Programmiertechnik II

Ralf Herbrich

Datentypen

Überblick

1. Abstrakte Datentypen
2. Stapel (*stacks*)
   - Auswertung Arithmetischer Ausdrücke
3. Warteschlangen (*queue*)
4. Listen (*lists*)

Überblick

1. **Abstrakte Datentypen**
2. Stapel (*stacks*)
   - Auswertung Arithmetischer Ausdrücke
3. Warteschlangen (*queue*)
4. Listen (*lists*)

Abstrakte Datentypen (ADT)

- **Motivation**: Wiederverwendbarkeit und Strukturierung von Software
- **Ziel**: Beschreibung von Datenstrukturen **unabhängig** von ihrer späteren Implementierung in einer konkreten Programmiersprache
- **Konkrete Datentypen**: Konstruiert aus Basisdatentypen/C++ Klassen
- **Abstrakte Datentypen**:
  - Spezifikation der Schnittstelle nach außen
  - Operationen und ihre Funktionalität
  - **Kapselung**: Darf nur über Schnittstelle benutzt werden
  - **Geheimnisprinzip**: Interne Realisierung ist verborgen
  - Eine Grundlage des Prinzips der objektorientierten Programmierung

ADT Beispiel: Ganzzahlen

- **Algebraische Beschreibung**: Beschreibung von **Operationen** (*operators*) und der Zusammenhänge zwischen den Operationen (*axioms*)
- **Beispiel**: (Positive) Ganzzahlen

type Nat

operators

$0: \_ \rightarrow \text{Nat}$ ← Konstante ohne Parameter

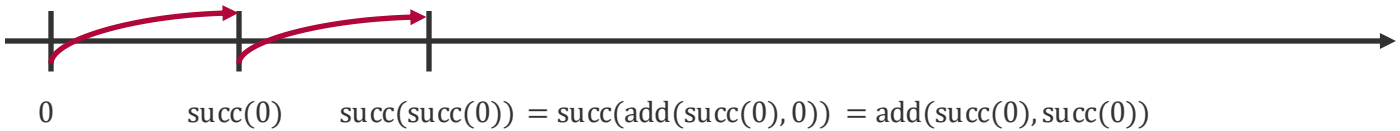
$\text{succ}: \text{Nat} \rightarrow \text{Nat}$ ← Konstruktor

$\text{add}: \text{Nat} \times \text{Nat} \rightarrow \text{Nat}$

axioms

$\forall i: \quad \text{add}(i, 0) = i$

$\forall i, j: \text{Nat} \quad \text{add}(i, \text{succ}(j)) = \text{succ}(\text{add}(i, j))$



Abstrakte Datentypen Beispiel: Bool

type Bool

operators

$$\text{true}: \_ \to \text{Bool}$$
$$\text{false}: \_ \to \text{Bool}$$
$$\wedge: \text{Bool} \times \text{Bool} \to \text{Bool}$$
$$\vee: \text{Bool} \times \text{Bool} \to \text{Bool}$$
$$\neg: \text{Bool} \to \text{Bool}$$

axioms

$$\forall x \in \text{Bool}: \quad \wedge(\text{false}, x) = \text{false}$$
$$\forall x \in \text{Bool}: \quad \wedge(x, \text{false}) = \text{false}$$
$$\wedge(\text{true}, \text{true}) = \text{true}$$
$$\forall x \in \text{Bool}: \quad \vee(\text{true}, x) = \text{true}$$
$$\forall x \in \text{Bool}: \quad \vee(x, \text{true}) = \text{true}$$
$$\vee(\text{false} \vee \text{false}) = \text{false}$$
$$\neg(\text{false}) = \text{true}$$
$$\neg(\text{true}) = \text{false}$$

Umsetzung von ADTs

- ADT in **Programmiersprachen**
  - Konzept der **Kapselung**: Verbergen der internen Repräsentation
  - Gleiche Verwendung trotz unterschiedlicher Implementierung
  - **Vorteile**:
    - Stabilität gegenüber Änderungen
    - Auswahl einer geeigneten Implementierungsvariante

