# Singly / doubly Linked List

EINFACHE-DOPPELTE VERKETTETE LISTE

# History



- Herbert A. Simon (1916-2001)
- Nobelpreisträger und Gewinner des Turing Awards.
- Erfinder der verketteten Liste

  (im Rahmen der IPL Sprache)

# Themen



- Verkettete Liste
- Liste – Stack
- Liste – Queue
- Bauteile einer Liste
  - Knoten und Liste
  - Knoten und Liste erstellen
- Operationen
  - Elemente Hinzufügen(Rechts)
  - Elemente nach Index hinzufügen
  - Liste löschen
  - Letztes Element löschen
  - Der Vorgänger in singly linked list finden
  - Suchen

# Verkettete Liste



- Verkettete Listen sind dynamische Datenstrukturen
- Vorteile:
  - ✓ Es wird nur der benötigte Speicherplatz reserviert.
  - ✓ Man kann den Speicher wieder freigeben.

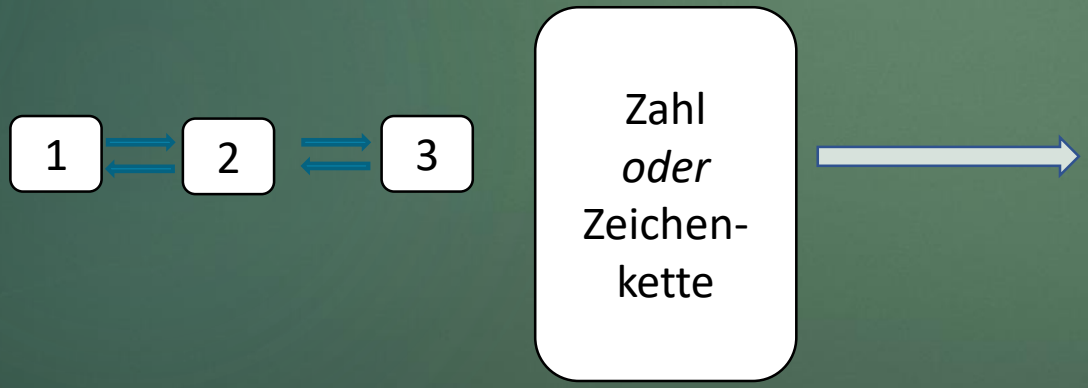
# Verkettete Liste



- Angenommen: Wir möchten 1,2,3 abspeichern


im normalen Fall wird dafür ein Array angelegt.

- Nun möchten wir noch ein Element weiter hinzufügen.

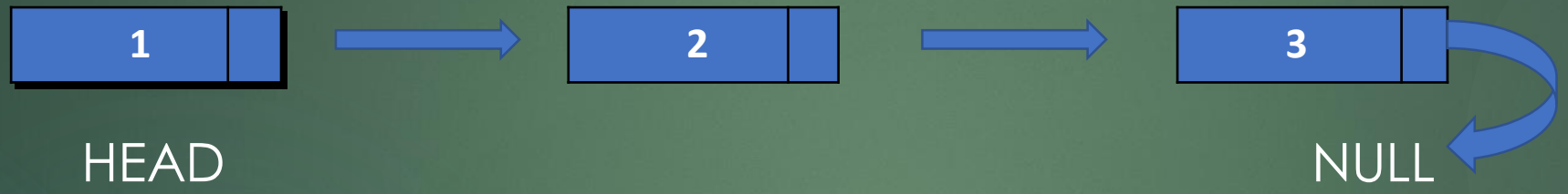



LISTEN sind dafür da

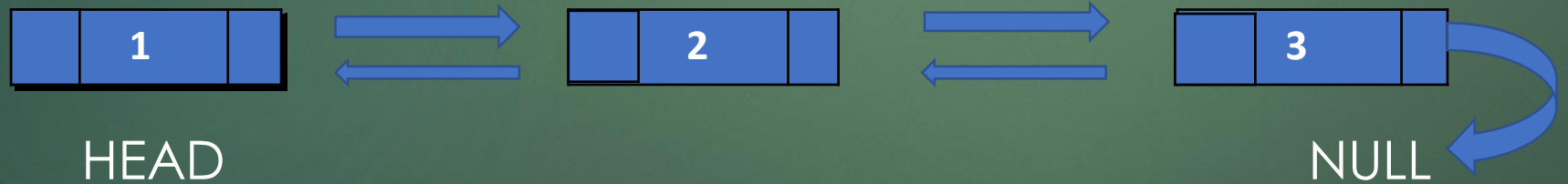
# Verkettete Listen



- Zwei Arten von Verkettete Listen
  - Einfache verkettete Listen (singly linked list)




  - Doppelte verkettete Listen (doubly linked list)



# Verkettete Listen
## Einfach verkettete Liste



- **Vorteile**
  - Elemente können sehr schnell am Anfang der Liste eingefügt werden.
- **Nachteile**
  - Es ist aufwändig nach Daten zu suchen, Knoten einzufügen und zu löschen.

