Informatik II

Übung 7

Giuseppe Accaputo, Felix Friedrich, Patrick Gruntz, Tobias Klenze, Max Rossmannek, David Sidler, Thilo Weghorn

FS 2017

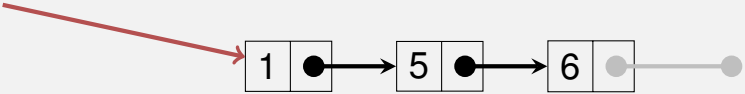
Heutiges Programm

1. Wiederholung: Stack
2. Queue
3. Allgemeiner Zirkulärer Buffer

Implementation Push in Java

```
class Stack{
ListNode top_node;
  void push (int value){
    top_node = new ListNode (value, top_node);
  }
}

push(6);
```

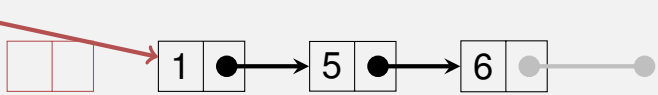
top_node



Implementation Push in Java

```
class Stack{
ListNode top_node;
  void push (int value){
    top_node = new ListNode (value, top_node);
  }
}

push(6);
```

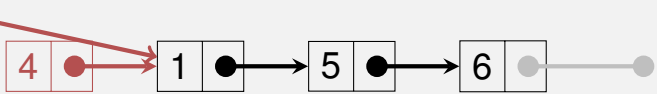
top_node



Implementation Push in Java

```
class Stack{
ListNode top_node;
  void push (int value){
    top_node = new ListNode (value, top_node);
  }
}

push(6);
```

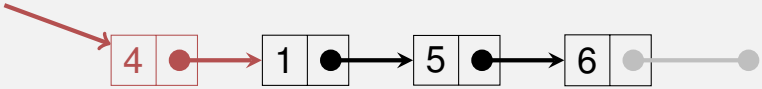
top_node



Implementation Push in Java

```
class Stack{
ListNode top_node;
  void push (int value){
    top_node = new ListNode (value, top_node);
  }
}

push(6);
```

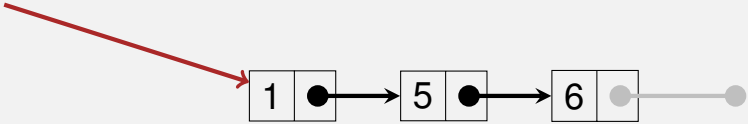
top_node



Implementation Pop in Java

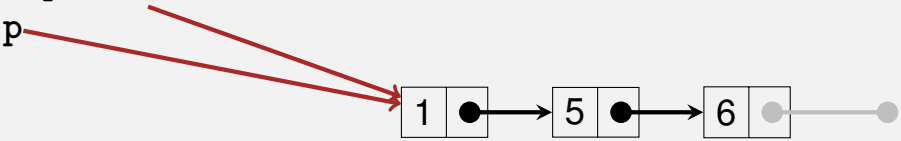
```
int pop()
{
    assert (!empty());
    ListNode p = top_node;
    top_node = top_node.next;
    return p.value;
}
```

top_node



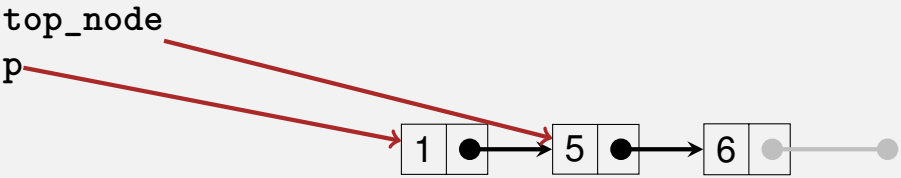
Implementation Pop in Java

```
int pop()
{
    assert (!empty());
    ListNode p = top_node;
    top_node = top_node.next;
    return p.value;
}
```



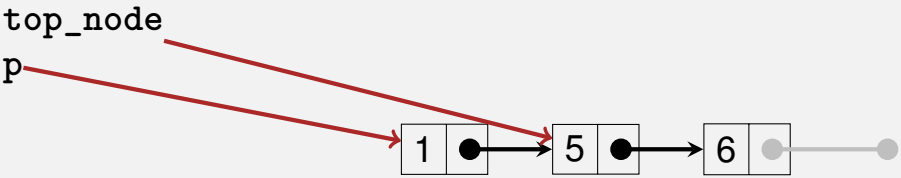
Implementation Pop in Java

```
int pop()
{
    assert (!empty());
    ListNode p = top_node;
    top_node = top_node.next;
    return p.value;
}
```



