

# 13. Dynamische Datenstrukturen

Verkettete Listen, Abstrakte Datentypen Stapel, Warteschlange ,  
Sortierte Liste

# Motivation: Stapel



# Motivation: Stapel

3
5
1
2

# Motivation: Stapel

3
5
1
2

push(4)

4
3
5
1
2

# Motivation: Stapel

3
5
1
2

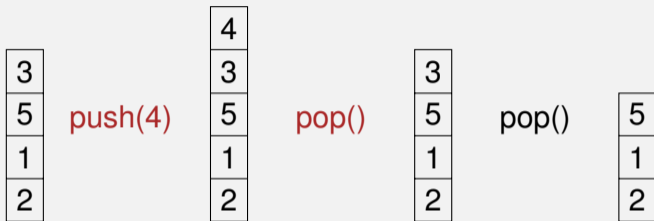
push(4)

4
3
5
1
2

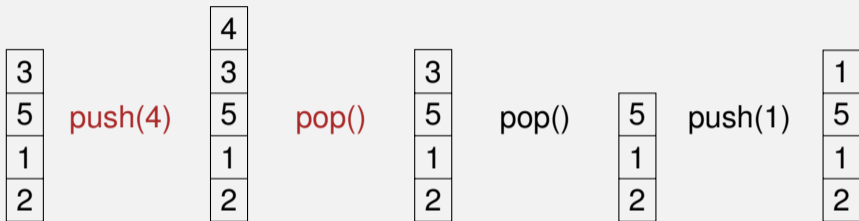
pop()

