

## 23. Dynamische Datentypen und Speicherverwaltung

### Problem

Letzte Woche: Dynamischer Datentyp

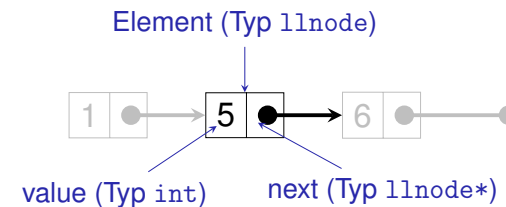
Haben im Vektor dynamischen Speicher angelegt, aber nicht wieder freigegeben. Insbesondere: keine Funktionen zum Entfernen von Elementen aus `llvec`.

Heute: Korrektes Speichermanagement!

### Ziel: Stapelklasse mit Speichermanagement

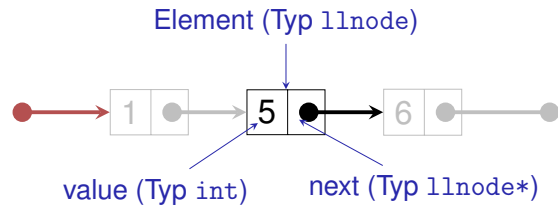
```
class stack{
public:
    // post: Element auf den Stapel legen
    void push(int value);
    // pre: Stack nicht leer
    // post: Entfernt oberstes Element vom Stapel
    void pop();
    // pre: Stack nicht leer
    // post: Gibt Wert des obersten Elementes zurück
    int top() const;
    // post: gibt zurück, ob Stack leer ist
    bool empty() const;
    // post: gibt den Stapel aus
    void print(std::ostream& out) const;
    ...
};
```

### Erinnerung: Verkettete Liste



```
struct llnode {
    int value;
    llnode* next;
    // constructor
    llnode (int v, llnode* n) : value (v), next (n) {}
};
```

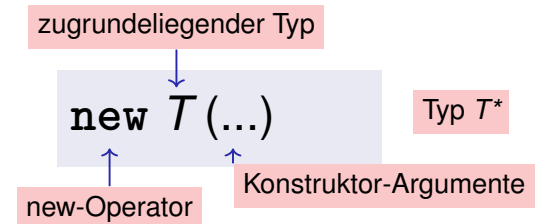
## Stapel = Zeiger aufs oberste Element



```
class stack {
public:
    void push (int value);
    ...
private:
    llnode* topn;
};
```

700

## Erinnerung: der new-Ausdruck



- **Effekt:** Neues Objekt vom Typ  $T$  wird im Speicher angelegt ...
- ... und mit Hilfe des passenden Konstruktors initialisiert.
- **Wert:** Adresse des neuen Objekts

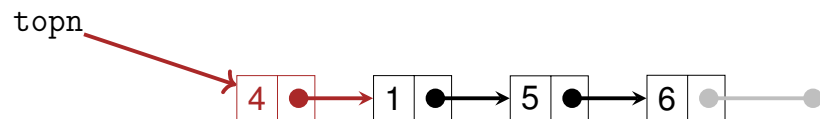
701

## Der new-Ausdruck:

push(4)

- **Effekt:** Neues Objekt vom Typ  $T$  wird im Speicher angelegt ...
- ... und mit Hilfe des passenden Konstruktors initialisiert.
- **Wert:** Adresse des neuen Objekts

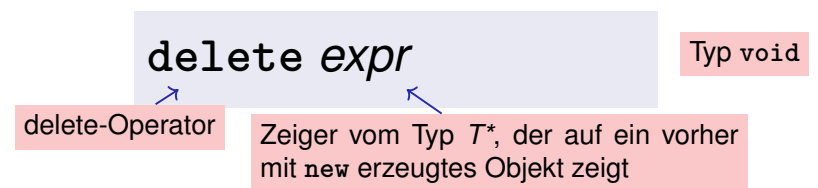
```
void stack::push(int value){
    topn = new llnode (value, topn);
}
```



702

## Der delete-Ausdruck

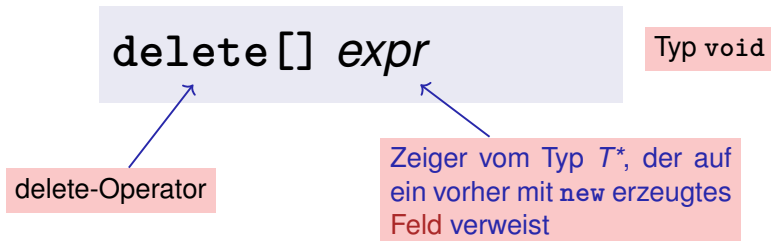
Objekte, die mit `new` erzeugt worden sind, haben *dynamische Speicherdauer*: sie "leben", bis sie explizit *gelöscht* werden.