- **ADT in C++**
  - Typen → Klassen
  - Null-wertige Operatoren → Konstanten
  - Mehr-wertige Operatoren → Methoden

Überblick

1. Abstrakte Datentypen
2. **Stapel (*stacks*)**
   - Auswertung Arithmetischer Ausdrücke
3. Warteschlangen (*queue*)
4. Listen (*lists*)

Stack as Abstrakter Datentyp

- **Prinzip**: *Last-In-First-Out* Speicher

type Stack(T)

operators

$\text{empty}: \_ \rightarrow \text{Stack}$

$\text{is\_empty}: \text{Stack} \rightarrow \text{Bool}$

$\text{push}: \text{Stack} \times \text{T} \rightarrow \text{Stack}$

$\text{pop}: \text{Stack} \rightarrow \text{Stack}$

$\text{top}: \text{Stack} \rightarrow \text{T}$

axioms

$\forall s \in \text{Stack}, x \in \text{T}: \quad \text{pop}\big(\text{push}(s, x)\big) = s$

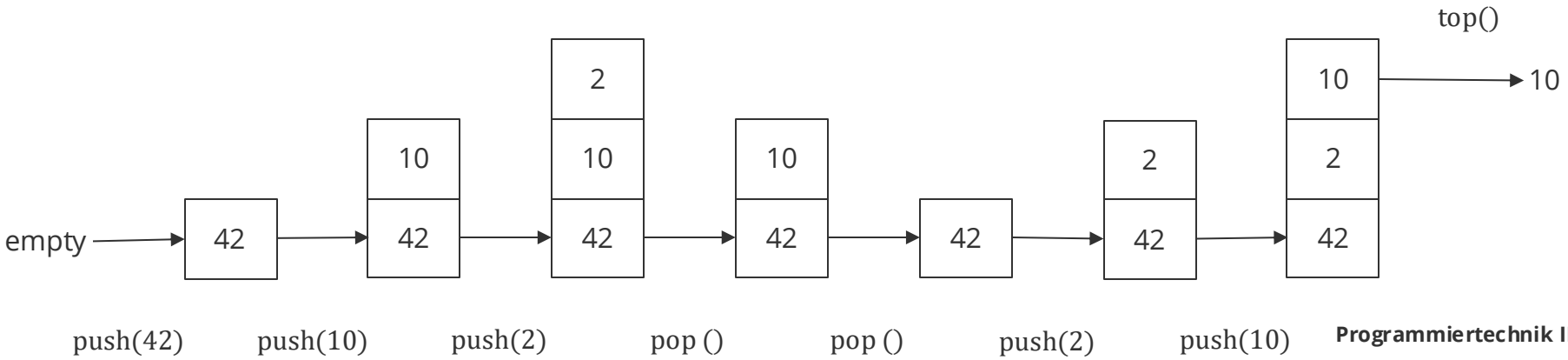
$\forall s \in \text{Stack}, x \in \text{T}: \quad \text{top}\big(\text{push}(s, x)\big) = x$

$\forall s \in \text{Stack}, x \in \text{T}: \quad \text{is\_empty}\big(\text{push}(s, x)\big) = \text{false}$

$\text{is\_empty}(\text{empty}) = \text{true}$

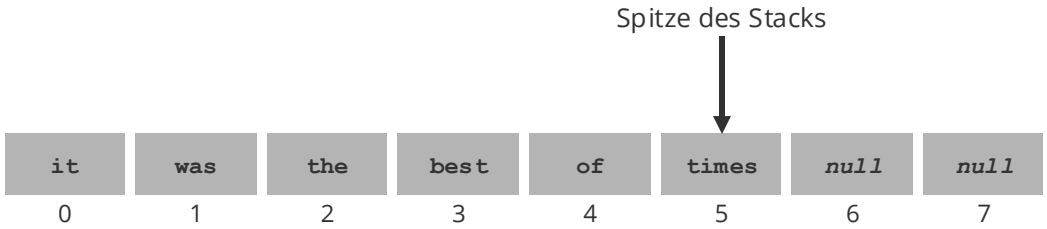
Stack in Bildern

- **Beispiel**: Stack(Int)

