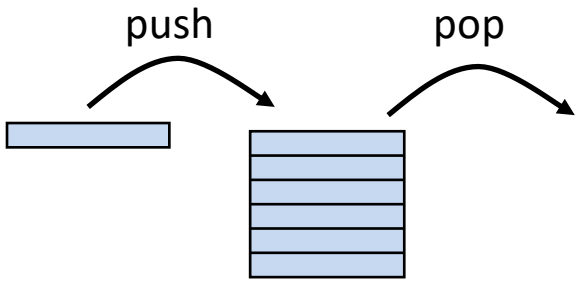
Kapitel 3: Datentyp Keller und Schlange

- Keller (Stack)
- Schlange (Queue)

Keller und seine Operationen

Definition

Ein Keller (engl. Stack; Stapel) ist eine endliche Menge von Elementen mit einer LIFO-Organisation (Last-In First-Out).



Interface Keller

- push(x) fügt x in Keller ein.
- pop() entfernt zuletzt eingefügtes („oberstes“) Kellerelement und liefert es zurück.
  Falls Keller leer, dann EmptyStackException.
- top() liefert zuletzt eingefügtes („oberstes“) Kellerelement zurück.
  Falls Keller leer, dann EmptyStackException.
- isEmpty() prüft, ob Keller leer ist.

```
public interface Stack {
    void push(int x);
    int pop();
    int top();
    boolean isEmpty();
}
```

Anwendungen (1)

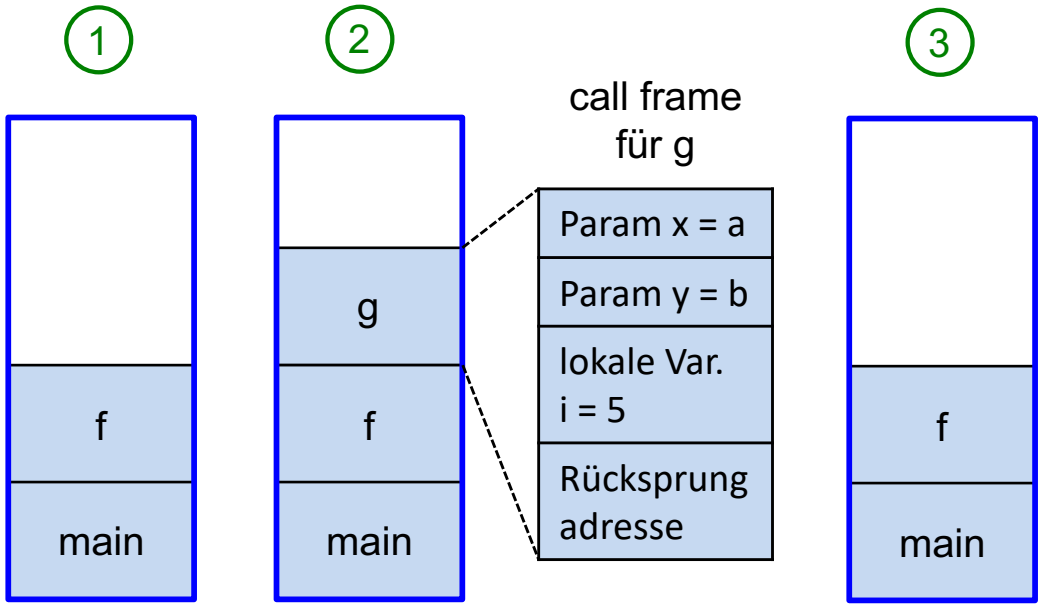
- Funktionsaufrufe zur Laufzeit
  Bei jedem Funktionsaufruf wird ein sogenannter call frame – bestehend aus Parameter, lokalen Variablen und Rücksprungadresse – in den Laufzeitkeller geschoben und beim Verlassen der Funktion wieder aus dem Keller entfernt.

```
static void main() {
    f();
}
```

```
void f() {
 1  ...
    g(a,b);
 3  ...
}
```

```
void g(int x, int y) {
 2  int i = 5;
    ...
}
```

Laufzeitkeller zu verschiedenen Zeitpunkten



Anwendungen (2)

- Rekursion
  Jede rekursive Funktion kann mit Hilfe eines Kellers in eine Funktion ohne Rekursion umgewandelt werden (später).
- Compilerbau
  Bei der syntaktischen Analyse von Programmen (Parser) spielen Keller eine zentrale Rolle.
- In einigen Programmiersprachen gehören Keller zum Sprachumfang
  Beispiel PostScript: Programme werden in Postfix-Notation geschrieben und mit Hilfe eines Kellers ausgewertet.

