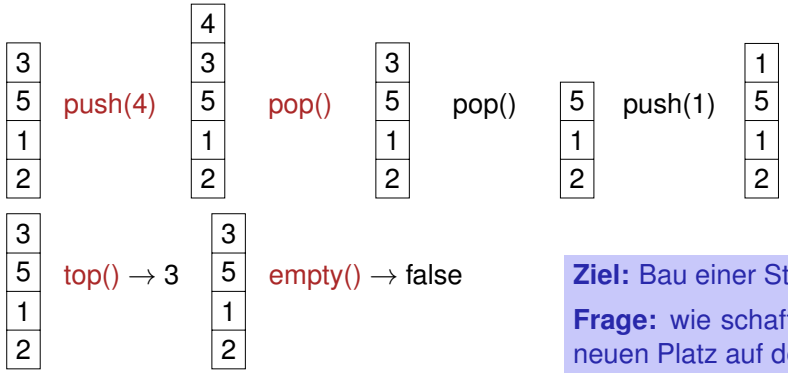
15. Dynamische Datenstrukturen

Verkettete Listen, Abstrakte Datentypen Stapel, Warteschlange

Motivation: Stapel

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



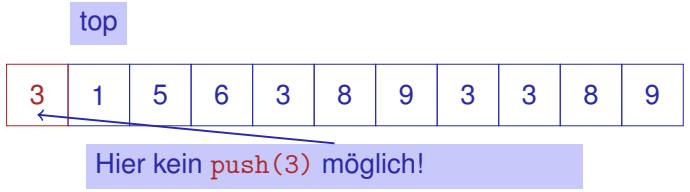
Ziel: Bau einer Stapel-Klasse!

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

Wir brauchen einen neuen Container!

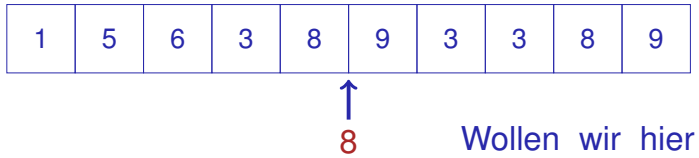
Container bisher: Array (`T[]`)

- 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...

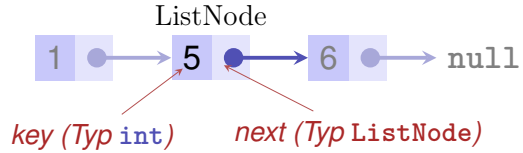
- Das Einfügen oder Löschen von Elementen „in der Mitte” ist aufwändig.

Wollen wir hier löschen, müssen wir alles rechts davon verschieben

Der neue Container: Verkettete Liste

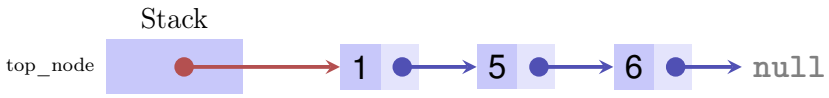
- *Kein* zusammenhängender Speicherbereich und *kein* wahlfreier Zugriff
- Jedes Element “kennt” seinen Nachfolger
- Einfügen und Löschen beliebiger Elemente ist einfach, *auch am Anfang der Liste*
- ⇒ Ein Stapel kann als Liste realisiert werden

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



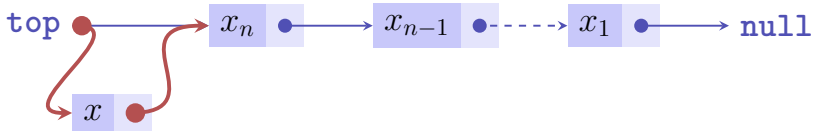

```
public class Stack {
    private ListNode top_node;

    public void push (int value) {...}
};
```


Abstrakte Datentypen

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

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

Implementation Push



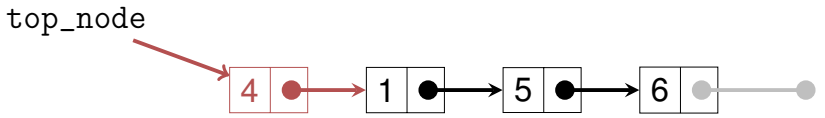
`push`(x, S):

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

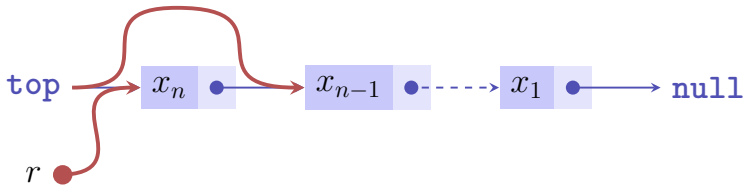
Implementation Push in Java

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

```
push(6);
```



Implementation Pop

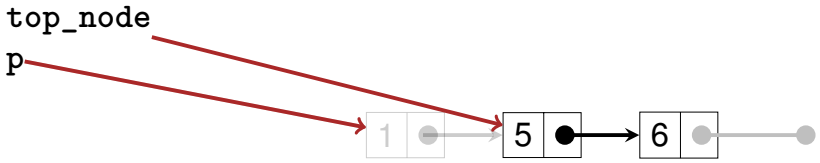


`pop`(S):

