# Stacks

>> Stacks sind ein Beispiel für einen sog. **Abstrakten Datentyp (ADT)**: eine Menge strukturierter Daten und eine Definition von Operationen, die auf diesen Daten ausgeführt werden können.

Stacks können auf eine von zwei Arten implementiert werden: als **Array** oder als **verkettete Liste**. Beide Varianten sehen „von außen“ gleich aus. Von den Details der Implementation kann man bei der Benutzung (nahezu) vollständig abstrahieren.

Prinzip: **Last in, first out (LIFO)**

>> Es gibt nur zwei Operationen, die mit einem Stack ausgeführt werden können:

**Push:** Ein neues Element wird oben auf den Stack gelegt.

**Pop:** Das zuletzt hinzugefügte Element wird vom Stack entfernt.

## Array-basierte Implementierung

```
typedef struct _stack
{
    VALUE array[CAPACITY];
    int top;
}
stack;
```

## Array-basierte Implementierung

```
stack s;
s.top = 0;
```

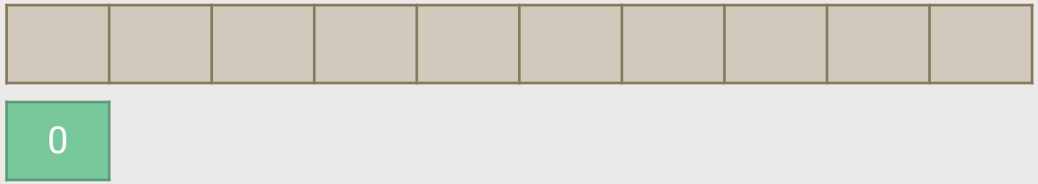
```
typedef struct _stack
{
    VALUE array[CAPACITY];
    int top;
}
stack;
```

## Array-basierte Implementierung

**push:** Neues Element oben auf Stack legen.

```
void push(stack* s, VALUE data);
```

```
typedef struct _stack
{
    VALUE array[CAPACITY];
    int top;
}
stack;
```



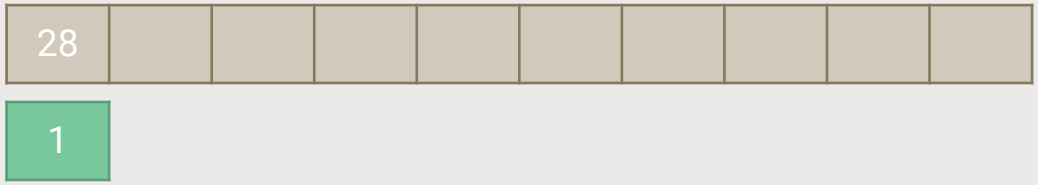

Auszuführende Schritte:

1. Einen Pointer auf den Stack akzeptieren
2. Daten vom Typ VALUE akzeptieren
3. Diese Daten an der Position top ablegen
4. Die Position von top ändern

## Array-basierte Implementierung

```
stack s;
s.top = 0;
push(&s, 28);
```

```
typedef struct _stack
{
    VALUE array[CAPACITY];
    int top;
}
stack;
```



## Array-basierte Implementierung

```
stack s;
s.top = 0;
push(&s, 28);
push(&s, 33);
```


```
typedef struct _stack
{
    VALUE array[CAPACITY];
    int top;
}
stack;
```

## Array-basierte Implementierung

```
stack s;
s.top = 0;
push(&s, 28);
push(&s, 33);
push(&s, 19);
```


```
typedef struct _stack
{
    VALUE array[CAPACITY];
    int top;
}
stack;
```

## Array-basierte Implementierung

**pop:** Oberstes Element vom Stack entfernen.

```
VALUE pop(stack* s);
```

Auszuführende Schritte:

1. Einen Pointer auf den Stack akzeptieren
2. Die Position von top ändern
3. Den Wert des entfernten Elements zurückgeben

```
typedef struct _stack
{
    VALUE array[CAPACITY];
    int top;
}
stack;
```

## Array-basierte Implementierung

```
stack s;
s.top = 0;
push(&s, 28);
push(&s, 33);
push(&s, 19);
```


```
int x = pop(&s);
```


```
typedef struct _stack
{
    VALUE array[CAPACITY];
    int top;
}
stack;
```

## Array-basierte Implementierung

```
typedef struct _stack
{
    VALUE array[CAPACITY];
    int top;
}
stack;
```

```
int y = pop(&s);
```


```
push(&s, 40);
```

**Listen-basierte Implementierung**
*Erster, einfacher Ansatz*

```
typedef struct _stack
{
    VALUE val;
    struct _stack *next;
}
stack;
```

Wichtig: Immer den Pointer auf den Kopf der Liste behalten!