Überprüfung auf korrekte Klammerung

Problemstellung

- Eine beliebige Zeichenfolge (String oder ASCII-Datei) soll auf korrekte Klammerung geprüft werden.
- Zulässige Klammern: (,), [,], {, }

Beispiele

… { ... [... (...) ...] ... (...) ... } …	Korrekt geklammert
… { ... { ... } ...	Nicht korrekt geklammert
… [... (...] ...)	Nicht korrekt geklammert

"..." steht für eine beliebige Zeichenfolge ohne Klammer

Algorithmus

- Durchlaufe Zeichenfolge zeichenweise von links nach rechts:
  - Falls Zeichen eine öffnende Klammer ist, dann kellere sie ein.
  - Falls Zeichen eine schließende Klammer ist, dann prüfe, ob sie mit der obersten Klammer im Keller ein passendes Klammerpaar bildet. Falls Klammerpaar passend, dann kellere Klammer aus. Fehlermeldung, falls Klammerpaar nicht passt oder Keller leer ist.
- Fehlermeldung, falls am Ende der Zeichenfolge Keller nicht leer ist.

Keller s	Eingabe (aktuelle Klammer k in rot)	Aktion
leer	… { … (…) … […] … }	s.push(k);
{	… { … (…) … […] … }	s.push(k);
{ (*)	… { … (…) … […] … }	Da s.top() zu k passt, mache s.pop();
{	… { … (…) … […] … }	s.push(k);
{ [	… { … (…) … […] … }	Da s.top() zu k passt, mache s.pop();
{	… { … (…) … […] … }	Da s.top() zu k passt, mache s.pop();
leer	… { … (…) … […] … }	Alles OK. Fertig!

*) Jedes Zeichen ist ein Kellerelement und oberstes Kellerelement steht rechts.

Auswertung von arithmetischen Ausdrücken

Problemstellung

- Gegeben ist ein arithmetischer Ausdruck (als Zeichenfolge)
- mit Gleitkommazahlen und den üblichen Operatoren: +, −, *, /
- * und / haben eine höhere Präzedenz als + und –
- * und / bzw. + und – haben jeweils dieselbe Präzedenz und sind linksassoziativ.

Beispiele

- 1.0 / 2.0 * 2.0 ergibt 1.0
- (2 + 3 * 4 − 4) / 2 ergibt 5.0

Shift-Reduce-Parser (1)

- Zerlege Eingabe in Tokens.
  (Token = lexikalische Einheit: Klammer, Operand oder Operator)
- In Abhängigkeit vom nächsten Token und den obersten Elementen des Kellers führe shift- oder reduce-Operation durch.
- Shift-Operation: Schiebe aktuelles Token von der Eingabe in den Keller und hole nächstes Token aus der Eingabe.
- Reduce-Operation: Werte den obersten Ausdruck im Keller aus.

Stack s	Eingabe (aktuelles Token t in rot)	Aktion
$ *)	2 + 3 * 4 $ *)	shift
$ 2	2 + 3 * 4 $	shift
$ 2 +	2 + 3 * 4 $	shift
$ 2 + 3	2 + 3 * 4 $	shift
$ 2 + 3 *	2 + 3 * 4 $	shift
$ 2 + 3 * 4	2 + 3 * 4 $	reduce
$ 2 + 12	2 + 3 * 4 $	reduce
$ 14	2 + 3 * 4 $	accept

*) Im Kellerboden und am Ende der Eingabe steht das Sonderzeichen '$'.