- **Effekt:** Objekt wird *dekonstruiert* (Erklärung folgt) ... und *Speicher wird freigegeben*.

703

## Der delete-Ausdruck für Felder



- **Effekt:** Feld wird gelöscht, Speicher wird wieder freigegeben

704

## Wer geboren wird, muss sterben...

### Richtlinie "Dynamischer Speicher"

Zu jedem `new` gibt es ein passendes `delete`!

Nichtbeachtung führt zu *Speicherlecks*:

- "Alte" Objekte, die den Speicher blockieren...
- ... bis er irgendwann voll ist (**heap overflow**)

705

## Aufpassen mit new und delete!

```
rational* t = new rational; ← Speicher für t wird angelegt
rational* s = t; ← Auch andere Zeiger können auf das Objekt zeigen..
delete s; ← ... und zur Freigabe verwendet werden.
int nominator = (*t).denominator(); > Fehler: Speicher freigegeben!
```

↑  
Dereferenzieren eines „dangling pointers“

- Zeiger auf freigegebene Objekte: hängende Zeiger (*dangling pointers*)
- Mehrfache Freigabe eines Objektes mit `delete` ist ein ähnlicher schwerer Fehler.

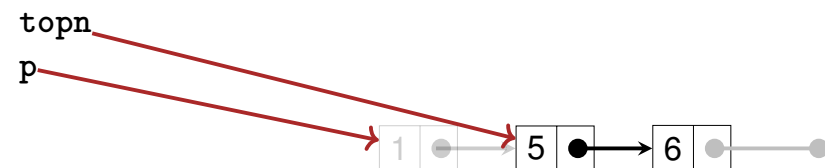
706

## Weiter mit dem Stapel:

pop()

```
void stack::pop(){
    assert (!empty());
    llnode* p = topn;
    topn = topn->next;
    delete p;
}
```

↑  
Erinnerung: Abkürzung für (\*topn).next

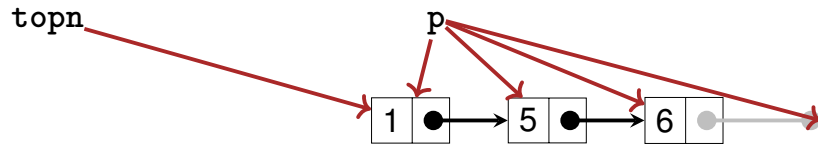


707

## Stapel ausgeben:

print()

```
void stack::print (std::ostream& out) const {
    for(const llnode* p = topn; p != nullptr; p = p->next)
        out << p->value << " "; // 1 5 6
}
```



708

## Stapel ausgeben:

operator<<

```
class stack {
public:
    void push (int value);
    void pop();
    void print (std::ostream& o) const;
    ...
private:
    llnode* topn;
};

// POST: s is written to o
std::ostream& operator<< (std::ostream& o, const stack& s){
    s.print (o);
    return o;
}
```

709

## empty(), top()

```
bool stack::empty() const {
    return top == nullptr;
}
```

```
int stack::top() const {
    assert(!empty());
    return topn->value;
}
```

710

## Leerer Stapel

```
class stack{
public:
    stack() : topn (nullptr) {} // default constructor

    void push(int value);
    void pop();
    void print(std::ostream& out) const;
    int top() const;
    bool empty() const;
private:
    llnode* topn;
}
```

711

## Zombie-Elemente

```
{
  stack s1; // lokale Variable
  s1.push (1);
  s1.push (3);
  s1.push (2);
  std::cout << s1 << "\n"; // 2 3 1
}
// s1 ist gestorben (nicht mehr zugreifbar)
```

- ... aber die drei *Elemente* des Stapels s1 leben weiter (Speicherleck)!
- Sie sollten zusammen mit s1 aufgeräumt werden!