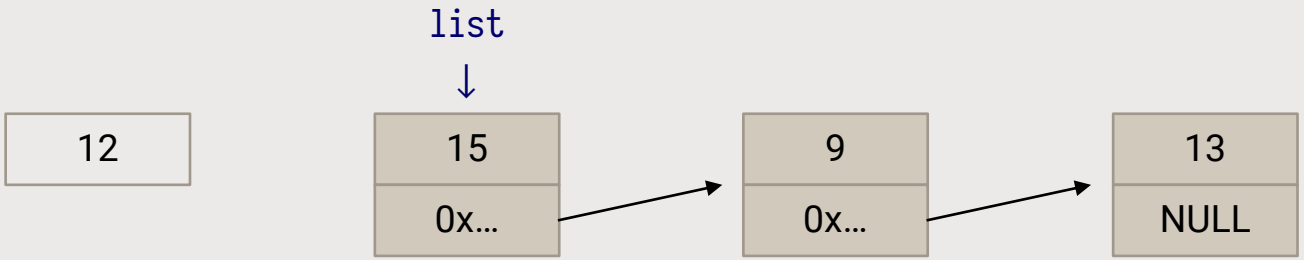
Für **push**:
1. Neuen Knoten dynamisch allozieren
2. *next*-Pointer auf aktuellen Kopf setzen
3. Kopf-Pointer auf neuen Knoten setzen

## Listen-basierte Implementierung: push

```
stack* push(stack* head, VALUE val);

list = push(list, 12);
```

```
typedef struct _stack
{
    VALUE val;
    struct _stack *next;
}
stack;
```

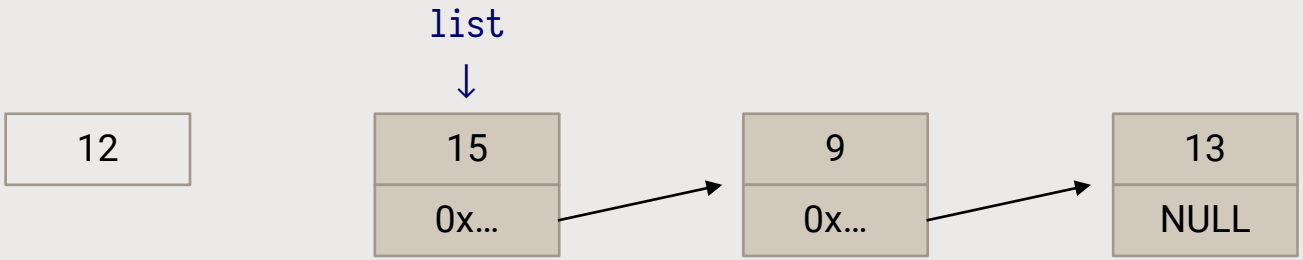


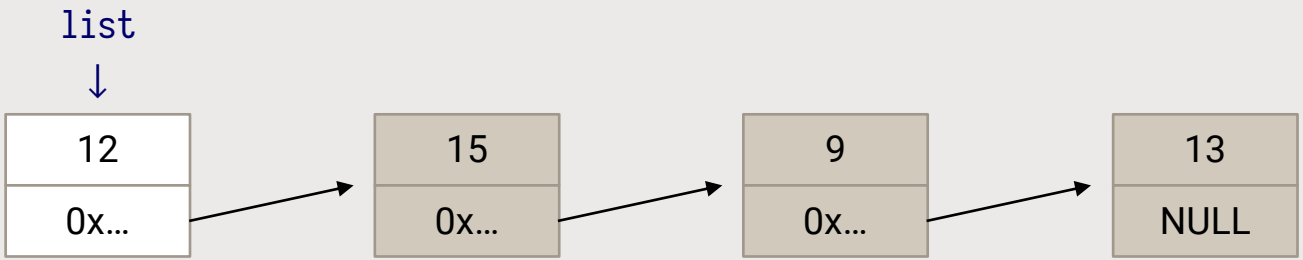
## Listen-basierte Implementierung: push

```
stack* push(stack* head, VALUE val);

list = push(list, 12);
```

```
typedef struct _stack
{
    VALUE val;
    struct _stack *next;
}
stack;
```