Shift-Reduce-Parser (2)

- Die folgende Tabelle beschreibt die Aktionen des Parsers durch eine Menge von Wenn-Dann-Regeln.
- Kann keine der Regeln angewandt werden, dann ist die Eingabe fehlerhaft.

Regel	Oberste Elemente des Stacks s	Nächstes Token t in der Eingabe	Aktion
1	$ [1]	(, val [2]	shift: s.push(t); t = nextToken();
2	op [3]	(, val	shift
3	(	(, val	shift
4	(val) [4]	), op, $	reduce: ersetze im Keller ″(val)″ durch ″val″
5	$ val	$	accept: Ergebnis ist val
6	$ val	op	shift
7	(val	), op	shift
8	val_1 op val_2	), $	reduce: werte im Keller val_1 op val_2 aus
9	val_1 op_1 val_2	op_2	shift, falls op_2 eine höhere Präzedenz hat als op_1, sonst reduce (werte im Keller val_1 op val_2 aus).

[1] Im Kellerboden und am Ende der Eingabe steht das Sonderzeichen ‘$‘.

[2] val steht für einen beliebigen Operand (Gleitkommazahl).

[3] op steht für einen beliebigen Operator: +, −, *, /.

[4] oberstes Kellerelement steht rechts.

Beispiel

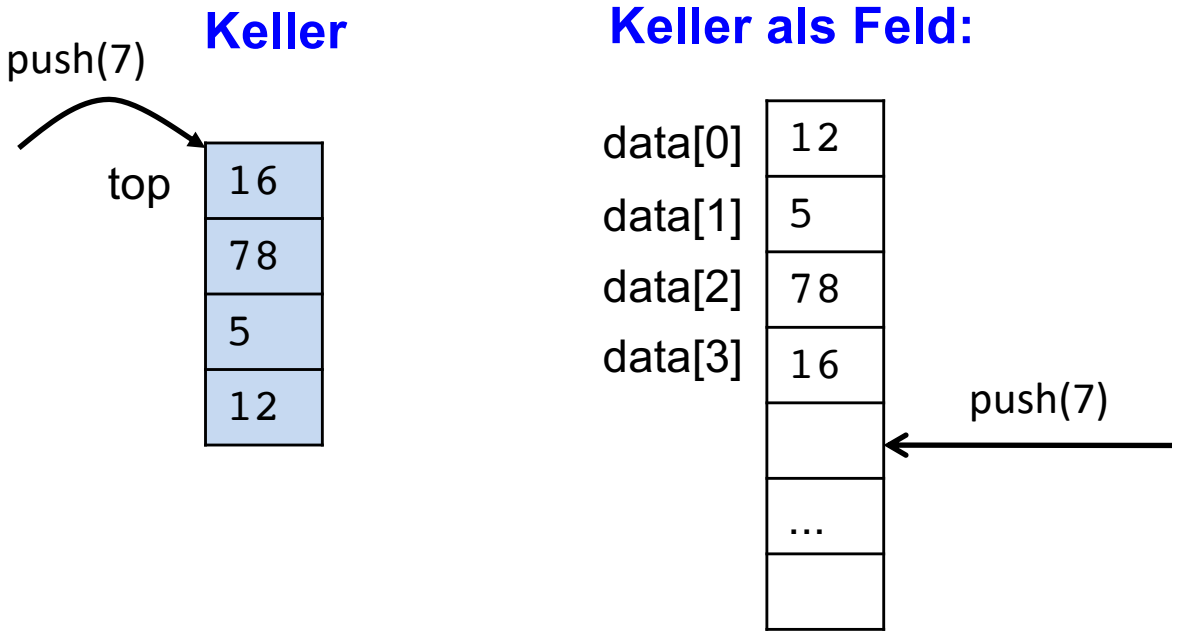
Stack s	Eingabe (aktuelles Token t rot)	Aktion
$	(2 + 3 * 4 − 4) / 2 $	shift
$ (	(2 + 3 * 4 − 4) / 2 $	shift
$ (2	(2 + 3 * 4 − 4) / 2 $	shift
$ (2 +	(2 + 3 * 4 − 4) / 2 $	shift
$ (2 + 3	(2 + 3 * 4 − 4) / 2 $	shift
$ (2 + 3 *	(2 + 3 * 4 − 4) / 2 $	shift
$ (2 + 3 * 4	(2 + 3 * 4 − 4) / 2 $	reduce
$ (2 + 12	(2 + 3 * 4 − 4) / 2 $	reduce
$ (14	(2 + 3 * 4 − 4) / 2 $	shift
$ (14 −	(2 + 3 * 4 − 4) / 2 $	shift
$ (14 − 4	(2 + 3 * 4 − 4) / 2 $	reduce
$ (10	(2 + 3 * 4 − 4) / 2 $	shift
$ (10)	(2 + 3 * 4 − 4) / 2 $	reduce
$ 10	(2 + 3 * 4 − 4) / 2 $	shift
$ 10 /	(2 + 3 * 4 − 4) / 2 $	shift
$ 10 / 2	(2 + 3 * 4 − 4) / 2 $	reduce
$ 5	(2 + 3 * 4 − 4) / 2 $	accept