712

## Der Destruktor

- Der Destruktor einer Klasse  $T$  ist die eindeutige Memberfunktion mit Deklaration

$$\sim T ();$$

- Wird automatisch aufgerufen, wenn die Speicherdauer eines Klassenobjekts vom Typ  $T$  endet – z.B. bei Aufruf von `delete` auf einem Objekt vom Typ  $T^*$  oder wenn der Gültigkeitsbereich eines Objektes vom Typ  $T$  endet.
- Falls kein Destruktor deklariert ist, so wird er automatisch erzeugt und ruft die Destruktoren für die Membervariablen auf (Zeiger `topn`, kein Effekt – Grund für Zombie-Elemente)

713

## Mit dem Destruktor klappt's!

```
// POST: the dynamic memory of *this is deleted
stack::~~stack(){
  while (topn != nullptr){
    llnode* t = topn;
    topn = t->next;
    delete t;
  }
}
```

- löscht automatisch alle Stapel-elemente, wenn der Stapel ungültig wird
- Unsere Stapel-Klasse scheint jetzt die Richtlinie "Dynamischer Speicher" zu befolgen (?)

714

## Stapel fertig?

Offenbar noch nicht...

```
stack s1;
s1.push (1);
s1.push (3);
s1.push (2);
std::cout << s1 << "\n"; // 2 3 1
```

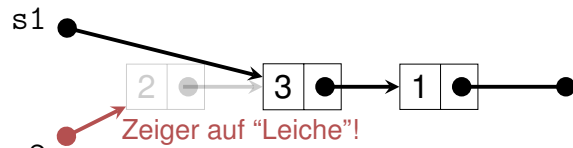
```
stack s2 = s1;
std::cout << s2 << "\n"; // 2 3 1
```

```
s1.pop ();
std::cout << s1 << "\n"; // 3 1
```

```
s2.pop (); // Oops, Programmabsturz!
```

715

## Was ist hier schiefgegangen?



Memberweise Initialisierung: kopiert  
nur den topn-Zeiger

```
...  
stack s2 = s1;  
std::cout << s2 << "\n"; // 2 3 1
```

```
s1.pop ();  
std::cout << s1 << "\n"; // 3 1
```

```
s2.pop (); // Oops, Programmabsturz!
```

## Das eigentliche Problem

Schon das geht schief:

```
{  
    stack s1;  
    s1.push(1);  
    stack s2 = s1;  
}
```

Beim Verlassen des Gültigkeitsbereiches werden beide Stacks aufgeräumt (dekonstruiert). Aber beide Stacks versuchen dieselben Daten zu löschen, denn sie haben *Zugriff auf denselben Zeiger*.

716

717

## Lösungsmöglichkeiten

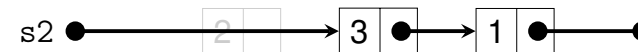
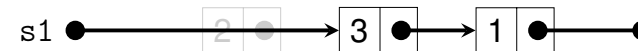
Smart-Pointers (werden hier nicht weiter vertieft):

- Zähle die Anzahl Zeiger, die auf ein Objekt verweisen. Lösche nur wenn diese Anzahl auf 0 zurückfällt: `std::shared_pointer`
- Verhindere, dass mehrere Zeiger auf ein Objekt zeigen können: `std::unique_pointer`.

oder:

- Wir erstellen eine echte Kopie aller Daten – wie folgt.

## Wir erstellen eine echte Kopie!



```
...  
stack s2 = s1;  
std::cout << s2 << "\n"; // 2 3 1
```

```
s1.pop ();  
std::cout << s1 << "\n"; // 3 1
```

```
s2.pop (); // ok
```

718

719

## Der Copy-Konstruktor

- Der Copy-Konstruktor einer Klasse  $T$  ist der eindeutige Konstruktor mit Deklaration

```
T ( const T& x );
```

- wird automatisch aufgerufen, wenn Werte vom Typ  $T$  mit Werten vom Typ  $T$  *initialisiert* werden