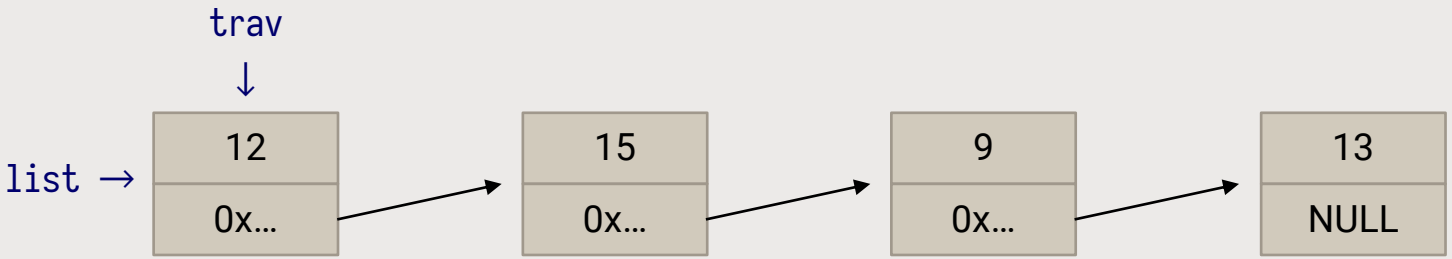


## Listen-basierte Implementierung: pop

1. Zum zweiten Element wandern (falls vorhanden)
2. Ersten Knoten freigeben
3. Kopf-Pointer auf das (ehemalige) zweite Element setzen.

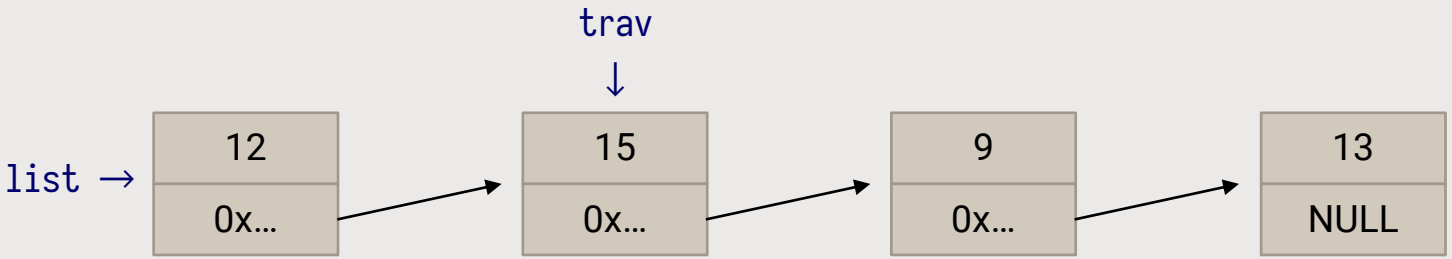
```
typedef struct _stack
{
    VALUE val;
    struct _stack *next;
}
stack;
```



## Listen-basierte Implementierung: pop

1. Zum zweiten Element wandern (falls vorhanden)
2. Ersten Knoten freigeben
3. Kopf-Pointer auf das (ehemalige) zweite Element setzen.

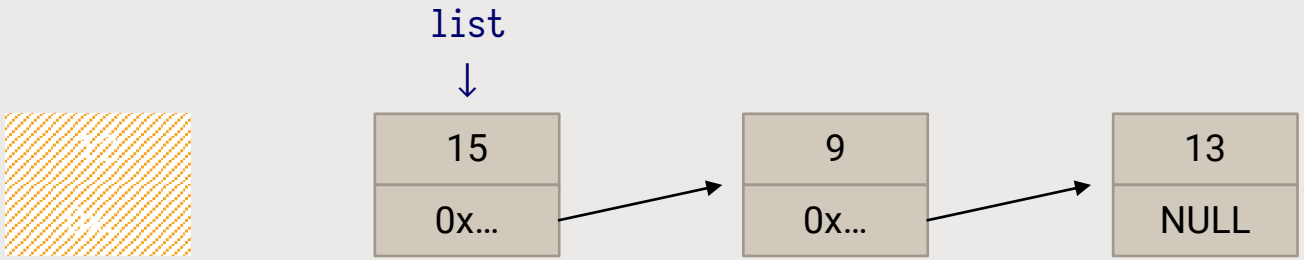
```
typedef struct _stack
{
    VALUE val;
    struct _stack *next;
}
stack;
```



## Listen-basierte Implementierung: pop

1. Zum zweiten Element wandern (falls vorhanden)
2. Ersten Knoten freigeben
3. Kopf-Pointer auf das (ehemalige) zweite Element setzen.