Beachte:

- bei shift wird nächstes Token aus der Eingabe geholt.
- bei reduce bleibt aktuelles Token in der Eingabe.

Keller als Feld

- Die Realisierung eines Kellers als Feld ist naheliegend
- Elemente werden lückenlos im Feld gehalten.
- Keller wächst in Richtung größere Indizes.
  D.h. push, pop und top greifen auf das hintere Feldende zu.
- Falls das Feld gefüllt ist, muss das Feld vergrößert werden,
  d.h. umkopieren in ein größeres Feld



Class ArrayStack (1)

```
public class ArrayStack implements Stack{

    public ArrayList() {
        size = 0;
        data = new int[DEF_CAPACITY];
    }

    public boolean isEmpty() {return size == 0;}

    // ...

    private static final int DEF_CAPACITY = 16;
    private int size;
    private int[] data;
}
```

Class ArrayStack (2)

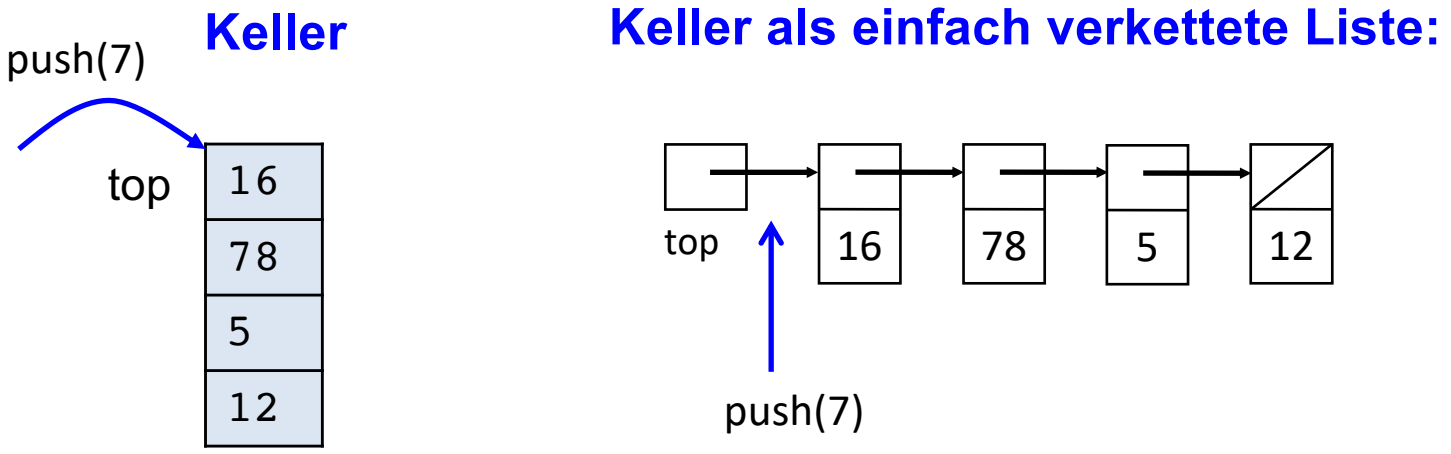
```
public class ArrayStack implements Stack {
    // ...

    public void push(int x) {
        if (data.length == size)
            data = Arrays.copyOf(data, 2*size);
        data[size++] = x;
    }

    public int top() {
        if (size == 0)
            throw new EmptyStackException();
        return data[size-1];
    }

    public int pop() {
        if (size == 0)
            throw new EmptyStackException();
        return data[--size];
    }

    @Override public String toString() {...}
}
```