# Verkettete Listen

## Doppelt verkettete Liste



- **Vorteile**
  - Schnelles Einfügen und Löschen von Elementen.
  - Über die Liste kann von hinten nach vorne iteriert werden.

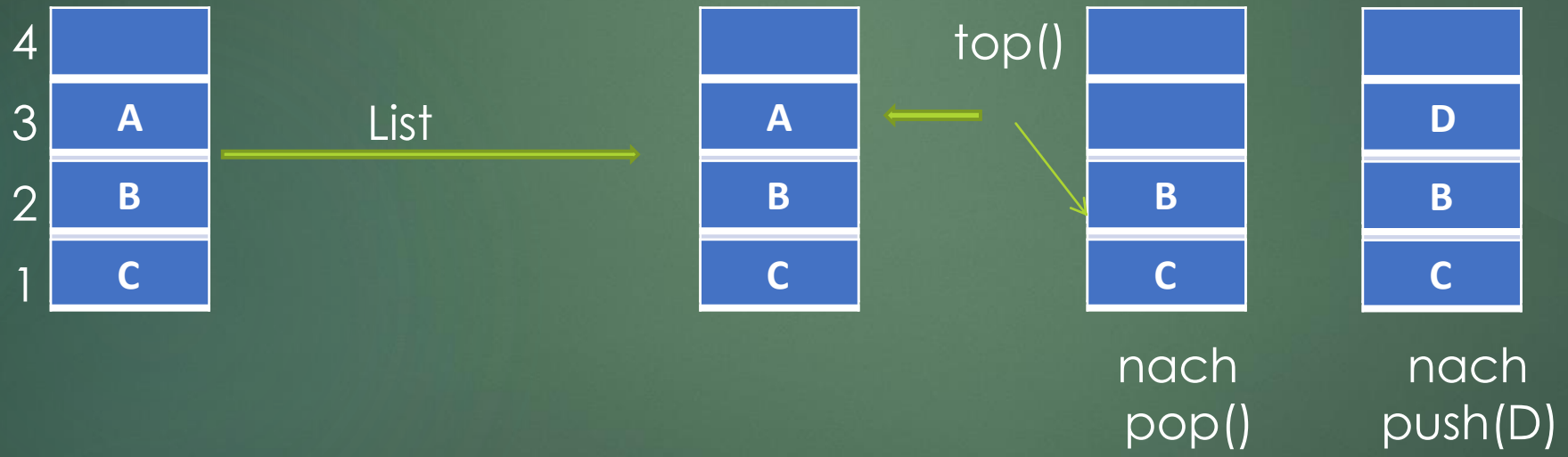
- **Nachteile**
  - Höherer Speicherbedarf .
  - Implementierung ist manchmal aufwendig.

# Liste - Stack



- Ein Stack ist ein LIFO-Speicher. Die Operationen eines Stacks sind top() , pop() und push(Element).

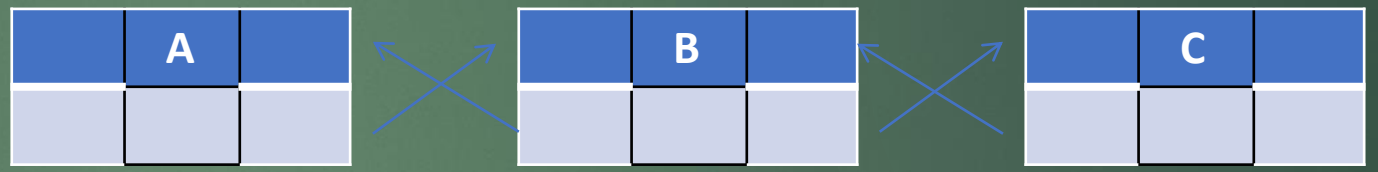
Ein Stack sieht so aus:



