16. Dynamische Datenstrukturen

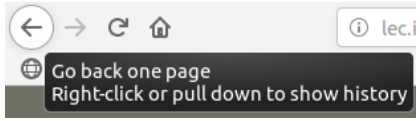
Verkettete Listen, Abstrakte Datentypen Stapel, Warteschlange

Datenstrukturen

Eine Datenstruktur *organisiert Daten* so in einem Computer, dass man sie *effizient nutzen* kann.

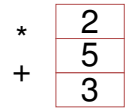
Motivation: Stapel

Beispiele zur Verwendung eines Stapels



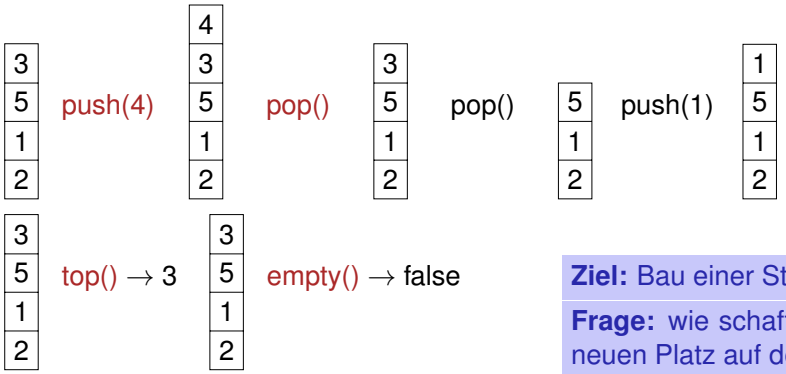
- Webseiten Besuch (Back-Button)
- Undo-Funktion im Texteditor
- Rechner (mit Suffix Notation)

```
3 5 2 * + = 3 + (5 * 2) = 13
```



- Datenstruktur geeignet für eine Einführung in einer Vorlesung wie dieser ☺

Stapel Operationen (`push, pop, top, empty`)



Ziel: Bau einer Stapel-Klasse!

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

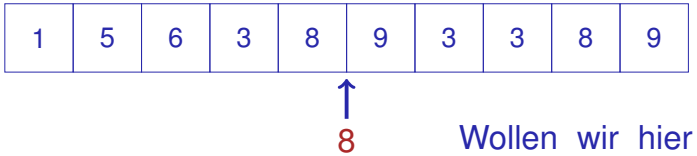
Wir brauchen einen neuen Container!

Container bisher: Array (`T[]`)

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

Arrays können wirklich nicht alles...

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



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

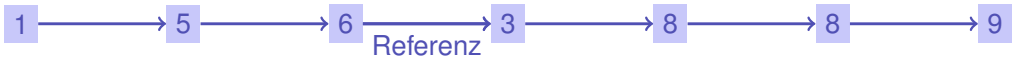
Arrays können wirklich nicht alles...

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

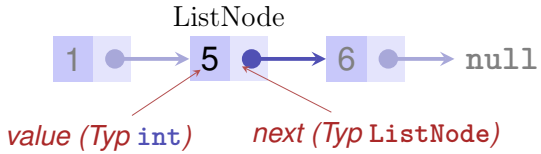
Wollen wir hier löschen, müssen wir alles rechts davon verschieben

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



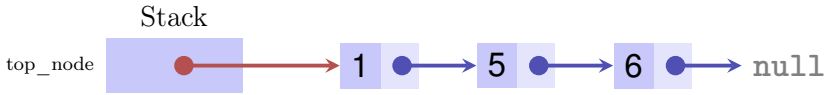

```
class ListNode {
  int value;
  ListNode next;

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


Abstrakte Datentypen

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

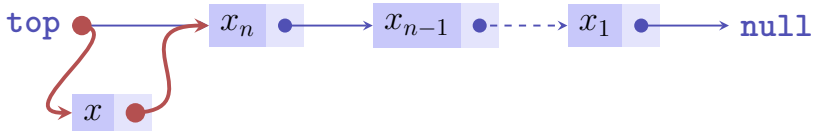
- `s.push(x)`: Legt Element `x` auf den Stapel `s`.
- `s.pop()`: Entfernt und liefert oberstes Element von `s`, oder `null` (oder Fehlermeldung).
- `s.top()`: Liefert oberstes Element von `s`, oder `null` (oder Fehlermeldung).
- `s.empty()`: Liefert `true` wenn Stack `s` leer, sonst `false`.
- `new Stack()`: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

Stapel = Referenz aufs oberste Element




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

Implementation `push`



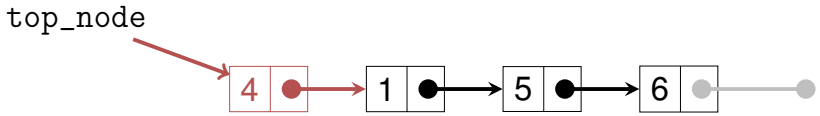
`push(x)`:

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

Implementation `push` in Java

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

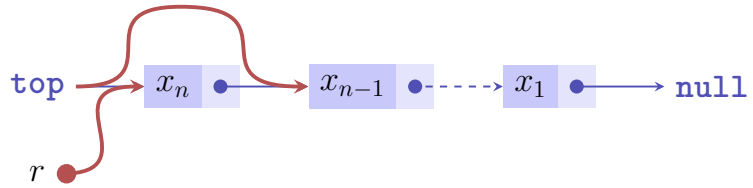
```
push(4);
```



Implementation `empty` in Java