3
5
1
2

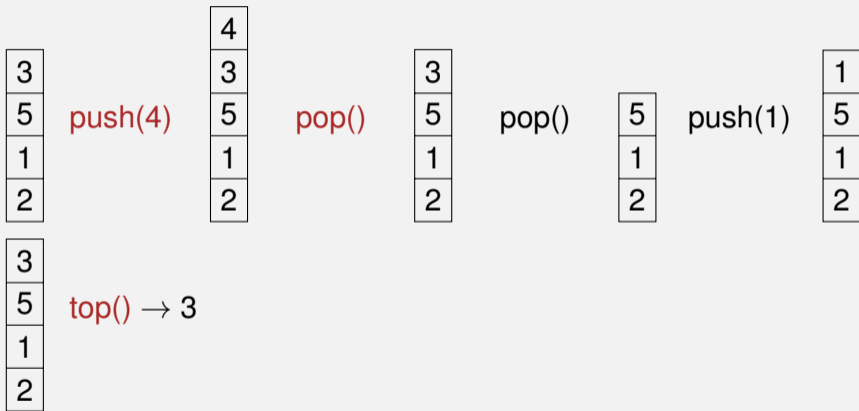
# Motivation: Stapel



# Motivation: Stapel

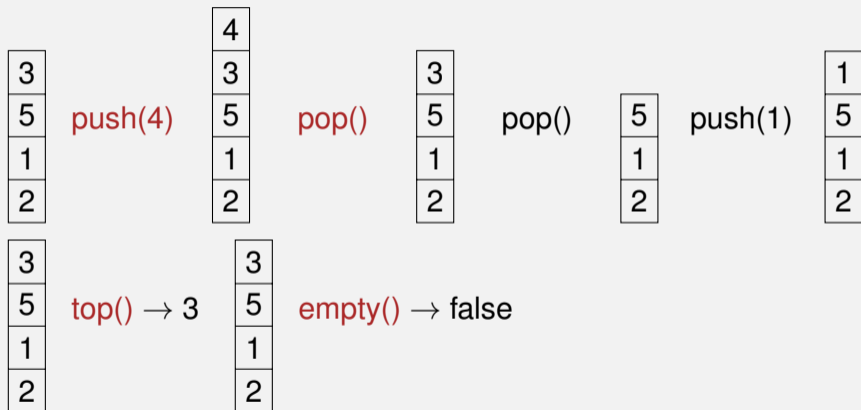


# Motivation: Stapel

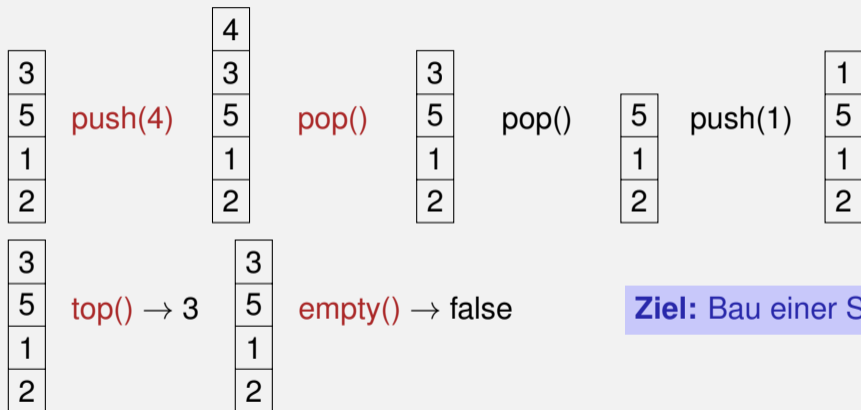




# Motivation: Stapel (push, pop, top, empty)

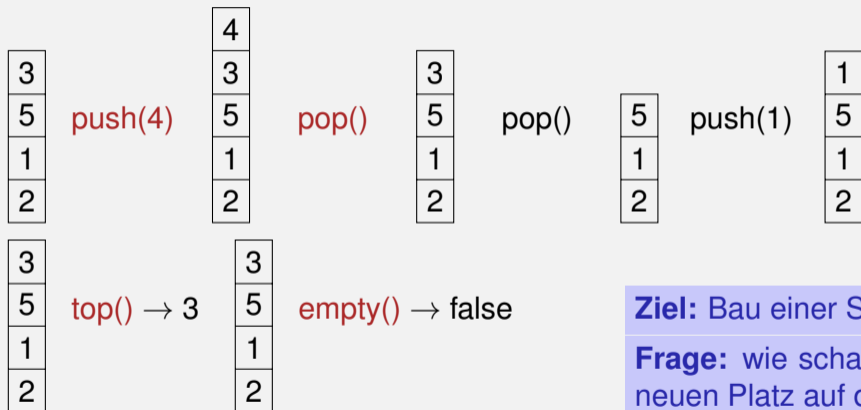


# Motivation: Stapel (push, pop, top, empty)



**Ziel: Bau einer Stapel-Klasse!**

# Motivation: Stapel (push, pop, top, empty)



**Ziel:** Bau einer Stapel-Klasse!

**Frage:** wie schaffen wir bei push neuen Platz auf dem Stapel?

# Wir brauchen einen neuen Container!

Container bisher: `Array (T [])`

# Wir brauchen einen neuen Container!

Container bisher: Array (T [])

- Zusammenhängender Speicherbereich, wahlfreier Zugriff (auf  $i$ -tes Element)

1	5	6	3	8	9	3	3	8	9
---	---	---	---	---	---	---	---	---	---

# Wir brauchen einen neuen Container!

Container bisher: `Array (T [])`

- Zusammenhängender Speicherbereich, wahlfreier Zugriff (auf  $i$ -tes Element)
- Simulation eines Stapels durch ein Array?



# Wir brauchen einen neuen Container!

Container bisher: `Array (T [])`

- Zusammenhängender Speicherbereich, wahlfreier Zugriff (auf  $i$ -tes Element)
- Simulation eines Stapels durch ein Array?
- Nein, irgendwann ist das Array “voll.”



Hier kein `push(3)` möglich!

# Arrays können wirklich nicht alles...

- Einfügen oder Löschen von Elementen „in der Mitte“ ist aufwändig.

1	5	6	3	8	9	3	3	8	9
---	---	---	---	---	---	---	---	---	---

↑  
8

Wollen wir hier einfügen, müssen wir alles rechts davon verschieben (falls da überhaupt noch Platz ist!)



# Arrays können wirklich nicht alles...

- Einfügen oder Löschen von Elementen „in der Mitte“ ist aufwändig.



↑  
8

Wollen wir hier einfügen, müssen wir alles rechts davon verschieben (falls da überhaupt noch Platz ist!)

# Arrays können wirklich nicht alles...

- Einfügen oder Löschen von Elementen „in der Mitte“ ist aufwändig.

1	5	6	3	8	8	9	3	3	8	9
---	---	---	---	---	---	---	---	---	---	---

Wollen wir hier einfügen, müssen wir alles rechts davon verschieben (falls da überhaupt noch Platz ist!)

# Arrays können wirklich nicht alles...

- Einfügen oder Löschen von Elementen „in der Mitte“ ist aufwändig.