# Listen – Queue



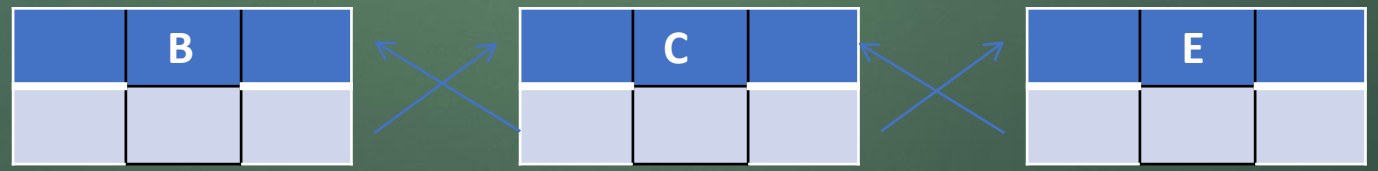
- Ein Queue ist ein FIFO-Speicher. Die Operationen eines Queues sind top() , pop(), push(Element).
- Ein Queue sieht so aus:




Top

Nach pop =>


Nach Push(E) =>



# Bauteile einer Liste

# Knoten und Liste



- Knoten (Einfache verkettete Listen) besteht aus:
  - ✓ Zeiger, der auf das nächste Element verweist.
  - ✓ Zeiger (nicht unbedingt), der Daten trägt.
- Knoten (Doppelte verkettete Listen) besteht aus:
  - ✓ Zeiger, der auf das nächste Element verweist.
  - ✓ Zeiger, der auf das Letzte Element verweist.
  - ✓ Zeiger (nicht unbedingt), der Daten trägt.
- Liste besteht aus:
  - ✓ Head.
  - ✓ Tail.

# Knoten und Liste



```c
typedef struct single_node
{
    struct single_node* next;
    void* val;
} single_node_t;
```

```c
typedef struct double_node
{
    struct double_node* next;
    struct double_node* prev;
    void* val;
} double_node_t;
```

```c
typedef struct single_list
{
    struct single_node* head;
    struct single_node* tail;
    unsigned int len;
} single_list_t;
```

```c
typedef struct double_list
{
    struct double_node* head;
    struct double_node* tail;
    unsigned int len;
} double_list_t;
```

# Liste und Knoten erstellen



- Mit Hilfe der *malloc()* Funktion entwirft man die Liste und den Knoten.

```
single_list_t* create_list_S()
{
    single_list_t* new_list = (single_list_t*) malloc(sizeof(single_list_t));
    //checking if list contains a elements(node)
    if(!new_list)
        return NULL;

    new_list->head = NULL;
    new_list->tail = NULL;
    new_list->len = 0;

    return new_list;
}
```

# Liste und Knoten erstellen



```c
double_node_t* create_node_D(void* val)
{
    double_node_t* new_node = (double_node_t*) malloc(sizeof(double_node_t));
    if(!new_node)
        return NULL;

    new_node->next = NULL;
    new_node->prev = NULL;
    new_node->val = val;

    return new_node;
}
```