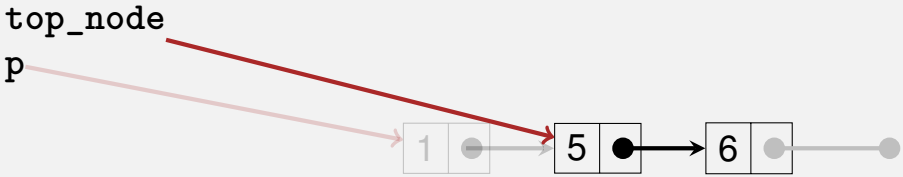
Implementation Pop in Java

```
int pop()
{
    assert (!empty());
    ListNode p = top_node;
    top_node = top_node.next;
    return p.value;
}
```



Implementation Pop in Java

```
int pop()
{
    assert (!empty());
    ListNode p = top_node;
    top_node = top_node.next;
    return p.value;
}
```



Queue (Schlange / Warteschlange / Fifo)

Queue ist ein ADT mit folgenden Operationen:

Queue (Schlange / Warteschlange / Fifo)

Queue ist ein ADT mit folgenden Operationen:

- `enqueue`(x, Q): fügt x am Ende der Schlange an.

Queue (Schlange / Warteschlange / Fifo)

Queue ist ein ADT mit folgenden Operationen:

- `enqueue`(x, Q): fügt x am Ende der Schlange an.
- `dequeue`(Q): entfernt x vom Beginn der Schlange und gibt x zurück (`null` sonst.)

Queue (Schlange / Warteschlange / Fifo)

Queue ist ein ADT mit folgenden Operationen:

- `enqueue`(x, Q): fügt x am Ende der Schlange an.
- `dequeue`(Q): entfernt x vom Beginn der Schlange und gibt x zurück (`null` sonst.)
- `head`(Q): liefert das Objekt am Beginn der Schlage zurück (`null` sonst.)

Queue (Schlange / Warteschlange / Fifo)

Queue ist ein ADT mit folgenden Operationen:

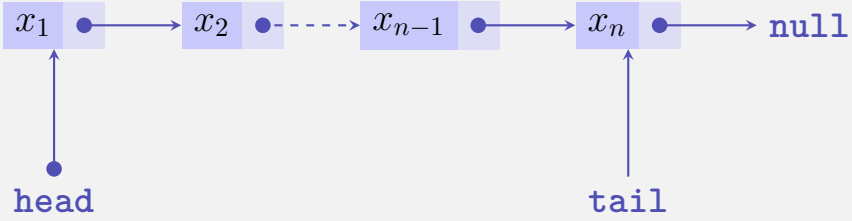
- `enqueue`(x, Q): fügt x am Ende der Schlange an.
- `dequeue`(Q): entfernt x vom Beginn der Schlange und gibt x zurück (`null` sonst.)
- `head`(Q): liefert das Objekt am Beginn der Schlage zurück (`null` sonst.)
- `empty`(Q): liefert `true` wenn Queue leer, sonst `false`.

Queue (Schlange / Warteschlange / Fifo)

Queue ist ein ADT mit folgenden Operationen:

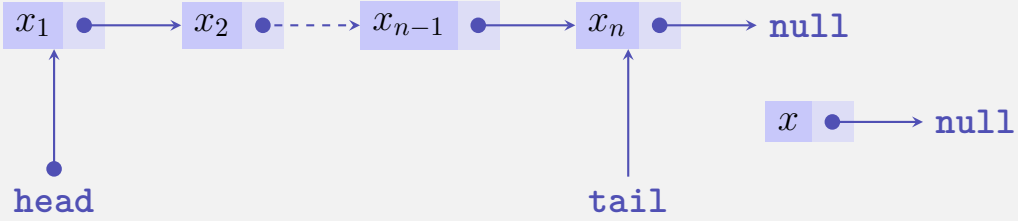
- `enqueue`(x, Q): fügt x am Ende der Schlange an.
- `dequeue`(Q): entfernt x vom Beginn der Schlange und gibt x zurück (`null` sonst.)
- `head`(Q): liefert das Objekt am Beginn der Schlage zurück (`null` sonst.)
- `empty`(Q): liefert `true` wenn Queue leer, sonst `false`.
- `emptyQueue`$()$: liefert leere Queue zurück.

