

Praktische Informatik 1

Listen, Stapel, Warteschlangen

Thomas Röfer

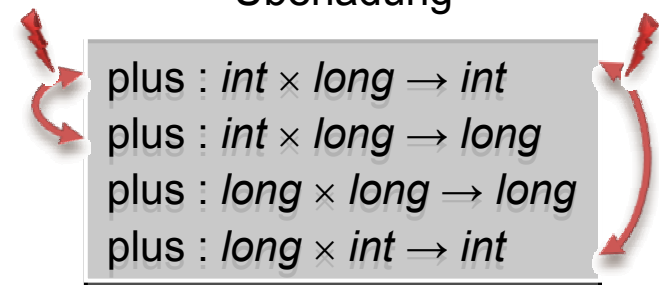
- Abstrakte Datentypen
- Reihung
- Stapel
- Warteschlange
- Einfach und doppelt verkettete Liste
- Musterlösung, Übungsblatt

Rückblick „Prozeduren, Funktionen, JavaDoc, Datenströme“

Unterprogramme

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

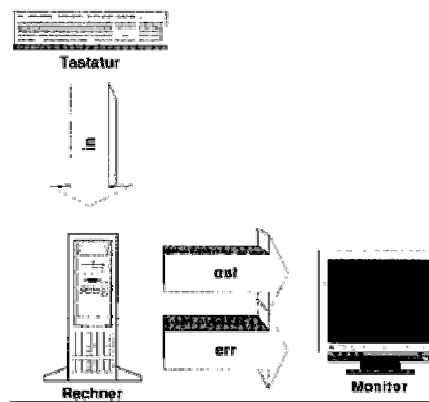
Überladung



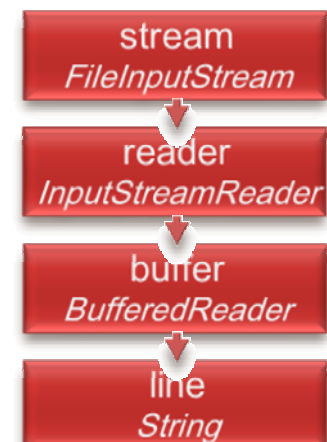
JavaDoc



Datenströme



Datenströme in Java



Abstrakte Datentypen

- 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
- Beispiele
 - $plus : int \times int \rightarrow int$
 - $null : \rightarrow int$
 - $neg : int \rightarrow int$

Algebraische abstrakte Datentypen

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

Reihungen

- 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 hinüber kopieren)
- Einfügen und Löschen sind teure Operationen, da alle Elemente hinter dem eingefügten/gelöschten kopiert werden müssen

Operationen auf Reihungen

- 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 variabler Größe (zusätzlich)
 - Einfügen
 - $insert : Array \times int \times Entry \rightarrow Array$
 - Löschen
 - $remove : Array \times int \rightarrow Array$
 - Auf Leere testen
 - $empty : Array \rightarrow boolean$

Reihung mit Einfügen und Löschen

```
class Array {  
    int[] entries = new int[0];  
  
    int length() {  
        return entries.length;  
    }  
  
    boolean empty() {  
        return length() == 0;  
    }  
  
    int get(int pos) {  
        return entries[pos];  
    }  
  
    void set(int pos, int entry) {  
        entries[pos] = entry;  
    }  
}
```

- Datentyp der Einträge:
int

```
void insertBefore(int pos, int entry) {  
    int[] temp = new int[length() + 1];  
  
    for (int i = 0; i < pos; ++i) {  
        temp[i] = entries[i];  
    }  
    temp[pos] = entry;  
    for (int i = pos; i < length(); ++i) {  
        temp[i + 1] = entries[i];  
    }  
    entries = temp;  
}
```

Reihung mit Einfügen und Löschen

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

Stapel (Stack)

- Ein Stapel dient zum Zwischenspeichern von Elementen
- Der Zugriff erfolgt nach dem *last-in, first-out* Prinzip (*lifo*)
 - 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