Wollen wir hier löschen,  
müssen wir alles rechts  
davon verschieben

# Arrays können wirklich nicht alles...

- Einfügen oder Löschen von Elementen „in der Mitte“ ist aufwändig.



Wollen wir hier löschen,  
müssen wir alles rechts  
davon verschieben

# Arrays können wirklich nicht alles...

- Einfügen oder Löschen von Elementen „in der Mitte“ ist aufwändig.

1	5	6	8	8	9	3	3	8	9
---	---	---	---	---	---	---	---	---	---

Wollen wir hier löschen,  
müssen wir alles rechts  
davon verschieben

# Der neue Container: Verkettete Liste

- *Kein* zusammenhängender Speicherbereich und *kein* wahlfreier Zugriff



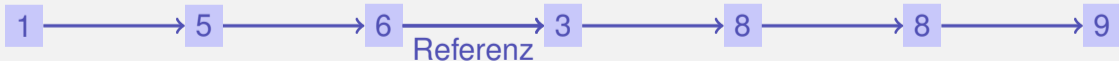
# Der neue Container: Verkettete Liste

- *Kein* zusammenhängender Speicherbereich und *kein* wahlfreier Zugriff
- Jedes Element “kennt” seinen Nachfolger



# Der neue Container: Verkettete Liste

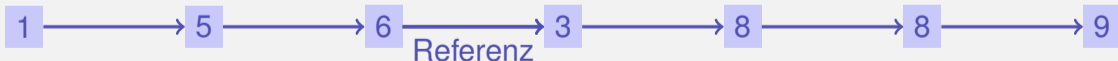
- *Kein* zusammenhängender Speicherbereich und *kein* wahlfreier Zugriff
- Jedes Element “kennt” seinen Nachfolger





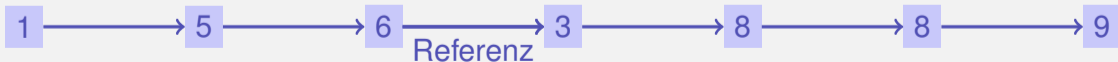
# Der neue Container: Verkettete Liste

- *Kein* zusammenhängender Speicherbereich und *kein* wahlfreier Zugriff
- Jedes Element “kennt” seinen Nachfolger
- Einfügen und Löschen beliebiger Elemente ist einfach, *auch am Anfang der Liste*



# Der neue Container: Verkettete Liste

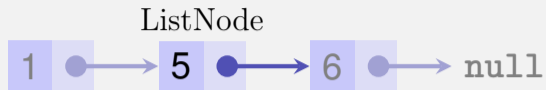
- *Kein* zusammenhängender Speicherbereich und *kein* wahlfreier Zugriff
- Jedes Element “kennt” seinen Nachfolger
- Einfügen und Löschen beliebiger Elemente ist einfach, *auch am Anfang der Liste*
- ⇒ Ein Stapel kann als verkettete Liste realisiert werden



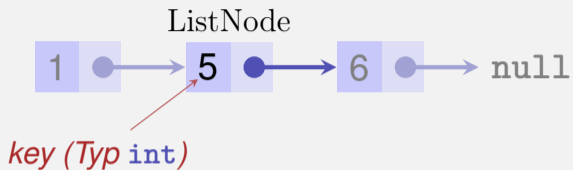
# Verkettete Liste: Zoom



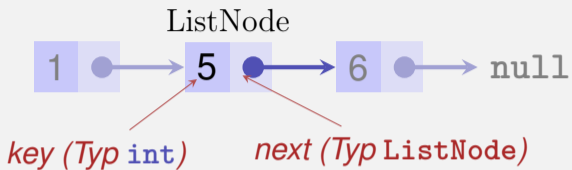
# Verkettete Liste: Zoom



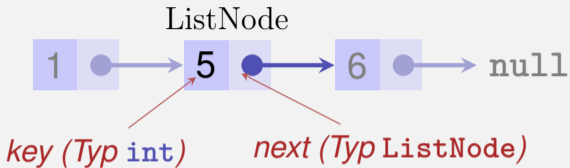
# Verkettete Liste: Zoom



# Verkettete Liste: Zoom

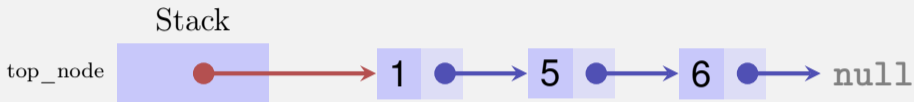


# Verkettete Liste: Zoom



```
class ListNode {  
    int key;  
    ListNode next;  
  
    ListNode (int key, ListNode next){  
        this.key = key;  
        this.next = next;  
    }  
}
```