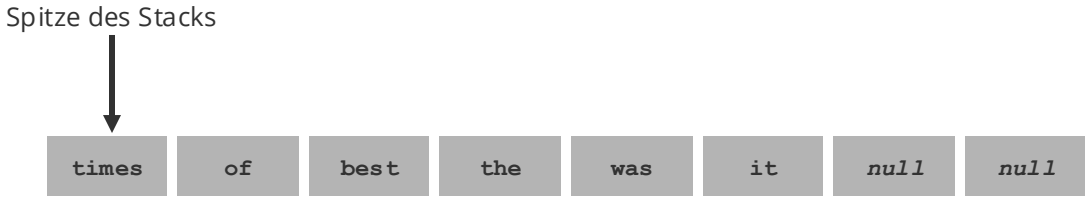


Quiz: Implementierung eines Stacks

- **A**: Man kann einen Stack nicht effizient mit einem Feld implementieren!
- **B**:



- **C**:



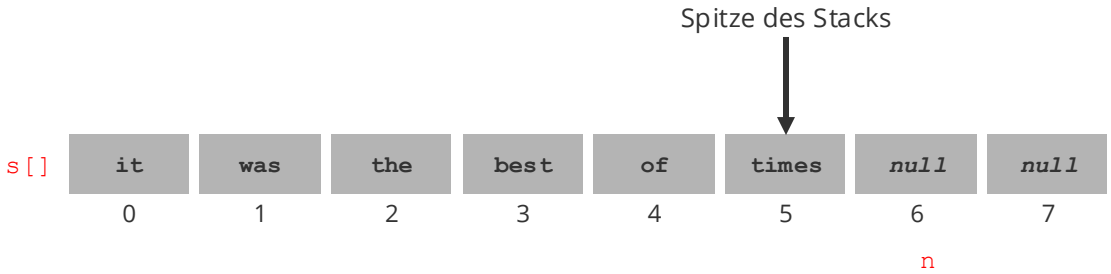
A

B

C

Implementierung eines Stacks mit Feldern

- **Feste Größe**: Benutze ein Feld `s` um die Elemente im Stack zu speichern.



- **Variable Größe**: Wie soll das Feld wachsen und schrumpfen?
  - **1. Versuch**: Nach jedem `push` **n** um eins erhöhen

$$\#\text{ops} = n + \big(2 \cdot 1 + 2 \cdot 2 + 2 \cdot 3 + \cdots 2 \cdot (n-1)\big) = O(n^2)$$

  - **2. Versuch**: Wenn das Feld voll ist, verdoppele die Größe

$$\#\text{ops} = n + (2 + 4 + 8 + \cdots n) = O(n)$$

```cpp
// Implements a stack
template <typename T>
class Stack {
    T* s;
    int n;
    int capacity;
public:
    // constructor
    Stack(int cap = 1) : n(0), capacity(cap) {
        s = new T[capacity];
    }
    // destructor
    ~Stack() { delete[] s; }
    // push method
    void push(const T& x) { s[n++] = x; }
    // pop method
    T& pop() { return (s[--n]); }
    // top method
    const T& top() const { return (s[n - 1]); }
};
```

stack.h

Überblick

1. Abstrakte Datentypen
2. Stapel (*stacks*)
   - **Auswertung Arithmetischer Ausdrücke**
3. Warteschlangen (*queue*)
4. Listen (*lists*)

Auswertung Arithmetischer Ausdrücke

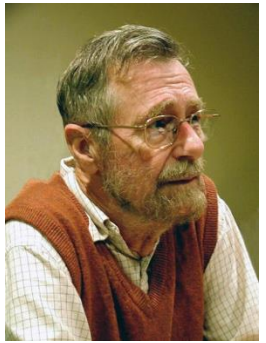
- **Arithmetischer Ausdruck**: Sprache definiert durch folgende EBNF

```
expr ::= <number> | ( expr + expr ) | ( expr - expr )
                  | ( expr * expr ) | ( expr / expr )
```