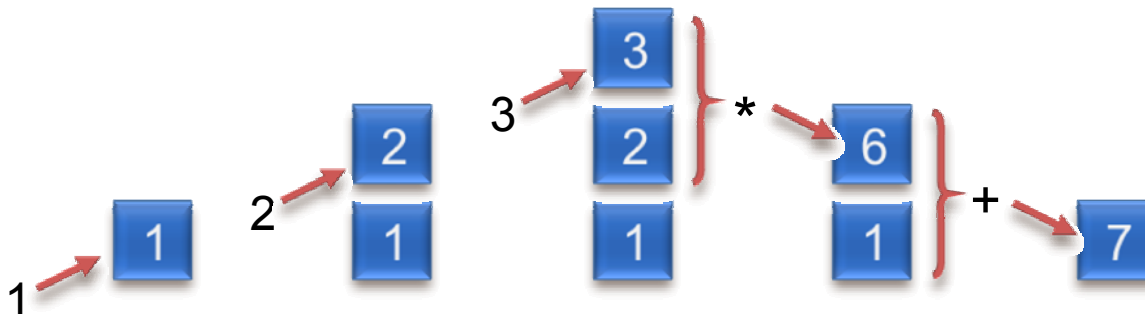
- $top(push(s, e)) = e$
- $pop(push(s, e)) = s$
- $empty(create())$
- $\neg empty(push(s, e))$

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 (String s : expression) {  
            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(1.0 / stack.pop() * stack.pop());  
            } else {  
                stack.push(Double.parseDouble(s));  
            }  
        }  
        return stack.pop();  
    }  
}
```

Beschränkter Stapel

s.push(6) →

Stack											
length	3 4 3										
entries	17	5	3	0 6	0	0	0	0	0	0	0

← s.pop()

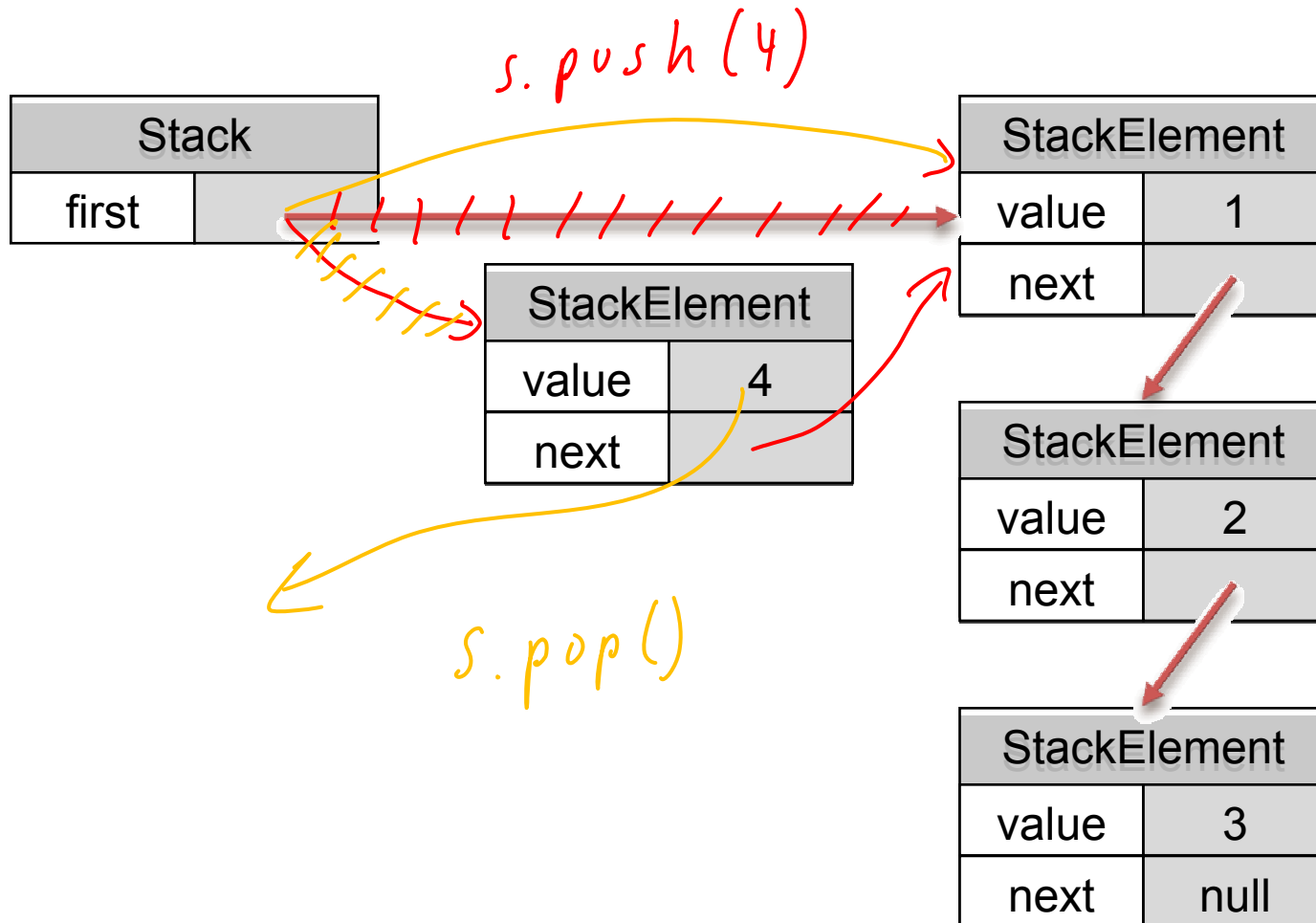
Beschränkter Stapel

- Datentyp der Einträge: *double*