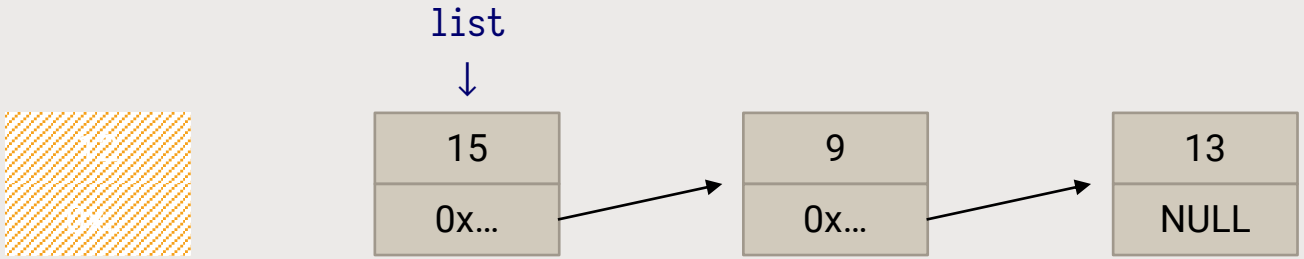
```
typedef struct _stack
{
    VALUE val;
    struct _stack *next;
}
stack;
```



## Listen-basierte Implementierung: pop

```
typedef struct _stack
{
    VALUE val;
    struct _stack *next;
}
stack;
```

1. Zum zweiten Element wandern (falls vorhanden)
2. Ersten Knoten freigeben
3. Kopf-Pointer auf das (ehemalige) zweite Element setzen.



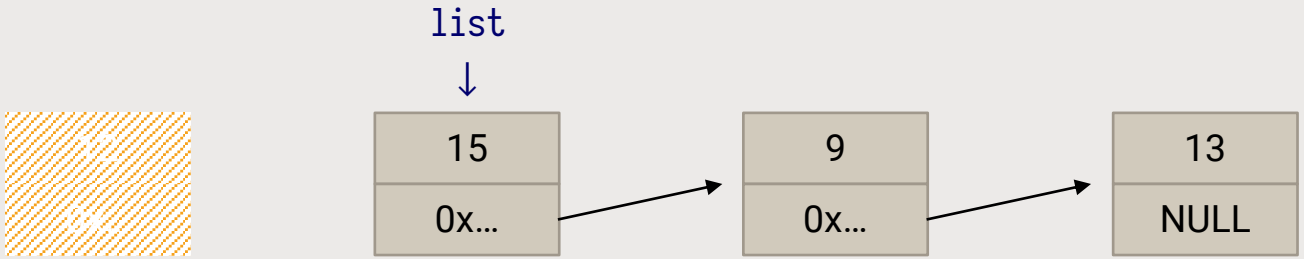

Signatur von pop?

```
stack* pop(stack* head);
```

## Listen-basierte Implementierung: pop

```
typedef struct _stack
{
    VALUE val;
    struct _stack *next;
}
stack;
```

1. Zum zweiten Element wandern (falls vorhanden)
2. Ersten Knoten freigeben
3. Kopf-Pointer auf das (ehemalige) zweite Element setzen.




Signatur von pop?

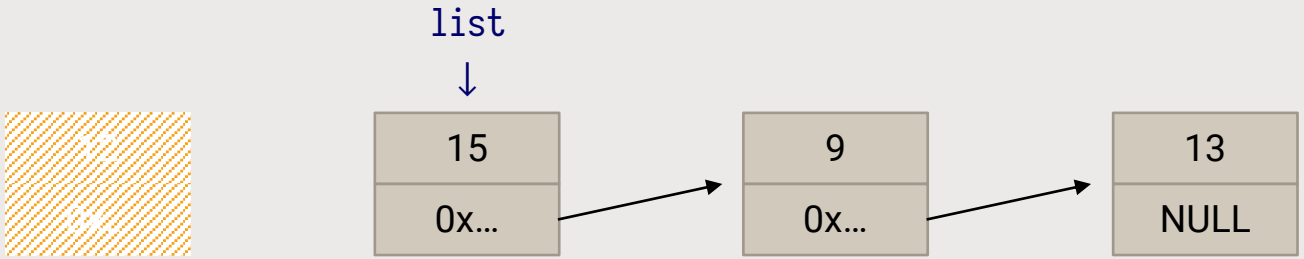
~~`stack* pop(stack* head);`~~

```
VALUE pop(stack* head);
```

## Listen-basierte Implementierung: pop

```
typedef struct _stack
{
    VALUE val;
    struct _stack *next;
}
stack;
```

1. Zum zweiten Element wandern (falls vorhanden)
2. Ersten Knoten freigeben
3. Kopf-Pointer auf das (ehemalige) zweite Element setzen.