  - **Beispiele**: `( 1 + 1 )` oder `( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) )`

- **Auswertung**: Wert des Ausdrucks nach Anwendung von Arithmetik
  - **Beispiel**: `eval("( ( 1 + ( 2 * 3 ) ) * ( 2 + 1 ) )")`
    - Hat den Rückgabewert 21

- **Dijkstras Algorithmus**: Auswertung solcher Ausdrücke durch Nutzung zweier Stacks (einen für Operatoren und einen für Teilergebnisse)



Edsger Dijkstra
(1930 – 2002)

Dijkstra's Algorithmus

- **Eingabe**: Liste von Zeichen des artihmetischen Ausdrucks (`expr_list`)
- **Ausgabe**: Werte des artihmetischen Ausdrucks
- **Algorithmus**:

$S_{\text{ops}} = \text{empty}, S_{\text{vals}} = \text{empty}$

for s **in** `expr_list` **do**

if $s \in \{+, -, *, / \}$ **then** $\text{push}(S_{\text{ops}}, s)$

if $s \in \mathbb{R}$ **then** $\text{push}(S_{\text{vals}}, s)$

if $s =)$ **then**

$o = \text{pop}(S_{\text{ops}})$

$x_1 = \text{pop}(S_{\text{vals}}), \ldots, x_n = \text{pop}(S_{\text{vals}})$

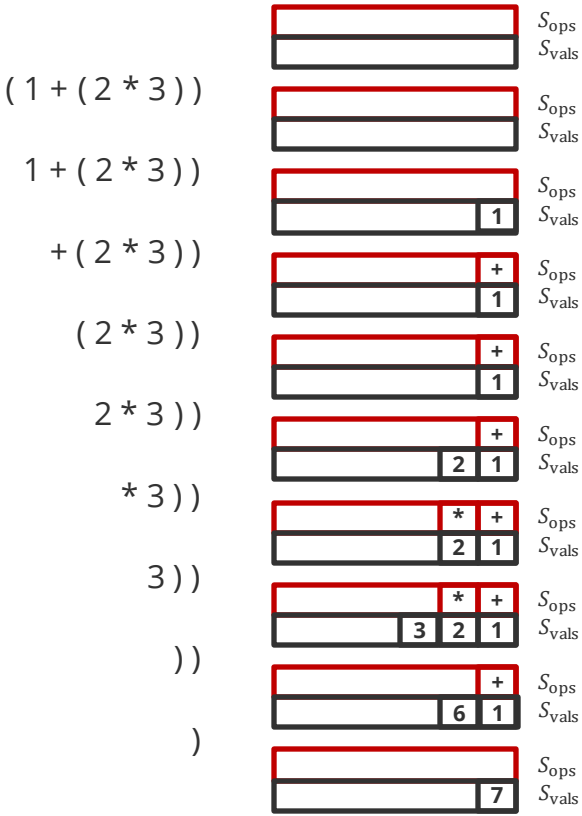
Berechne $y = o(x_1, \ldots, x_n)$ und $\text{push}(S_{\text{vals}}, y)$

end

end

return $\text{pop}(S_{\text{vals}})$

Dijkstra's Algorithmus (Beispiel)




```cpp
// main entry point of the program
int main(void) {
    Stack<int> vals;
    Stack<string> ops;
    string token;

    while (getline(cin, token, ' ')) {
        if (token[0] == '(') {
        } else if (token[0] == '+' || token[0] == '-' || token[0] == '*' || token[0] == '/')
            ops.push(token);
        else if (token[0] == ')') {
            string op = ops.pop();
            double v = vals.pop();
            if (op == "+")
                v = vals.pop() + v;
            else if (op == "-")
                v = vals.pop() - v;
            else if (op == "*")
                v = vals.pop() * v;
            else if (op == "/")
                v = vals.pop() / v;
            vals.push(v);
        } else
            vals.push(stof(token));
    }
    cout << vals.pop() << endl;
```

eval_exp.cpp

Korrektheit

- **Frage:** Warum ist der Algorithmus **korrekt**?
- **Antwort**:
  1. Jedes Mal wenn ein Ausdruck von Klammern umrundet wird, werden die beiden Operatoren von *value* Stack genommen, die Operation vom *ops* Stack und das Ergebnis zurück auf den *value* Stack gelegt
  2. Der geklammerte Ausdruck wird also durch seinen Wert ersetzt!
- **Beobachtung**:
  1. Der Algorithmus liefert den gleichen Wert, wenn die Operation **nach** den Operatoren kommt!