# Stapel = Referenz aufs oberste Element



```
public class Stack {  
    private ListNode top_node;  
  
    public void push (int value) {...}  
};
```



# Abstrakte Datentypen

Ein *Stack* ist ein abstrakter Datentyp (ADT) mit Operationen

# Abstrakte Datentypen

Ein *Stack* ist ein abstrakter Datentyp (ADT) mit Operationen

- **push**( $x, S$ ): Legt Element  $x$  auf den Stapel  $S$ .

# Abstrakte Datentypen

Ein *Stack* ist ein abstrakter Datentyp (ADT) mit Operationen

- **push**( $x, S$ ): Legt Element  $x$  auf den Stapel  $S$ .
- **pop**( $S$ ): Entfernt und liefert oberstes Element von  $S$ , oder **null** (oder Fehlermeldung).

# Abstrakte Datentypen

Ein *Stack* ist ein abstrakter Datentyp (ADT) mit Operationen

- **push**( $x, S$ ): Legt Element  $x$  auf den Stapel  $S$ .
- **pop**( $S$ ): Entfernt und liefert oberstes Element von  $S$ , oder **null** (oder Fehlermeldung).
- **top**( $S$ ): Liefert oberstes Element von  $S$ , oder **null** (oder Fehlermeldung).

# Abstrakte Datentypen

Ein *Stack* ist ein abstrakter Datentyp (ADT) mit Operationen

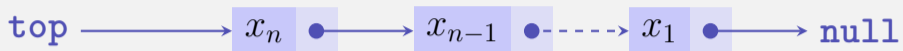
- **push**( $x, S$ ): Legt Element  $x$  auf den Stapel  $S$ .
- **pop**( $S$ ): Entfernt und liefert oberstes Element von  $S$ , oder **null** (oder Fehlermeldung).
- **top**( $S$ ): Liefert oberstes Element von  $S$ , oder **null** (oder Fehlermeldung).
- **empty**( $S$ ): Liefert **true** wenn Stack leer, sonst **false**.

# Abstrakte Datentypen

Ein *Stack* ist ein abstrakter Datentyp (ADT) mit Operationen

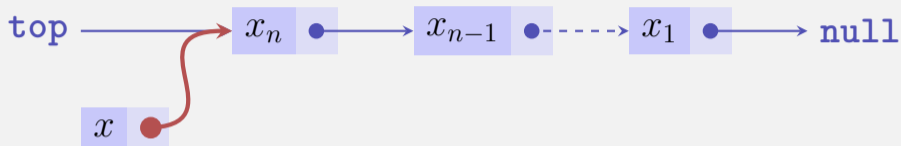
- **push**( $x, S$ ): Legt Element  $x$  auf den Stapel  $S$ .
- **pop**( $S$ ): Entfernt und liefert oberstes Element von  $S$ , oder **null** (oder Fehlermeldung).
- **top**( $S$ ): Liefert oberstes Element von  $S$ , oder **null** (oder Fehlermeldung).
- **empty**( $S$ ): Liefert **true** wenn Stack leer, sonst **false**.
- **emptyStack**(): Liefert einen leeren Stack.

# Implementation Push



`push( $x, S$ ):`

# Implementation Push

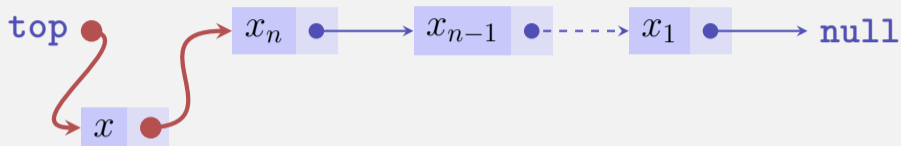


`push`( $x, S$ ):

- 1 Erzeuge neues Listenelement mit  $x$  und Referenz auf den Wert von `top`.



# Implementation Push



`push`( $x, S$ ):

- 1 Erzeuge neues Listenelement mit  $x$  und Referenz auf den Wert von `top`.
- 2 Setze `top` auf den Knoten mit  $x$ .