Queue (FIFO)



`enqueue`(x, S):

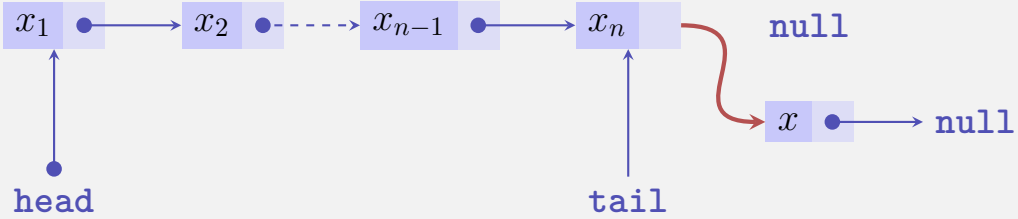
Queue (FIFO)



`enqueue`(x, S):

1. Erzeuge neues Listenelement mit x und Referenz auf `null`.

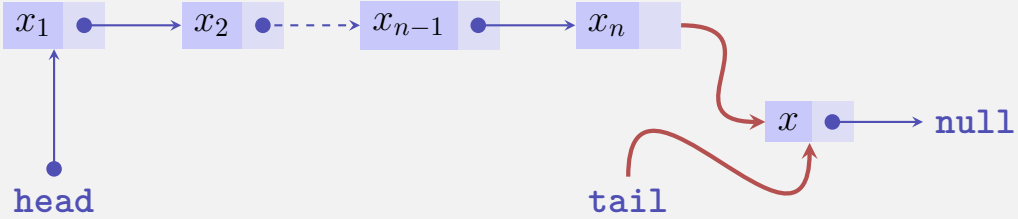
Queue (FIFO)



`enqueue`(x, S):

1. Erzeuge neues Listenelement mit x und Referenz auf `null`.
2. Wenn `tail` $\neq$ `null` , setze `tail.next` auf den Knoten mit x.

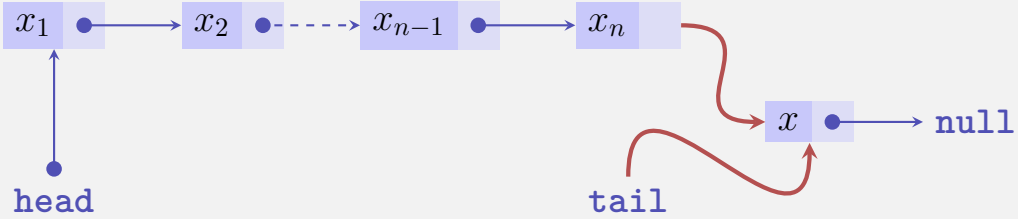
Queue (FIFO)



`enqueue`(x, S):

1. Erzeuge neues Listenelement mit x und Referenz auf `null`.
2. Wenn `tail` $\neq$ `null` , setze `tail.next` auf den Knoten mit x.
3. Setze `tail` auf den Knoten mit x.

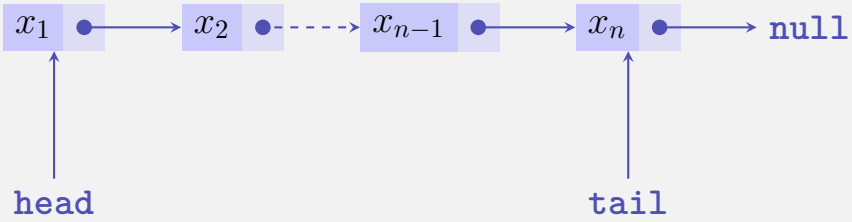
Queue (FIFO)



`enqueue`(x, S):

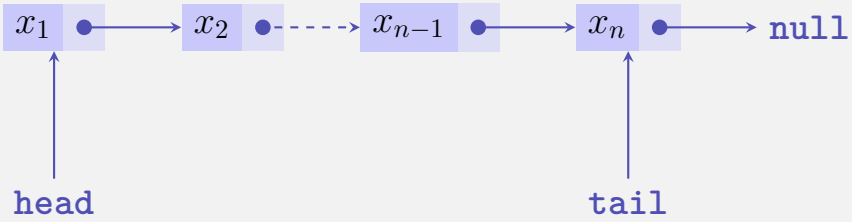
1. Erzeuge neues Listenelement mit x und Referenz auf `null`.
2. Wenn `tail` $\neq$ `null` , setze `tail.next` auf den Knoten mit x.
3. Setze `tail` auf den Knoten mit x.
4. Ist `head` $=$ `null`, dann setze `head` auf `tail`.

Invarianten!



Mit dieser Implementation gilt

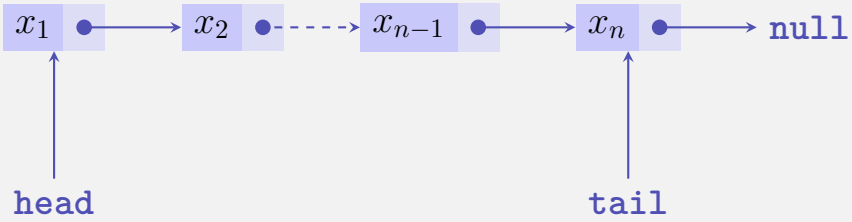
Invarianten!