Signatur von pop?

~~`stack* pop(stack* head);`~~

~~`VALUE pop(stack* head);`~~

```
VALUE pop(stack** head);
```

**Listen-basierte Implementierung**

*Besser gekapselte Version (Beispiel für int)*

```
typedef struct Stack {
    Node* head;
} Stack;


typedef struct Node {
    int data;
    struct Node* next;
} Node;


void push(Stack* s, int value);
int pop(Stack* s);
```

```c
void stack_init(Stack* stack)
{
    stack->head = NULL;
}

void push(Stack* stack, int value)
{
    Node* new_node = malloc(sizeof(Node));
    if (new_node == NULL)
    {
        return; // Fehlerbehandlung hier vereinfacht
    }
    new_node->data = value;
    new_node->next = stack->head;
    stack->head = new_node;
}
```

---

```c
Stack* stack = malloc(sizeof(Stack));
stack_init(stack);
push(stack, 42);
```

```c
typedef struct Stack {
    Node* head;
} Stack;

typedef struct Node {
    int data;
    struct Node* next;
} Node;
```

```c
void stack_init(Stack* stack)
{
    stack->head = NULL;
}

void push(Stack* stack, int value)
{
    Node* new_node = malloc(sizeof(Node)); // new_node: ???
    if (new_node == NULL)
    {
        return; // Fehlerbehandlung hier vereinfacht
    }
    new_node->data = value;           // new_node: 42 -> ?
    new_node->next = stack->head;     // new_node: 42 -> NULL
    stack->head = new_node;           // head -> 42 -> NULL
}
```

---

```c
Stack* stack = malloc(sizeof(Stack));
stack_init(stack);
push(stack, 42);
```

```c
typedef struct Stack {
    Node* head;
} Stack;

typedef struct Node {
    int data;
    struct Node* next;
} Node;
```

```c
int pop(Stack* stack)
{
    if (stack->head == NULL)
    {
        return -1; // Fehlerbehandlung hier vereinfacht
    }
    Node* temp = stack->head;
    int value = temp->data;
    stack->head = temp->next;
    free(temp);
    return value;
}
```

---

```c
int value = pop(stack); // value ist nun 42
```

```c
typedef struct Stack {
    Node* head;
} Stack;

typedef struct Node {
    int data;
    struct Node* next;
} Node;
```

```c
int pop(Stack* stack)
{
    if (stack->head == NULL)
    {
        return -1; // Fehlerbehandlung hier vereinfacht
    }
    Node* temp = stack->head; // temp -> (42) -> NULL
    int value = temp->data;   // value = 42
    stack->head = temp->next; // head -> NULL
    free(temp);               // temp -> 42
    return value;
}
```

---

```c
int value = pop(stack); // value ist nun 42
```

```c
typedef struct Stack {
    Node* head;
} Stack;

typedef struct Node {
    int data;
    struct Node* next;
} Node;
```

```c
int pop(Stack* stack)
{
    if (stack->head == NULL)
    {
        return -1; // Fehlerbehandlung hier vereinfacht
    }
    Node* temp = stack->head; // temp -> (42) -> NULL
    int value = temp->data;   // value = 42
    stack->head = temp->next; // head -> NULL
    free(temp);               // temp -> 42
    return value;
}
```

```c
typedef struct Stack {
    Node* head;
} Stack;

typedef struct Node {
    int data;
    struct Node* next;
} Node;
```

---

```c
int value = pop(stack); // value ist nun 42
```

*Nicht verwechseln:*

Stack im Heap:

head -> [42] -> NULL

Call-Stack von pop:

```
|--------------|
| value = 42   |
| temp = 0x... |
|--------------|
```

```c
// Hilfsfunktion: Prüfen ob Stack leer ist
bool is_empty(Stack* stack)
{
    return stack->head == NULL;
}

// Verbesserte Pop-Implementation
int pop(Stack* stack)
{
    if (is_empty(stack))
    {
        return -1; // Fehlerbehandlung hier vereinfacht
    }
    Node* temp = stack->head;
    int value = temp->data;
    stack->head = temp->next;
    free(temp);
    return value;
}
```

```c
typedef struct Stack {
    Node* head;
} Stack;

typedef struct Node {
    int data;
    struct Node* next;
} Node;
```