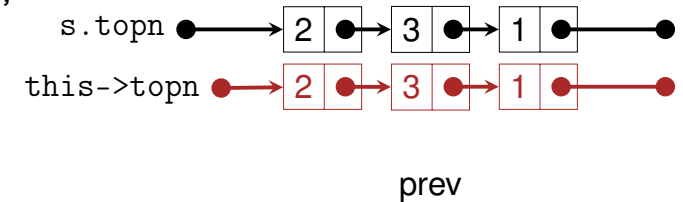
```
T x = t;    (t vom Typ T)
```

```
T x (t);
```

- Falls kein Copy-Konstruktor deklariert ist, so wird er automatisch erzeugt (und initialisiert memberweise – Grund für obiges Problem)

## Mit dem Copy-Konstruktor klappt's!

```
// POST: *this is initialized with a copy of s
stack::stack (const stack& s) : topn (nullptr) {
    if (s.topn == nullptr) return;
    topn = new llnode(s.topn->value, nullptr);
    llnode* prev = topn;
    for(llnode* n = s.topn->next; n != nullptr; n = n->next){
        llnode* copy = new llnode(n->value, nullptr);
        prev->next = copy;
        prev = copy;
    }
}
```



720

721

## NB: rekursives Kopieren

```
llnode* copy (node* that){
    if (that == nullptr) return nullptr;
    return new llnode(that->value, copy(that->next));
}
```

Elegant, oder? Warum haben wir das nicht gleich so gemacht?

Grund: verkettete Listen können sehr lang werden. Dann könnte *copy* zum Stapelüberlauf<sup>7</sup> führen. Aufrufstapel ist nämlich meist kleiner als Heapspeicher.

<sup>7</sup>nicht von dem Stapel, den wir gerade implementieren, sondern vom Aufrufstapel der Rekursion

## Initialisierung $\neq$ Zuweisung!

```
stack s1;
s1.push (1);
s1.push (3);
s1.push (2);
std::cout << s1 << "\n"; // 2 3 1
```

```
stack s2;
s2 = s1; // Zuweisung
```

```
s1.pop ();
std::cout << s1 << "\n"; // 3 1
s2.pop (); // Oops, Programmabsturz!
```

722

723

## Der Zuweisungsoperator

- Überladung von `operator=` als Memberfunktion
- Wie Copy-Konstruktor ohne Initialisierer, aber zusätzlich
  - Freigabe des Speichers für den „alten“ Wert
  - Prüfen auf Selbstzuweisungen (`s1=s1`), die keinen Effekt haben sollen
- Falls kein Zuweisungsoperator deklariert ist, so wird er automatisch erzeugt (und weist memberweise zu – Grund für obiges Problem)

724

## Mit dem Zuweisungsoperator klappt's!

```
// POST: *this (left operand) becomes a
//           copy of s (right operand)
stack& stack::operator= (const stack& s){
    if (topn != s.topn){ // keine Selbstzuweisung
        stack copy = s; // Kopierkonstruktor
        std::swap(topn, copy.topn); // copy hat nun den Müll!
    } // copy wird aufgeräumt -> Dekonstruktion
    return *this; // Rueckgabe als L-Wert (Konvention)
}
```

Cooler Trick! 😊

725

## Fertig

```
class stack{
public:
    stack(); // constructor
    ~stack(); // destructor
    stack(const stack& s); // copy constructor
    stack& operator=(const stack& s); // assignment operator

    void push(int value);
    void pop();
    int top() const;
    bool empty() const;
    void print(std::ostream& out) const;
private:
    llnode* topn;
}
```

726

## Dynamischer Datentyp

- Typ, der dynamischen Speicher verwaltet (z.B. unsere Klasse für Stapel)
- Mindestfunktionalität:
  - Konstruktoren
  - Destruktor
  - Copy-Konstruktor
  - Zuweisungsoperator