Mit dieser Implementation gilt

- entweder `head` $=$ `tail` $=$ `null`,

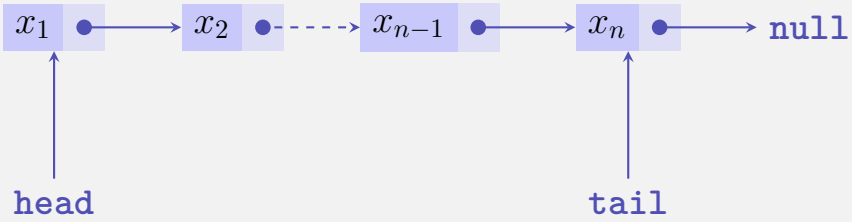
Invarianten!



Mit dieser Implementation gilt

- entweder `head` $=$ `tail` $=$ `null`,
- oder `head` $=$ `tail` $\neq$ `null` und `head.next` $=$ `null`

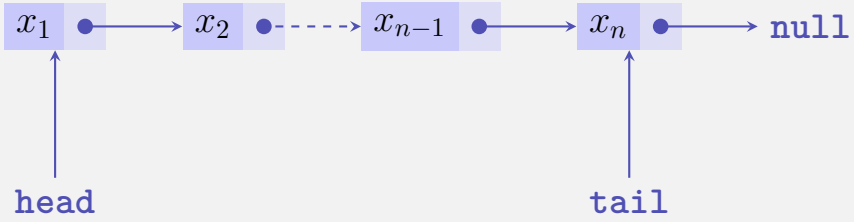
Invarianten!



Mit dieser Implementation gilt

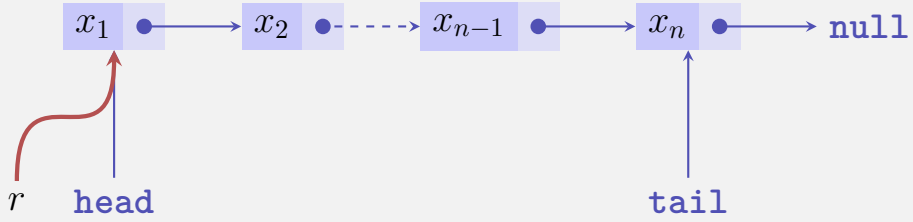
- entweder `head` $=$ `tail` $=$ `null`,
- oder `head` $=$ `tail` $\neq$ `null` und `head.next` $=$ `null`
- oder `head` $\neq$ `null` und `tail` $\neq$ `null` und `head` $\neq$ `tail` und `head.next` $\neq$ `null`.

Implementation Queue



`dequeue`(S):

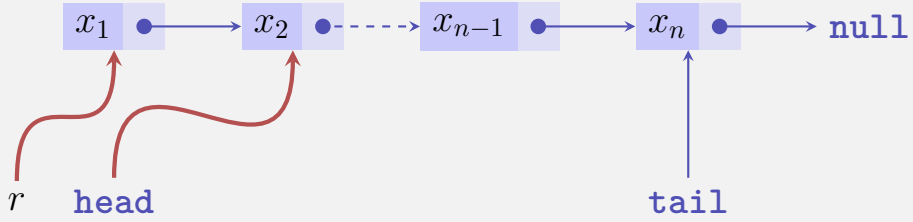
Implementation Queue



`dequeue`(S):

1. Merke Referenz von `head` in r. Wenn $r =$ `null`, gib r zurück.

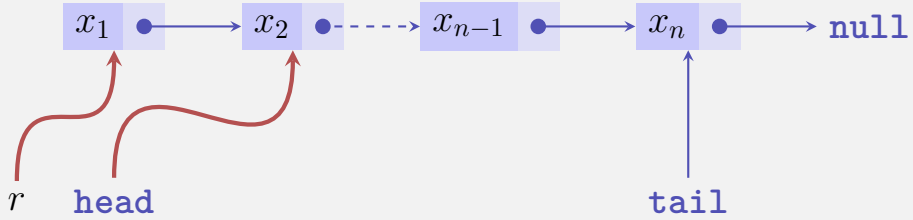
Implementation Queue



`dequeue`(S):