     ```
     ( 1 ( ( 2 3 + ) ( 4 5 * ) * ) + )
     ```

  2. Alle Klammern sind redundant!

     ```
     1 2 3 + 4 5 * * +
     ```

- **Schlussfolgerung**: Der Algorithmus funktioniert für umgekehrte polnische Notation von arithmetischen Ausdrücken (z.B. Stack Maschine, Postscript, JVM)

Jan Łukasiewicz
(1878 – 1956)

Überblick

1. Abstrakte Datentypen
2. Stapel (*stacks*)
   - Auswertung Arithmetischer Ausdrücke
3. **Warteschlangen (*queue*)**
4. Listen (*lists*)

Queue as Abstrakter Datentyp

- **Prinzip**: *First-In-First-Out* Speicher

type Queue(T)

operators

$\text{empty}: \_ \rightarrow \text{Queue}$

$\text{is\_empty}: \text{Queue} \rightarrow \text{Bool}$

$\text{enqueue}: \text{Queue} \times \text{T} \rightarrow \text{Queue}$

$\text{dequeue}: \text{Queue} \rightarrow \text{Queue}$

$\text{front}: \text{Queue} \rightarrow \text{T}$

axioms $\forall q \in \text{Queue}, x, y \in \text{T}$

$$\text{dequeue}(\text{enqueue}(\text{empty}, x)) = \text{empty}$$

$$\text{dequeue}(\text{enqueue}(\text{enqueue}(q, x), y)) = \text{enqueue}(\text{dequeue}(\text{enqueue}(q, x)), y)$$

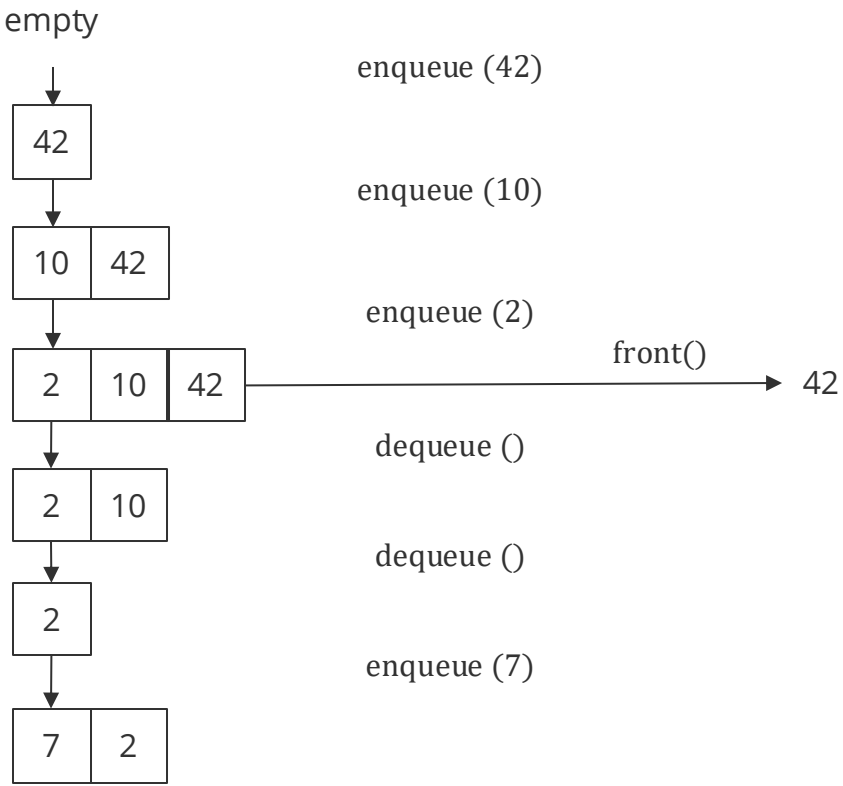
$$\text{front}(\text{enqueue}(\text{empty}, x)) = x$$