Feld verdoppeln, falls kein Platz mehr.

Wie bei ArrayList

Datentyp Keller als einfach verkettete Liste

- Realisierung eines Kellers als linear verkettete Liste (ohne Hilfskopfknoten)
- top ist vorderster Knoten der Liste



Class LinkedStack (1)

```
public class LinkedStack implements Stack{

    public LinkedStack() {
        top = null;
    }

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

    // ...

    private static class Node {
        int data;
        Node next;
        Node(int x, Node p) {
            data = x;
            next = p;
        }
    }
    private Node top;
}
```

Class LinkedStack (2)

```
public class LinkedStack implements Stack {
    // ...
    public void push(int x) {
        top = new Node(x,top);
    }

    public int top() {
        if (top == null)
            throw new EmptyStackException();
        return top.data;
    }

    public int pop() {
        if (top == null)
            throw new EmptyStackException();
        int x = top.data;
        top = top.next;
        return x;
    }

    @Override public String toString() {...}
}
```

Wie bei LinkedList

Stack-Anwendung

```
public class StackApplication {

    public static void main(String[] args) {

        Scanner in = new Scanner(System.in);
        int d  = in.nextInt();

        Stack s;

        if (d == 0)
            s = new ArrayStack();
        else
            s = new LinkedStack();

        s.push(3);
        s.push(2);
        s.push(1);

        System.out.println(s);
    }
}
```

3, 2, 1 = top

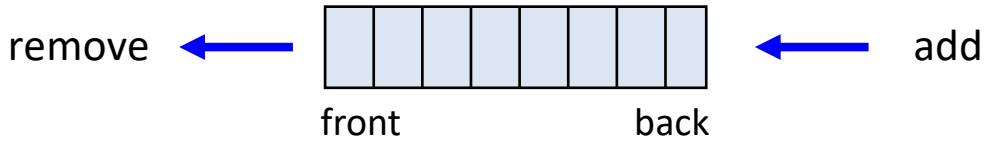
Kapitel 3: Datentyp Keller und Schlange

- Keller (Stack)
- Schlange (Queue)

Schlange und ihre Operationen

Definition

Eine Schlange (engl. Queue) ist eine endliche Menge von Elementen mit einer FIFO-Organisation (First-In First-Out).



