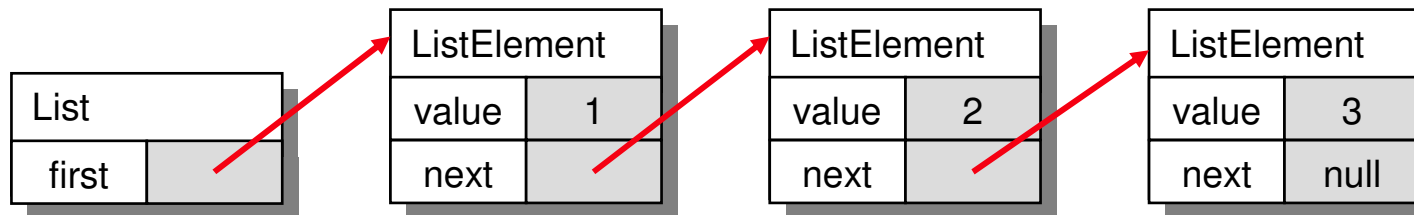
Listen

▶ Einfach verkettete Liste

- ▶ Jedes Element enthält einen Zeiger auf seinen Nachfolger
- ▶ Kann nur in einer Richtung durchlaufen werden
- ▶ Einfügen immer hinter ein existierendes Listenelement
- ▶ Ein Element kann nur gelöscht werden, wenn sein Vorgänger bekannt ist

▶ Doppelt verkettete Liste

- ▶ Jedes Element enthält einen Zeiger auf seinen Vorgänger und seinen Nachfolger
- ▶ Kann vor- und rückwärts durchlaufen werden
- ▶ Einfügen vor und hinter existierende Listenelemente
- ▶ Elemente können problemlos gelöscht werden





Operationen auf Listen

▶ **Eigenschaften**

- ▶ Zugriff auf Elemente nur in fester Reihenfolge
- ▶ Einfügen und Löschen sind in konstanter Zeit möglich
- ▶ Speicherplatzverbrauch höher als bei Reihungen

▶ **Operationen auf Listen (z.B. einfach verkettet)**

▶ List

- ▶ *create* : $\rightarrow List$
- ▶ *first* : $List \rightarrow ListElement$
- ▶ *insert* : $List \times Entry \times ListElement \rightarrow List$
- ▶ *remove* : $List \times ListElement \rightarrow List$
- ▶ *empty* : $List \rightarrow boolean$

▶ ListElement

- ▶ *last* : $ListElement \rightarrow boolean$
- ▶ *next* : $ListElement \rightarrow ListElement$
- ▶ *entry* : $ListElement \rightarrow Entry$



Einfach verkettete Liste mit Index

```
class ListElement
{
    int entry;
    ListElement next;

    ListElement(int e)
    {
        entry = e;
        next = null;
    }
}

class List
{
    ListElement first;

    List()
    {
        first = null;
    }
}
```

```
ListElement at(int pos)
{
    ListElement current = first;
    while(pos > 0 && current != null)
    {
        current = current.next;
        --pos;
    }
    return current;
}

int get(int pos)
{
    return at(pos).entry;
}

void set(int pos, int entry)
{
    at(pos).entry = entry;
}
```



Einfach verkettete Liste mit Index

```
void insertBefore(int pos, int entry)
{
    ListElement newElem =
        new ListElement(entry);
    if(pos == 0)
    {
        newElem.next = first;
        first = newElem;
    }
    else
    {
        ListElement current = at(pos - 1);
        newElem.next = current.next;
        current.next = newElem;
    }
}
```

```
void remove(int pos)
{
    if(pos == 0)
        first = first.next;
    else
    {
        ListElement current = at(pos - 1);
        current.next = current.next.next;
    }
}

boolean empty()
{
    return first == null;
}
```



Abbildung von Stack/Queue auf List

```
class Stack
{
    List list = new List();

    void push(Entry e)
    {
        list.insertBefore(e, list.first());
    }

    Entry pop()
    {
        ListElement e = first();
        list.remove(e);
        return e.entry;
    }

    boolean empty()
    {return list.empty();}
```

```
class Queue
{
    List list = new List();

    void push(Entry e)
    {
        list.insertBefore(e, list.first());
    }

    Entry pop()
    {
        ListElement e = last();
        list.remove(e);
        return e.entry;
    }

    boolean empty()
    {return list.empty();}
```