$$\text{front}(\text{enqueue}(\text{enqueue}(q, x), y)) = \text{front}(\text{enqueue}(q, x))$$

$$\text{is\_empty}(\text{enqueue}(q, x)) = \text{false}$$

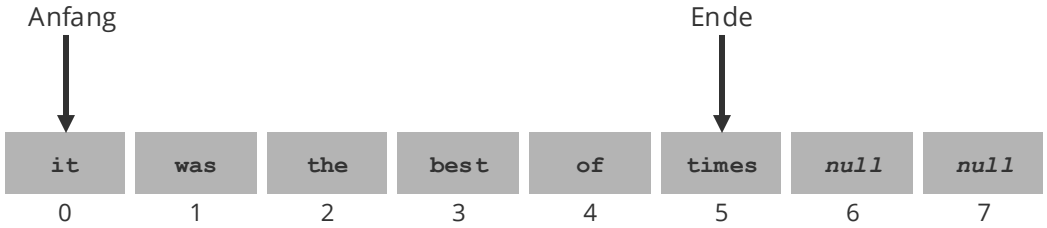
$$\text{is\_empty}(\text{empty}) = \text{true}$$

Queue in Bildern

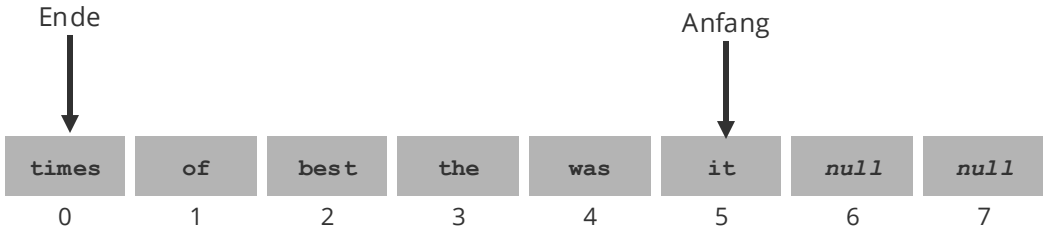


Quiz: Implementierung einer Warteschlange

- **A**: Man kann eine Warteschlange nicht effizient mit einem Feld implementieren!
- **B**:



- **C**:



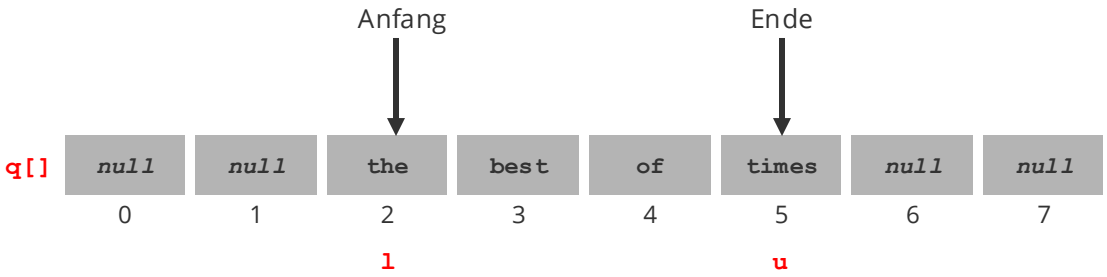
A

B

C

Implementierung einer Warteschlange mit Feldern

- **Feste Größe**: Benutze ein Feld `q` um die Elemente in der Warteschlange zu speichern.