# Implementation Push in Java

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

push(4);

top\_node

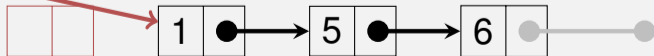


# Implementation Push in Java

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

push(4);

top\_node

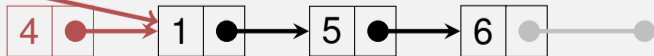


# Implementation Push in Java

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

push(4);

top\_node

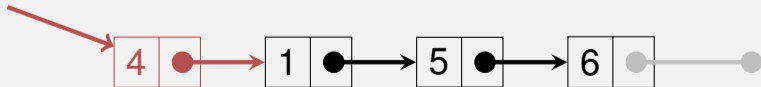


# Implementation Push in Java

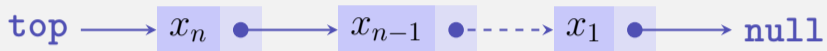
```
class Stack{  
  ListNode top_node;  
  void push (int value){  
    top_node = new ListNode (value, top_node);  
  }  
}
```

push(4);

top\_node

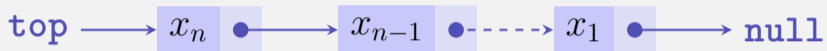


# Implementation Pop



`pop(S)`:

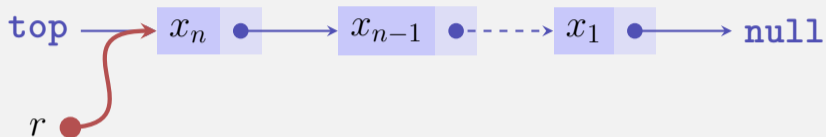
# Implementation Pop



`pop(S)`:

- 1 Ist `top=null`, dann gib `null` zurück (oder Fehlermeldung).

# Implementation Pop

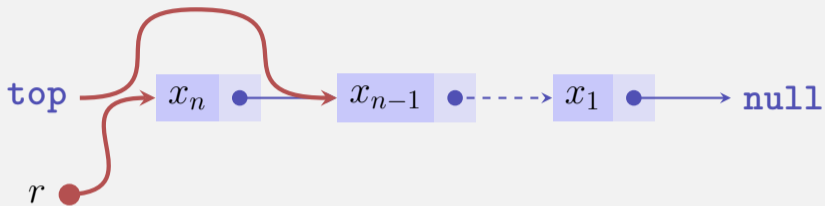


`pop(S)`:

- 1 Ist `top=null`, dann gib `null` zurück (oder Fehlermeldung).
- 2 Andernfalls merke Referenz  $p$  von `top` in  $r$ .



# Implementation Pop



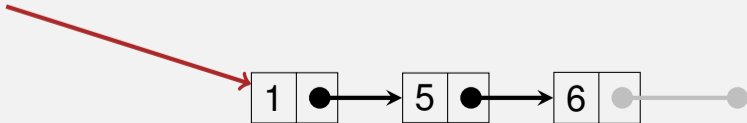
`pop(S)`:

- 1 Ist `top=null`, dann gib `null` zurück (oder Fehlermeldung).
- 2 Andernfalls merke Referenz  $p$  von `top` in  $r$ .
- 3 Setze `top` auf  $p.next$  und gib  $r$  zurück

# Implementation Pop in Java

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

top\_node



# Implementation Pop in Java

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

top\_node

p



# Implementation Pop in Java

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

top\_node

p



# Implementation Pop in Java

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

top\_node

p

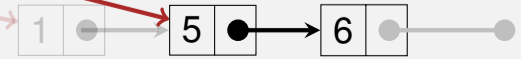


# Implementation Pop in Java

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

top\_node

p



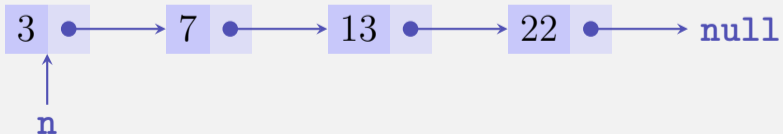
# Sortierte Verkettete Liste

Erwünschte Funktionalität:

- (Sortierte) Ausgabe
- Hinzufügen eines Wertes
- (Suchen eines Wertes)
- Entfernen eines Wertes