```
public class Stack{
  private ListNode top_node;
  ...

  public boolean empty(){
    return top_node == null;
  }
}
```


Implementation `pop`

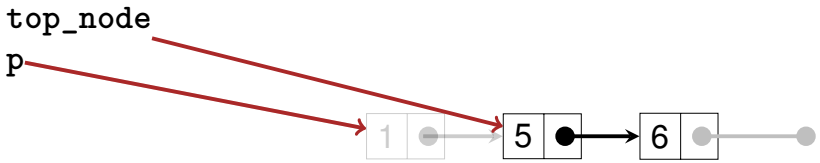


s.`pop`():

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

Implementation pop in Java

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



Weiteres Beispiel: Sortierte Verkettete Liste

Erwünschte Funktionalität:

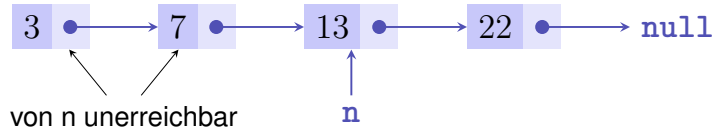
- (Sortierte) Ausgabe
- Hinzufügen eines Wertes
- (Suchen eines Wertes)
- Entfernen eines Wertes

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(){ ... }
}
```


ListNode

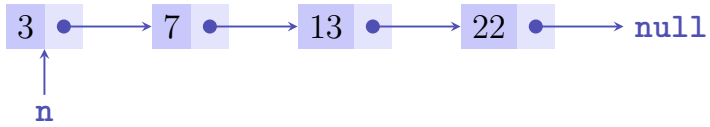
```
class ListNode{
  int value;
  ListNode next;

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



output

```
public class SortedList{
  ListNode head = null;
  ...
  // output list values element by element, starting from head
  public void output(){
    ListNode n = head;
    while (n != null){
      Out.print(n.value + " -> ");
      n = n.next;
    }
    Out.println("NIL");
  }
}
```


Invarianten!



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

- entweder `n` $=$ `null`,
- oder `n.next` $=$ `null`,
- oder `n.next` $\neq$ `null` und `n.value` $\leq$ `n.next.value`.

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.value){ // (a) or (b)
    head = new ListNode(value, head);
  }
  else { // (c), (d)
    ListNode n = head;
    ListNode prev = null;
    while (n != null && value > n.value){
      prev = n;
      n = n.next;
    }
    prev.next = 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.value){
    prev = n;
    n = n.next;
  }
  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.value) {
    prev = n; n = n.next;
  }
  if (n == null) { // (a)
    return false;
  } else if (prev == null){ // (b)
    head = head.next;
  } else { // (c)
    prev.setNext(n.next);
  }
  return true;
}
```


Queue (Schlange / Warteschlange / FIFO)

Queue ist ein ADT mit folgenden Operationen:

- `q.enqueue(x)`: fügt `x` am *Ende* der Schlange `q` an.
- `q.dequeue()`: entfernt `x` vom *Anfang* der Schlange und gibt `x` zurück (`null` (oder Fehlermeldung) sonst.)
- `q.empty()`: liefert `true` wenn Queue leer, sonst `false`.

First **I**n **F**irst **O**ut: Das, was zuerst hineinkommt, kommt zuerst wieder heraus. (Implementation: in der Übung)