```
class Stack {  
    double[] entries;  
    int length;  
  
    Stack(int size) {  
        entries = new double[size];  
        length = 0;  
    }  
  
    boolean empty() {  
        return length == 0;  
    }  
}
```

```
void push(double entry) {  
    assert length < entries.length;  
    entries[length++] = entry;  
}  
  
double pop() {  
    assert !empty();  
    return entries[--length];  
}  
  
double top() {  
    assert !empty();  
    return entries[length - 1];  
}  
}
```

Stack als einfach verkettete Liste



Warteschlange (Queue)

- Eine Warteschlange dient zum Zwischenspeichern von Daten
- Der Zugriff erfolgt nach dem *first-in, first-out* Prinzip (*fifo*)

- 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

- 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

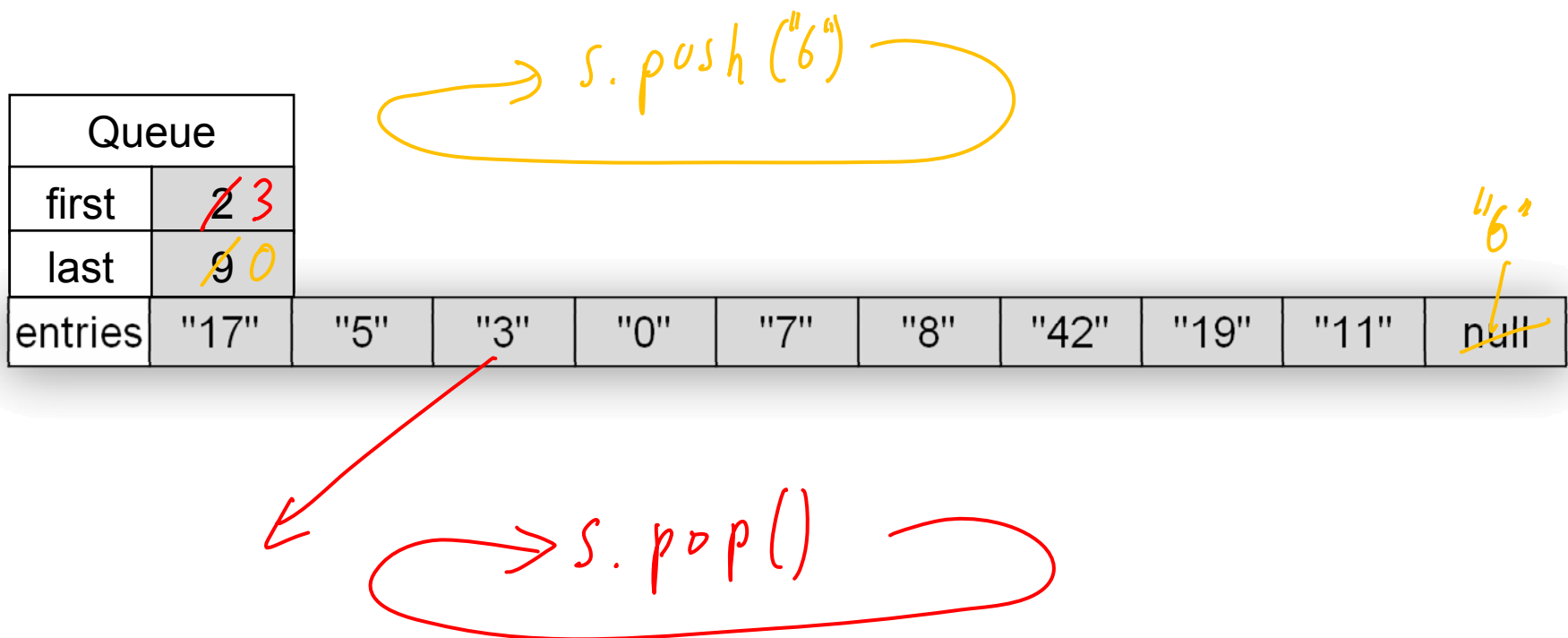
- Eintrag entnehmen und zurückliefern
 - $pop : Queue \rightarrow Queue \times Entry$

Beispiel: UPN-Rechner mit Puffer