# EXTRAS IN 3 MINUTEN

FRAGEN – ANTWORTEN – RÄTSEL
UND KURZE ZUSAMMENFASSUNG

```c
#include <stdio.h>

int evaluate(const char* expr) {
    Stack nums, ops; // ein Stack für Zahlen, einer für Operatoren
    stack_init(&nums); stack_init(&ops);

    // expr[i] != '\0' prüft auf das String-Ende
    for (int i = 0; expr[i] != '\0'; i++) {
        char c = expr[i];
        if (c == ' ') continue; // Leerzeichen ignorieren
        if (c == '(') {
            push(&ops, c); // öffnende Klammer merken
        } else if (c >= '0' && c <= '9') {
            int num = c - '0'; // ASCII zu int konvertieren
            push(&nums, num); // Ziffer merken
        } else if (c == '+' || c == '-' || c == '*' || c == '/') {
            push(&ops, c); // Operator merken
        } else if (c == ')') {
            char op_char = (char) pop(&ops); // Operator holen
            int b = pop(&nums); // Zweite Zahl
            int a = pop(&nums); // Erste Zahl
            int result = apply_op(a, b, op_char);
            push(&nums, result); // Ergebnis merken
            pop(&ops); // '(' entfernen
        }
    }
    return pop(&nums);
}
```

```c
int apply_op(int a, int b, char op) {
    switch(op) {
        case '+': return a + b;
        case '-': return a - b;
        case '*': return a * b;
        case '/': return a / b;
    }
    return 0;
}

int main() {
    const char* expr = "((3+4)*2)";
    printf("%d\n", evaluate(expr));
    return 0;
}
```

```c
#include <stdio.h>

int evaluate(const char* expr) {
    Stack nums, ops; // ein Stack für Zahlen, einer für Operatoren
    stack_init(&nums); stack_init(&ops);

    // expr[i] != '\0' prüft auf das String-Ende
    for (int i = 0; expr[i] != '\0'; i++) {
        char c = expr[i];
        if (c == ' ') continue; // Leerzeichen ignorieren
        if (c == '(') {
            push(&ops, c); // öffnende Klammer merken
        } else if (c >= '0' && c <= '9') {
            int num = c - '0'; // ASCII zu int konvertieren
            push(&nums, num); // Ziffer merken
        } else if (c == '+' || c == '-' || c == '*' || c == '/') {
            push(&ops, c); // Operator merken
        } else if (c == ')') {
            char op_char = (char) pop(&ops); // Operator holen
            int b = pop(&nums); // Zweite Zahl
            int a = pop(&nums); // Erste Zahl
            int result = apply_op(a, b, op_char);
            push(&nums, result); // Ergebnis merken
            pop(&ops); // '(' entfernen
        }
    }
    return pop(&nums);
}
```

```c
int apply_op(int a, int b, char op) {
    switch(op) {
        case '+': return a + b;
        case '-': return a - b;
        case '*': return a * b;
        case '/': return a / b;
    }
    return 0;
}

int main() {
    const char* expr = "((3+4)*2)";
    printf("%d\n", evaluate(expr));
    return 0;
}
```

## Ablauf der Auswertung und Inhalt der Stacks

Eingabe: ((3+4)*2)

```
1. Erste Klammer:   nums: []        ops: [(]
2. Zweite Klammer:  nums: []        ops: [(, (]
3. Zahl 3:          nums: [3]       ops: [(, (]
4. Plus:            nums: [3]       ops: [(, (, +]
5. Zahl 4:          nums: [3, 4]    ops: [(, (, +]
6. Erste ')':       nums: [7]       ops: [(        // 3+4 wurde berechnet
7. Mal:             nums: [7]       ops: [(, *]
8. Zahl 2:          nums: [7, 2]    ops: [(, *]
9. Letzte ')':      nums: [14]      ops: []        // 7*2 wurde berechnet
```

Stacks sind LIFO-Datenstrukturen: Last In, First Out

Gute Kapselung der Implementationsdetails erleichtert die Benutzung

Zwei Grundoperationen: push (Hinzufügen) und pop (Entfernen)

Inkonsistente Funktionssignaturen von push und pop (Pointer auf Pointer) konnten durch Kapselung vermieden werden

Implementierung als Array oder verkettete Liste

Häufige Anwendungen: Klammerausdrücke, Funktionsaufrufe, Undo-Operationen