1. Ist `top`=`null`, dann gib `null` zurück. Alternativ 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.value;
}
```



Analyse

Jede der Operationen `push`, `pop`, `top` und `empty` auf dem Stack ist in $\mathcal{O}(1)$ Schritten ausführbar.

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.

16. Stepwise Refinement

Stepwise Refinement, Beispiel verkettete Liste

Stepwise Refinement

- Einfache *Programmiertechnik* zum Lösen komplexer Probleme.
- Top-down Ansatz

Stepwise Refinement

Formulierung einer groben Lösung mit Hilfe von

- Kommentaren
- Aufrufe von fiktiven Methoden

Wiederholte Verfeinerung

- Kommentare $\Rightarrow$ Programmtext
- Fiktive Methoden $\Rightarrow$ Implementation der Methoden

Stepwise Refinement

- Verfeinerung kann sich auch auf die Entwicklung der Datenrepräsentation beziehen.
- Wird die Verfeinerung so weit wie möglich durch Methoden realisiert, entstehen Teillösungen, die auch bei anderen Problemen eingesetzt werden können.
- Stepwise refinement fördert (aber ersetzt nicht!) das strukturelle Verständnis des Problems.
- Stepwise refinement ersetzt nicht das Testen des Codes.

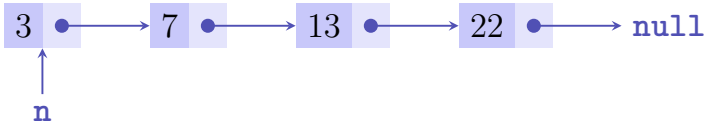
Beispiel: Sortierte verkettete Liste

Benötigte Operationen:

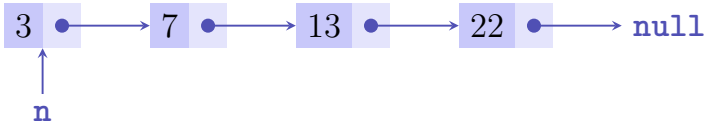
- Einfügen eines neuen Wertes
- Traversieren: Ausgeben aller Werte
- (Rückwärts Traversieren)

Überlegungen zur Datenstruktur

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

    ListNode (int v, ListNode n){
        value = v;
        next = n;
    }
}
```



Invarianten!



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

- entweder $\texttt{n} = \texttt{null}$,
- oder $\texttt{n.next} = \texttt{null}$
- oder $\texttt{n.next} \neq \texttt{null}$ und $\texttt{n.next.value} \geq \texttt{n.value}$.

SortedList

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

    public void Insert(int value){
    }
}
```

Invarianten für das Einfügen von x:

a leere Liste 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 $x < m.value$

LiveCoding

Der folgende Code wird in der Vorlesung entwickelt. Die Folien dienen nur als Referenz.

Einfügen

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

  public boolean empty(){
    return head == null;
  }

  public void Insert(int value){
    // wenn Liste leer, dann head = neues Element ohne Nachfolger
    // sonst suche Einf\"{u}geposition und f\"{u}ge ein
  }
}
```


Moment Mal...

Ich bin gerade dabei, eine neue Funktionalität einzubauen. Ich muss sofort testen.

Manuelle Testmöglichkeit vorsehen

```
import java. util .Scanner;
public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        SortedList s = new SortedList();
        String command;
        do{
            command = scanner.next();
            if (command.equals("in")){
                while (scanner.hasNextInt()){
                    int value = scanner.nextInt();
                    s. Insert(value);
                }
            }
            else if (command.equals("out")){
                s.Out();
            }
            else{
                assert(command.equals("end"));
            }
        } while (!command.equals("end"));
    }
}
```

Einfügen

```
public void Insert(int value){
// wenn Liste leer, dann head = neues Element ohne Nachfolger
    if (empty()){ // Fall (a): Leere Liste
        head = new ListNode(value, null);
    }
    else{     // sonst suche Einfuegeposition und fuege ein
        if (value <= head.value){ // Fall (b): Wert kleiner als alle anderen
            head = new ListNode(value, head);
        }
        else{ // Fall (c) oder (d)
            ListNode prev = head; // != null
            ListNode n = prev.next;
            while (n != null && value > n.value){
                prev = n;
                n = prev.next;
            }
            prev.next = new ListNode(value, n);
        }
    }
}
```


Vereinfachen

```
public void Insert(int value){
  if (empty()||value <= head.value){ // Fall (a) und (b)
    head = new ListNode(value, head);
  }
  else{ // Fall (c) oder (d)
    ListNode prev = head; // != null
    ListNode n = prev.next;
    while (n != null && value > n.value){ // suche Vorgaenger
      prev = n;
      n = prev.next;
    }
    prev.next = new ListNode(value, n);
  }
}
```


Umgekehrte Ausgabe

```
public void OutR(ListNode n){
  if (n != null){
    OutR(n.next);
    System.out.print(n.value + " ");
  }
}
```