# Operationen

# Elemente Hinzufügen(Rechts)



//A zu der Liste hinzufügen


HEAD HEAD = TAIL = A

//Weiter die Liste Elemente Hinzufügen


HEAD HEAD = A TAIL = HEAD -> next = B

# Elemente Hinzufügen(Rechts)



// A zu der Liste hinzufügen


HEAD HEAD = TAIL

//Weiter die Liste Elemente Hinzufügen


HEAD = A TAIL = HEAD->next

b->brev = list-> TAIL

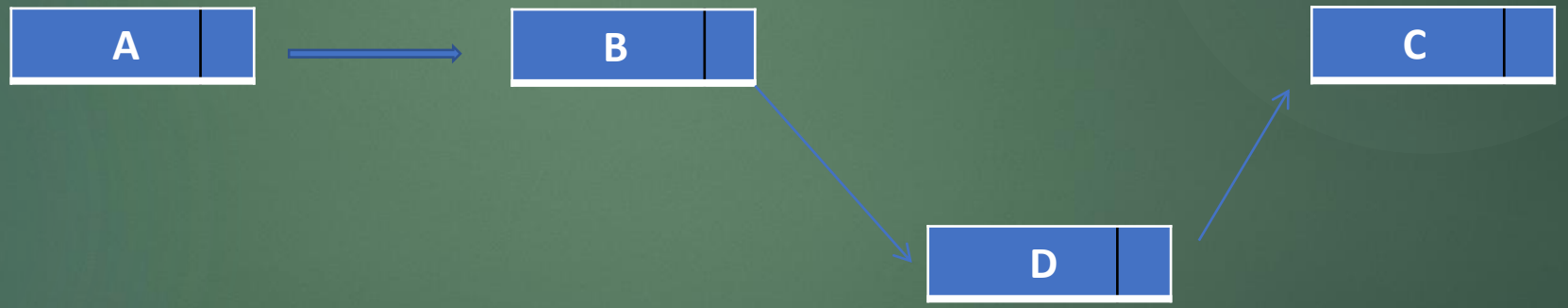
# Elemente nach Index hinzufügen



//wir haben

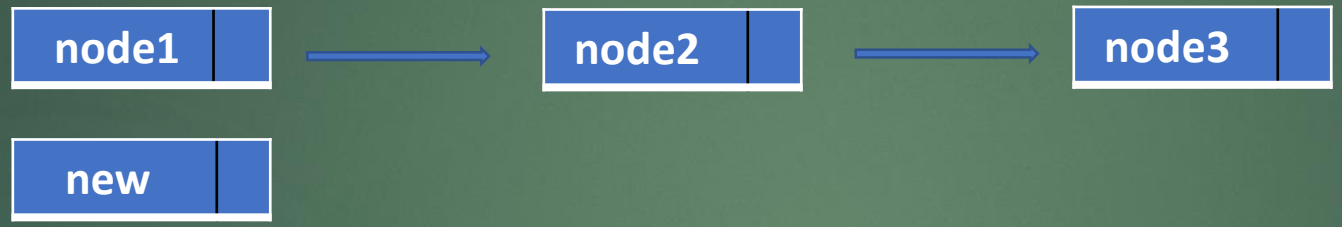

// wir wollen 

in der dritte Stelle zu der Liste einfügen:



# Elemente nach Index hinzufügen






// es war


Node2 ist der Vorgänger von node3

// nun


new ist der Vorgänger von node3

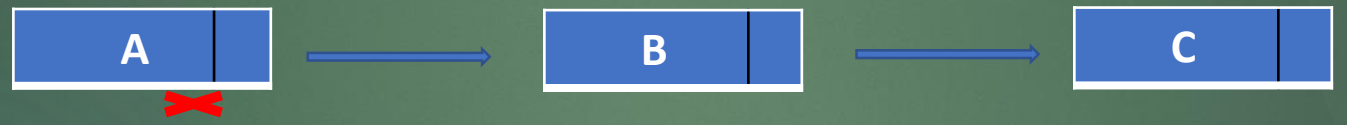
# Liste löschen



//wir haben


beim Löschen entfernt man z.B. den Kopf der Liste:

->




HEAD

->


HEAD

->


HEAD

# Letztes Element löschen



//wir haben
/


→


TAIL

→


TAIL

→


TAIL = HEAD

# Vorgänger in singly linked list finden



- Man kann den Vorgänger eines Knotens durch eine kleine Implementierung finden.

```
single_node_t* find_prev_node_S(single_list_t* list, single_node_t* node)
{
    if(!list || !node)
        return NULL;

    single_node_t* curr = list->head;
    while(curr->next != node)
    {
        curr = curr->next;
    }
    //ConsoleS(list);
    return curr;
```

# Vorgänger in singly linked list finden



//wir haben


Wir suchen den Vorgänger von C

//Hilfsknoten


1. //A.next != C


2. //B.next == C => ziel erreicht

   //gesuchte Knoten ist

# Suchen



- Man kann über z.B. den Wert nach einem Knoten suchen

```
double_node_t* finde_node_D(double_list_t* list, void* val)
{
    if(!list)
        return NULL;

    double_node_t* node = list->head;

    while(node != NULL)
    {
        if (node->val == val)
        {
            return node;
        }

        node = node->next;
    }

    return NULL;
}
```

```
single_node_t* finde_node_S(single_list_t* list, void* val)
{
    if(!list)
        return NULL;

    single_node_t* node = list->head;

    while(node != NULL)
    {
        if (node->val == val)
        {
            return node;
        }

        node = node->next;
    }
    //ConsoleS(list);
    return NULL;
}
```

# Suchen



// wir haben 


// wert1 = A, wert2 = B, wert3 = C

Wir suchen nach dem Knoten, der C trägt

// Hilfsknoten 

enthält in sich die Wert A;

1. //A != C 


2. // B != C 


3. // C == C 

gesuchter Knoten