```
class Eval2 {  
    static double eval(String expression) {  
        String[] tokens = expression.split(" ");  
        Queue queue = new Queue(tokens.length);  
        for (String t : tokens) {  
            queue.push(t);  
        }  
        return eval(queue);  
    }  
}
```

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

Beschränkte Warteschlange (Ringpuffer)



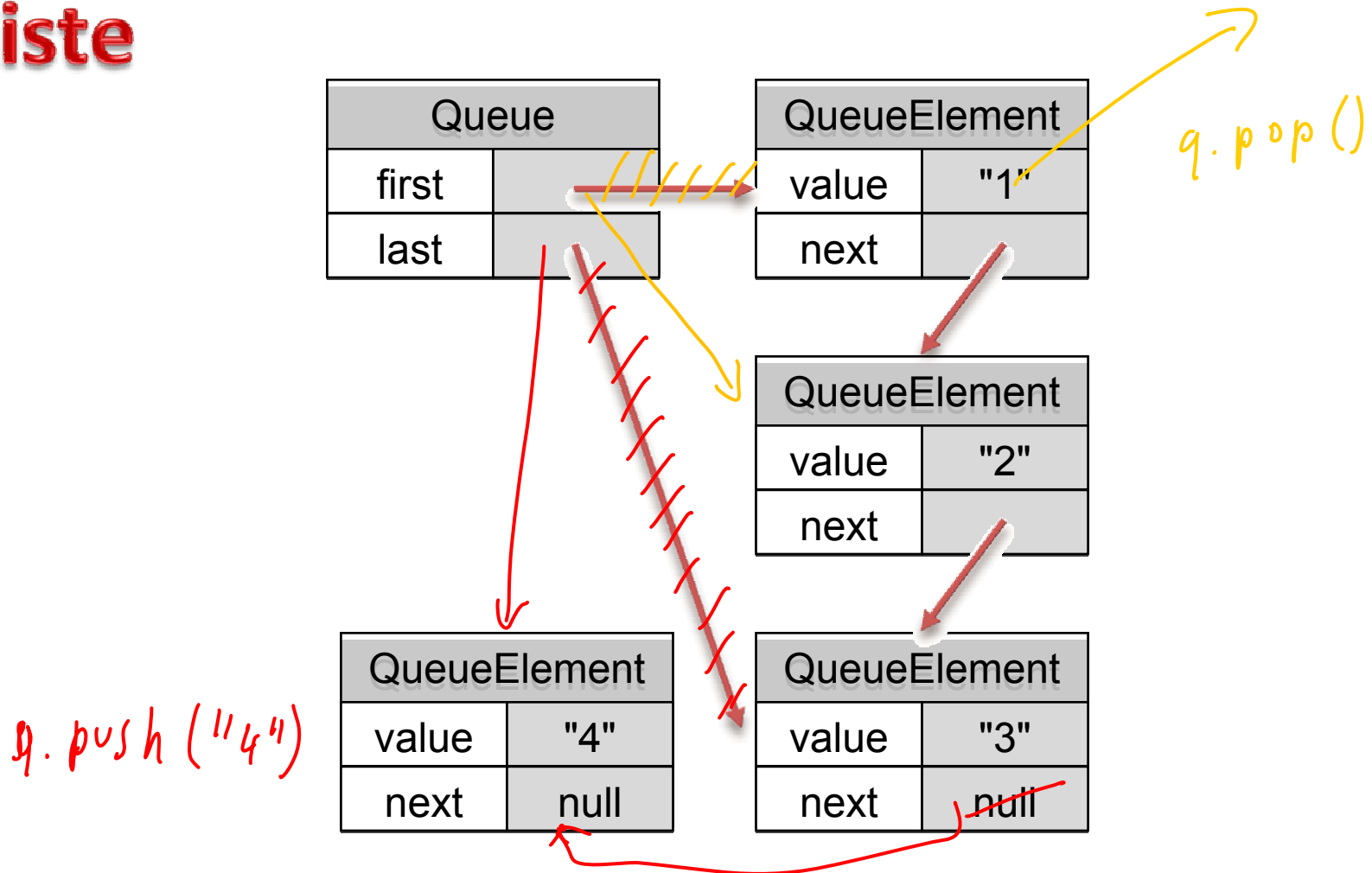
Beschränkte Warteschlange (Ringpuffer)

- Datentyp der Einträge:
String