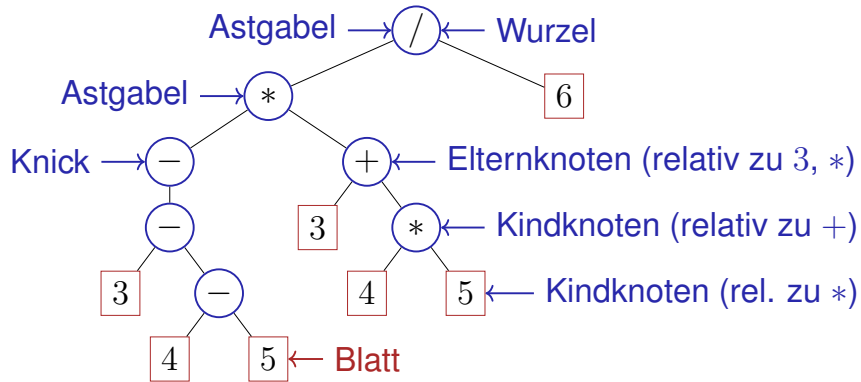
*Dreierregel:* definiert eine Klasse eines davon, so muss sie auch die anderen zwei definieren!

727

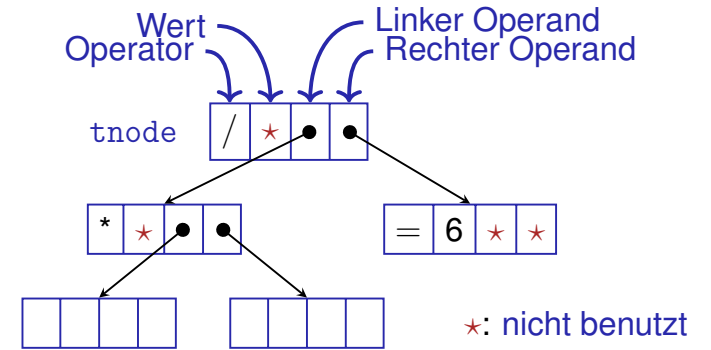
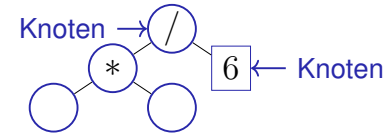


# (Ausdrucks-)Bäume

$$-(3-(4-5))*(3+4*5)/6$$



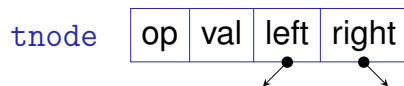
# Astgabeln + Blätter + Knicke = Knoten



728

729

# Knoten (struct tnode)



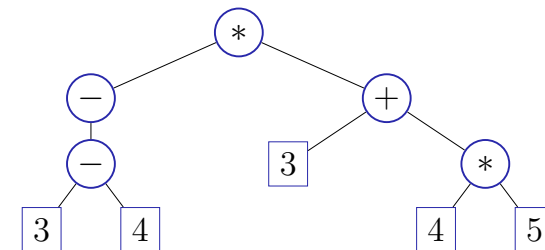
```

struct tnode {
    char op; // leaf node: op is '='
            // internal node: op is '+', '-', '*', or '/'
    double val;
    tnode* left; // == nullptr for unary minus
    tnode* right;

    tnode(char o, double v, tnode* l, tnode* r)
        : op(o), val(v), left(l), right(r) {}
};
    
```

730

# Grösse = Knoten in Teilbäumen zählen



- Grösse eines Blattes: 1
- Grösse anderer Knoten: 1 + Gesamtgrösse aller Kindknoten
- Z.B. Grösse des „+“-Knoten ist 5

731

## Knoten in Teilbäumen zählen

```
// POST: returns the size (number of nodes) of
//       the subtree with root n
int size (const tnode* n) {
    if (n){ // shortcut for n != nullptr
        return size(n->left) + size(n->right) + 1;
    }
    return 0;
}
```



732

## Teilbäume auswerten

```
// POST: evaluates the subtree with root n
double eval(const tnode* n){
    assert(n);
    if (n->op == '=') return n->val; ← Blatt...
    double l = 0;                               ...oder Astgabel:
    if (n->left) l = eval(n->left); ← op unär, oder linker Ast
    double r = eval(n->right); ← rechter Ast
    switch(n->op){
        case '+': return l+r;
        case '-': return l-r;
        case '*': return l*r;
        case '/': return l/r;
        default: return 0;
    }
}
```