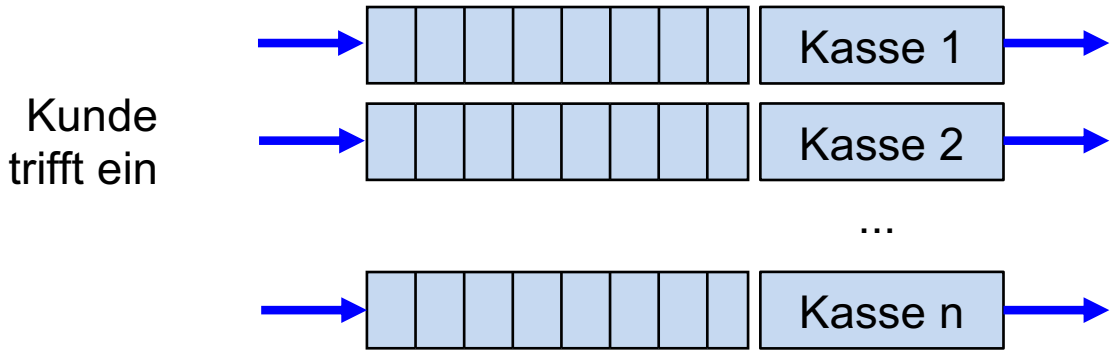
Interface Schlange

- add(x) fügt x in Schlange hinten an.
- remove() entfernt vorderstes Element der Schlange und liefert es zurück. Falls Schlange leer, dann NoSuchElementException .
- front() liefert vorderstes Element der Schlange zurück. Falls Schlange leer, dann NoSuchElementException.
- isEmpty() prüft, ob Schlange leer ist.

```
public interface Queue {
    void add(int x);
    int remove();
    int front();
    boolean isEmpty();
}
```

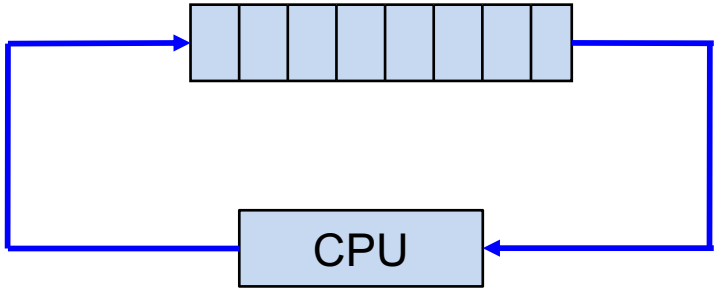
Anwendungen

Warteschlangen in ereignisorientierter Simulation



Round-Robin-Scheduler in Betriebssystemen

Warteschlange mit rechenbereiten Prozessen

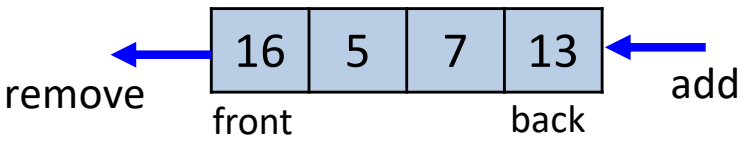


In regelmäßigen Zeitabständen wird der nächste bereite Prozess der CPU zugeteilt.

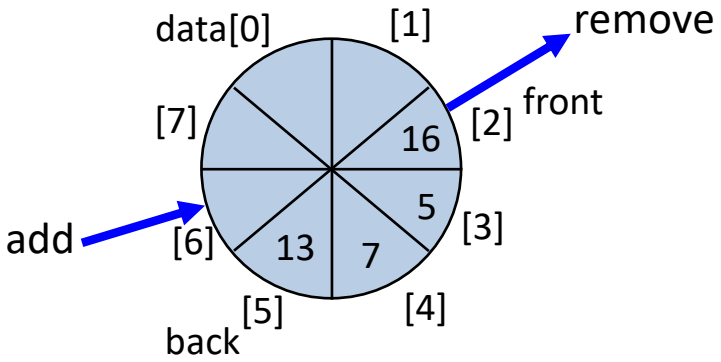
Schlange als zirkuläres Feld

- Elemente werden lückenlos im Feld gehalten.
- front- und back-Index definieren den Anfang und das Ende der Schlange.
- Da nach einer Folge von add- (und evtl. remove-) Operationen das Feldende erreicht wird, wird mit einer modolo-Rechnung wieder bei Index 0 begonnen (zirkuläres Feld).
- Falls das Feld gefüllt ist muss das Feld vergrößert werden, d.h. umkopieren in ein größeres Feld

Schlange



Schlange als zirkuläres Feld:



class ArrayQueue (1)

```
public class ArrayQueue implements Queue{

    public ArrayQueue() {
        size = 0;
        back = DEF_CAPACITY-1;
        front = 0;
        data = new int[DEF_CAPACITY];
    }

    public boolean isEmpty() {return size == 0;}

    // ...

    private static final int DEF_CAPACITY = 16;
    private int size;
    private int back;
    private int front;
    private int[] data;
}
```

Bei leerer Schlange muss front == back+1 mod n sein, wobei n = DEF_CAPACITY die Feldgröße ist.

size ist die Anzahl der Elemente in der Schlange

class ArrayQueue (2)

```
public class ArrayQueue implements Queue {
    // ...
    public void add(int x) {
        if (data.length == size)
            resize(2*size);
        back = (back + 1) % data.length;
        data[back] = x;
        size++;
    }
    public int front() {
        if (size == 0)
            throw new NoSuchElementException();
        return data[front];
    }

    public int remove() {
        if (size == 0)
            throw new NoSuchElementException();
        int x = data[front];
        front = (front + 1) % data.length;
        size--;
        return x;
    }
    //...
```

Feld verdoppeln, falls kein Platz mehr.