queue.h

```
// Implements a queue with a fixed size array of values
template <typename T>
class Queue {
    T* q;
    int l, u;
    int capacity;

public:
    // constructor
    Queue(int cap = 100) : l(0), u(0), capacity(cap) {
        q = new T[capacity];
    }
    // destructor
    ~Queue() { delete[] queue_data; }
    // enqueue method
    void enqueue(const T& x) {
        q[u] = x;
        u = (u + 1) % capacity;
    }

    // dequeue method
    T& dequeue() {
        auto old_l = l;
        l = (l + 1) % capacity;
        return (q[old_l]);
    }
    // front method
    const T& front() const { return (queue_data[l]); }
};
```

Überblick

1. Abstrakte Datentypen
2. Stapel (*stacks*)
   - Auswertung Arithmetischer Ausdrücke
3. Warteschlangen (*queue*)
4. **Listen (*lists*)**

Liste as Abstrakter Datentyp

type List(T)

operators

$\text{empty}: \_ \rightarrow \text{List}$

$\text{is\_empty}: \text{List} \rightarrow \text{Bool}$

$\text{add}: \text{List} \times \text{T} \rightarrow \text{List}$

$\text{head}: \text{List} \rightarrow \text{T}$

$\text{tail}: \text{List} \rightarrow \text{List}$

axioms

$\forall l \in \text{List}, x \in \text{T}: \quad \text{head}(\text{add}(l, x)) = x$

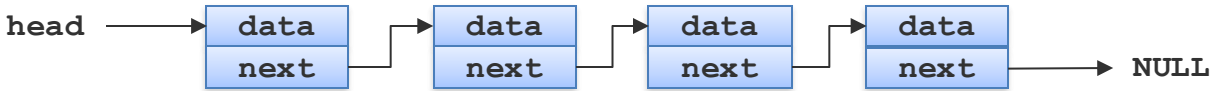
$\forall l \in \text{List}, x \in \text{T}: \quad \text{tail}(\text{add}(l, x)) = l$

$\forall l \in \text{List}, x \in \text{T}: \quad \text{is\_empty}(\text{add}(l, x)) = \text{false}$

$\text{is\_empty}(\text{empty}) = \text{true}$

Verkettete Liste

- Bisherige Datenstrukturen: **statisch**
  - Können zur Laufzeit nicht wachsen bzw. schrumpfen
  - Dadurch keine Anpassung an tatsächlichen Speicherbedarf
- **Ausweg**: Dynamische Datenstrukturen, im Besonderen *Verkettete Liste*
  - Menge von Knoten, die jeweils einen Verweis auf Nachfolgerknoten (`next`) sowie das zu speichernde Element (`data`) enthalten
  - **Listenkopf**: spezieller Knoten `head`
    - Alle anderen Knoten nur durch Navigation erreichbar
  - **Listenende**: `NULL`-Zeiger



bag.h

Stacks und Queues mit verketteten Listen

- **Stacks**
  - `push`: Fügt einen neuen Listenkopf ein
  - `pop`: Entfernt den Listenkopf
  - `top`: Gibt den Listenkopf zurück

- **Queues**
  - `enqueue`: Fügt einen neues Listenende ein
  - `dequeue`: Entfernt den Listenkopf
  - `front`: Gibt den Listenkopf zurück

list_stack.h

list_queue.h

Zusammenfassung

- **Abstrakte Datentypen**
  - Beschreibung von Datenstrukturen unabhängig von ihrer späteren Implementierung in einer konkreten Programmiersprache
  - Bestehen aus *Operatoren* und *Axiomen* - erlauben theoretische Analyse
  - Direkt abbildbar in objekt-orientierter Programmierung
- **Stapel**
  - LIFO-Datenstruktur die sehr oft benutzt wird in Algorithmenentwicklung
  - Hilfreich für die effiziente Auswertung von (arithmetischen) Ausdrücken
- **Queue**
  - FIFO-Datenstruktur die oft auf Betriebssystemebene eingesetzt wird
  - Effizient über zwei Zeiger implementierbar
- **Listen**
  - Am häufigsten benutzter abstrakter Datentyp
  - Stacks und Queues lassen sich leicht über verkettete Listen implementieren

Hasso Plattner Institut
Digital Engineering · Universität Potsdam

Viel Spaß bis zur nächsten Vorlesung!