# ListNode

```
class ListNode{  
    int key;  
    ListNode next;  
  
    ListNode (int k, ListNode n){  
        key = k;  
        next = n;  
    }  
}
```





# ListNode mit Getter/Setter Methoden

```
public class ListNode{
    private ListNode next;
    private int key;

    ListNode (int k, ListNode nxt){
        key = k; next = nxt;
    }

    public void setNext(ListNode next){
        this.next = next;
    }

    public ListNode getNext(){
        return next;
    }

    public int getKey(){
        return key;
    }
}
```

# Invarianten!

$n$

Für eine Referenz  $n$  auf einen Knoten in einer sortierten Liste gilt

# Invarianten!

↑  
`n`

Für eine Referenz `n` auf einen Knoten in einer sortierten Liste gilt

- entweder `n = null`,

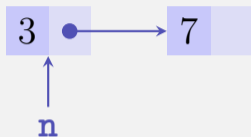
# Invarianten!



Für eine Referenz `n` auf einen Knoten in einer sortierten Liste gilt

- entweder `n = null`,
- oder `n.next = null`

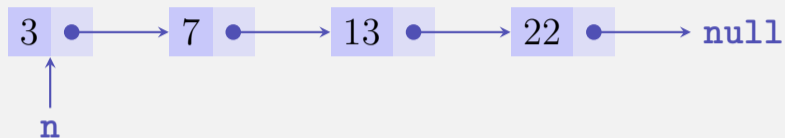
# Invarianten!



Für eine Referenz `n` auf einen Knoten in einer sortierten Liste gilt

- entweder `n = null`,
- oder `n.next = null`
- oder `n.next ≠ null` und `n.key ≤ n.next.key`.

# Invarianten!



Für eine Referenz `n` auf einen Knoten in einer sortierten Liste gilt

- entweder `n = null`,
- oder `n.next = null`
- oder `n.next ≠ null` und `n.key ≤ n.next.key`.

# Da wollen wir hin

```
public class SortedList{
    ListNode head = null;

    // insert value in a sorted way
    public void insert(int value){ ...
}

// remove value if in list, return if value was found in list
public boolean remove(int value){ ...
}

// output list values element by element
public void output(){ ...
}
}
```

# Ausgabe

```
// output list values element by element, starting from head
public void output(){
    ListNode n = head;
    while (n != null){
        Out.print(n.getKey() + " ");
        n = n.getNext();
    }
    Out.println();
}
```



# Invarianten: Einfügen von $x$

- (a) Liste ist leer oder
- (b)  $x \leq n.value$  für alle Knoten  $n$
- (c)  $x > n.value$  für alle Knoten  $n$
- (d) Es gibt einen Knoten  $n$  mit Nachfolger  $m$  so dass  $x > n.value$  und  $x \leq m.value$

Entwicklung des folgenden Codes live in der Vorlesung

# Einfügen

```
// insert value in a sorted way (sorted increasingly by value)
public void insert(int value){
    if (head == null || value <= head.getKey()){ // (a) or (b)
        head = new ListNode(value, head);
    }
    else { // (c), (d)
        ListNode n = head;
        ListNode prev = null;
        while (n != null && value > n.getKey()){
            prev = n;
            n = n.getNext();
        }
        prev.setNext(new ListNode(value, n));
    }
}
```

# Zusammenfassen

```
// insert value in a sorted way (sorted increasingly by value)
public void insert(int value){
    ListNode n = head;
    ListNode prev = null;
    while (n != null && value > n.getKey()){
        prev = n;
        n = n.getNext();
    }
    if (prev == null){
        head = new ListNode(value, n);
    } else {
        prev.setNext(new ListNode(value,n));
    }
}
```

# Invarianten: Löschen von $x$

- (a)  $x$  ist nicht enthalten
- (b)  $x$  ist das erste Element (head)
- (c)  $x$  hat einen Vorgänger

# Löschen

```
public boolean remove(int value){
    ListNode n = head;
    ListNode prev = null;
    while (n != null && value != n.getKey()) {
        prev = n; n = n.getNext();
    }
    if (n == null) { // (a)
        return false; }
    if (prev == null){ // (b)
        head = head.getNext();
    } else { // (c)
        prev.setNext(n.getNext());
    }
    return true;
}
```

# 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** (oder Fehlermeldung) 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** (oder Fehlermeldung) sonst.)
- **empty**( $Q$ ): liefert **true** wenn Queue leer, sonst **false**.