Warum nicht einfach back++?
(back + 1) wird nie >= data.length

Weil wenn front oder back am Ende der Liste sind, dann sollen sie zu Position 0 zurückkehren

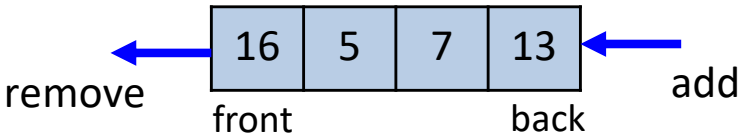
class ArrayQueue (2)

```
public class ArrayQueue implements Queue {
    // ...

    private void resize(int newLength) {
        assert size == data.length && newLength > size;

        int[] old = data;
        data = new int[newLength];
        System.arraycopy(old,front,data,0,old.length-front);
        if (front > 0)
            System.arraycopy(old,0,data,old.length-front,back+1);
        front = 0;
        back = size-1;
    }

    public String toString() {
        StringBuilder s = new StringBuilder("");
        for (int i = 0; i < size; i++) {
            s.append(data[(front+i)%data.length]).append(", ");
        }
        s.append("size = ").append(size);
        return s.toString();
    }
}
```

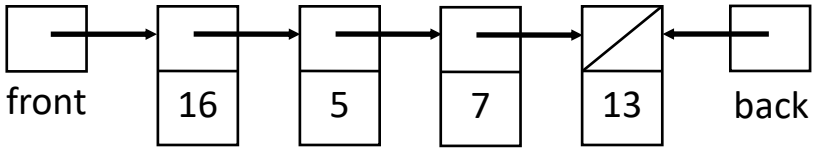
Schlange als einfach verkettete Liste

- Realisierung einer Schlange als einfach verkettete Liste (ohne Hilfskopfknoten)
- Zwei Referenzen:
  front zeigt auf ersten und back auf letzten Knoten.
  Damit lassen sich add und remove effizient realisieren.

Schlange



Schlange als einfach verkettete Liste:



Class LinkedQueue (1)

```
public class LinkedQueue implements Queue {

    public LinkedQueue() {
        front = back = null;
    }

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

    // ...

    private static class Node {
        int data;
        Node next;
        Node(Node p, int x) {
            data = x;
            next = p;
        }
    }
    private Node front;
    private Node back;
}
```

Class LinkedQueue (2)

```
public class LinkedQueue implements Queue {

    // ...

    public void add(int x) {
        if (front == null)
            front = back = new Node(x,null);
        else {
            back.next = new Node(x,null);
            back = back.next;
        }
    }

    public int front() {
        if (front == null)
            throw new NoSuchElementException();
        return front.data;
    }
```

Class LinkedQueue (3)

```
public class LinkedQueue implements Queue {

    // ...

    public int remove() {
        if (front == null)
            throw new NoSuchElementException();
        int x = front.data;
        front = front.next;
        if (front == null)
            back = null;
        return x;
    }

    @Override public String toString() {...}
}
```

Wie bei LinkedList

Queue-Anwendung

```
public class QueueApplication {

    public static void main(String[] args) {

        Scanner in = new Scanner(System.in);
        int d  = in.nextInt();

        Queue q;

        if (d == 0)
            q = new ArrayQueue();
        else
            q = new LinkedQueue();

        q.add(1);
        q.add(2);
        q.add(3);
        q.add(4);
        int x = q.remove();

        System.out.println(q);
    }
}
```

front = 2, 3, 4 = back