1. Merke Referenz von `head` in r. Wenn $r =$ `null`, gib r zurück.
2. Setze den Referenz von `head` auf `head.next`.

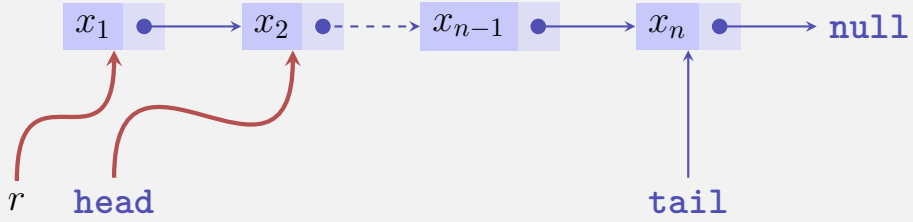
Implementation Queue



`dequeue`(S):

1. Merke Referenz von `head` in r. Wenn $r =$ `null`, gib r zurück.
2. Setze den Referenz von `head` auf `head.next`.
3. Ist nun `head` $=$ `null`, dann setze tail auf `null`.

Implementation Queue



`dequeue`(S):

1. Merke Referenz von `head` in r. Wenn $r =$ `null`, gib r zurück.
2. Setze den Referenz von `head` auf `head.next`.
3. Ist nun `head` $=$ `null`, dann setze tail auf `null`.
4. Gib den Wert von r zurück.

Analyse

Jede der Operationen `enqueue`, `dequeue`, `head` und `empty` auf der Queue ist in $\mathcal{O}(1)$ Schritten ausführbar.

Andere Variante mit begrenztem Speicherplatz

Zirkulärer FIFO[1]-Buffer mit Kapazität $n > 0$, welcher stets bis zu n Datenpunkte vorhalten und in der Einfügereihenfolge zurückgeben kann.

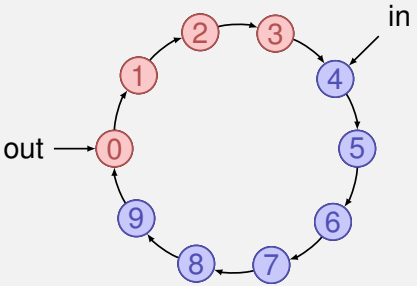
[1] First-In-First-Out

Interface

```
public class CircularBuffer {
    // Konstruktor: zirkulaerer Buffer mit Kapazit\"{a}t n
    public CircularBuffer(int n);
    // post: Rueckgabe ob kein Element (mehr) im Buffer
    public boolean empty();
    // post: Rueckgabe ob kein Platz mehr im Buffer
    public boolean full ();
    // pre: !empty()
    // post: Rueckgabe des zuerst gespeicherten Elements.
    public double get();
    // pre: ! full ()
    // post: Element ablegen
    public void put(double x);
}
```

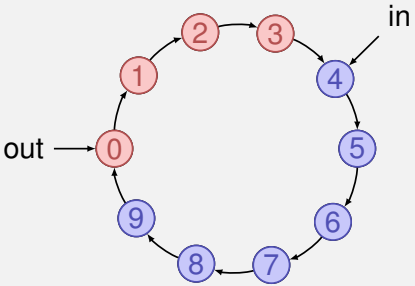
Allgemeiner zirkulärer Buffer

```
public class CircularBuffer {
  private double data[];   // Puffer
  private int in = 0;      // in-Position
  private int out = 0;     // out position

  // Konstruktor
  CircularBuffer(int n) {
    data = new double[n+1]; // warum nicht n?
  }

  ...
}
```



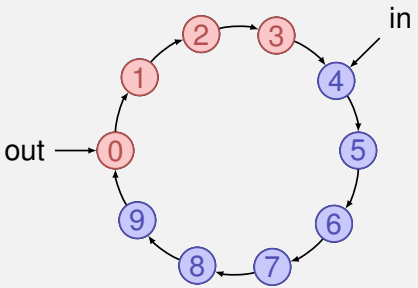
Allgemeiner zirkulärer Buffer

```
public class CircularBuffer {
   ...
  // post: Rueckgabe ob Buffer leer
  boolean empty(){
    return in == out;
  }
  // post: Rueckgabe ob Buffer voll
  boolean full(){
    return (in+1) % data.length == out;
  }
   ...
}
```



Allgemeiner zirkulärer Buffer

```
...
// pre: !full()
// post: Element ablegen
public void put(double x){
  assert(!full());
  data[in] = x;
  in = (in + 1) % data.length;
}
// pre: !empty()
// post: Rueckgabe des zuerst gespeicherten Elements.
public double get(){
  assert(!empty());
  double x = data[out];
  out = (out + 1) % data.length;
  return x;
}
```



Fragen oder Anregungen?