```
class Queue {  
    String[] entries;  
    int first;  
    int last;  
  
    Queue(int size) {  
        entries = new String[size + 1];  
        first = last = 0;  
    }  
  
    boolean empty() {  
        return first == last;  
    }  
}
```

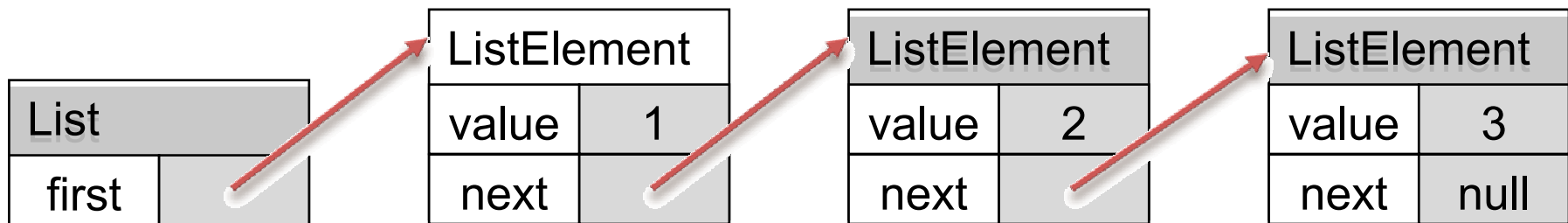
```
void push(String entry) {  
    entries[last++] = entry;  
    last %= entries.length;  
    assert !empty();  
}  
  
String pop() {  
    assert !empty();  
    String entry = entries[first++];  
    first %= entries.length;  
    return entry;  
}  
  
String top() {  
    assert !empty();  
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


Warteschlange als einfach verkettete Liste



Einfach verkettete Liste

- Jedes Element enthält einen Zeiger auf seinen Nachfolger
- Kann nur in einer Richtung durchlaufen werden
- Einfügen immer am Anfang der Liste oder hinter einem bekannten Listenelement
- Ein Element kann nur gelöscht werden, wenn sein Vorgänger bekannt ist



Operationen auf Listen

- Eigenschaften
 - Zugriff auf Elemente nur in fester Reihenfolge
 - Einfügen und Löschen sind in konstanter Zeit möglich
 - Speicherplatzverbrauch höher als bei Reihungen
- 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
 - $create : Entry \times ListElement \rightarrow ListElement$
 - $last : ListElement \rightarrow boolean$
 - $next : ListElement \rightarrow ListElement$
 - $entry : ListElement \rightarrow Entry$

Einfach verkettete Liste

- Datentyp der Einträge:
int

```
class ListElement {  
    int entry;  
    ListElement next;  
  
    ListElement(int entry, ListElement next) {  
        this.entry = entry;  
        this.next = next;  
    }  
  
    boolean last() {  
        return next == null;  
    }  
}
```

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

Diagram illustrating a linked list structure. A box labeled "anchor" is connected to a node box. The node box contains a pointer field (indicated by a horizontal arrow) and a data field (indicated by a diagonal arrow pointing to a blue-shaded area labeled "entry) {").

```
void insertAfter(ListElement anchor, int entry) {
    if (anchor == null) {
        first = new ListElement(entry, first);
    } else {
        anchor.next = new ListElement(entry, anchor.next);
    }
}

void removeNext(ListElement anchor) {
    if (anchor == null) {
        first = first.next;
    } else {
        anchor.next = anchor.next.next;
    }
}
```

Einfach verkettete Liste

```
ListElement previous(ListElement anchor) {  
    if (first == anchor) {  
        return null;  
    } else {  
        ListElement current = first;  
        while (current.next != anchor) {  
            current = current.next;  
        }  
        return current;  
    }  
}  
  
void insertBefore(ListElement anchor, int entry) {  
    insertAfter(previous(anchor), entry);  
}  
  
void remove(ListElement element) {  
    removeNext(previous(element));  
}
```