733

## Teilbäume klonen

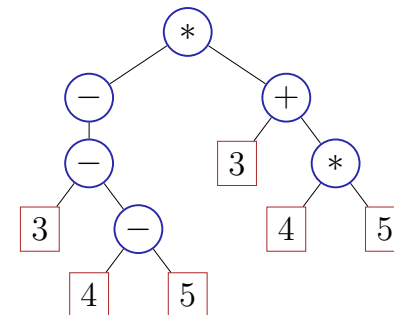
```
// POST: a copy of the subtree with root n is made
//       and a pointer to its root node is returned
tnode* copy (const tnode* n) {
    if (n == nullptr)
        return nullptr;
    return new tnode (n->op, n->val, copy(n->left), copy(n->right));
}
```



734

## Teilbäume fällen

```
// POST: all nodes in the subtree with root n are deleted
void clear(tnode* n) {
    if(n){
        clear(n->left);
        clear(n->right);
        delete n;
    }
}
```



735

## Teilbäume nutzen

```
// Construct a tree for 1 - (-(3 + 7))
tnode* n1 = new tnode('=', 3, nullptr, nullptr);
tnode* n2 = new tnode('=', 7, nullptr, nullptr);
tnode* n3 = new tnode('+', 0, n1, n2);
tnode* n4 = new tnode('-', 0, nullptr, n3);
tnode* n5 = new tnode('=', 1, nullptr, n4);
tnode* root = new tnode('-', 0, n5, n4);

// Evaluate the overall tree
std::cout << "1 - (-(3 + 7)) = " << eval(root) << '\n';

// Evaluate a subtree
std::cout << "3 + 7 = " << eval(n3) << '\n';

clear(root); // free memory
```

736

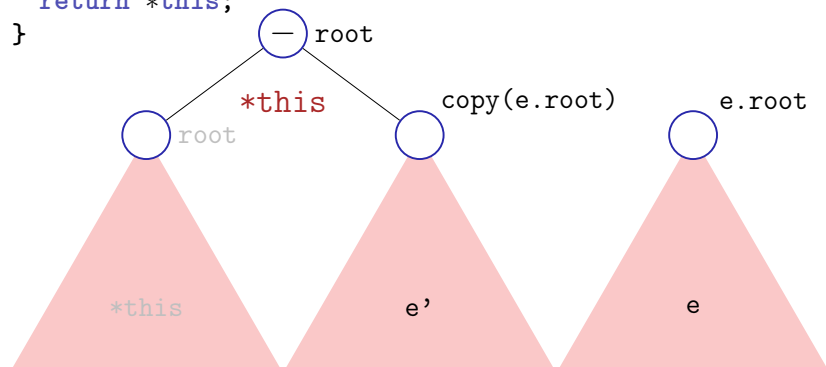
## Bäume pflanzen

```
class texpression {
public:
    texpression (double d) ← erzeugt Baum mit einem Blatt
        : root (new tnode ('=', d, 0, 0)) {}
    ...
private:
    tnode* root;
};
```

737

## Bäume wachsen lassen

```
texpression& texpression::operator-= (const texpression& e)
{
    assert (e.root);
    root = new tnode ('-', 0, root, copy(e.root));
    return *this;
}
```

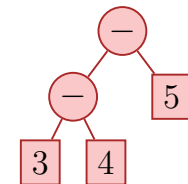


738

## Bäume züchten

```
texpression operator- (const texpression& l,
                      const texpression& r){
    texpression result = l;
    return result -= r;
}
```

```
texpression a = 3;
texpression b = 4;
texpression c = 5;
texpression d = a-b-c;
```



739

## Dreierregel: Bäume klonen, reproduzieren und fällen

```
expression::~~expression(){
    clear(root);
}

expression::expression (const expression& e)
    : root(copy(e.root)) { }

expression& expression::operator=(const expression& e){
    if (root != e.root){
        expression cp = e;
        std::swap(cp.root, root);
    }
    return *this;
}
```

740

## Zusammengefasst

```
class expression{
public:
    expression (double d); // constructor
    ~expression(); // destructor
    expression (const expression& e); // copy constructor
    expression& operator=(const expression& e); // assignment op
    expression operator-();
    expression& operator-=(const expression& e);
    expression& operator+=(const expression& e);
    expression& operator*=(const expression& e);
    expression& operator/=(const expression& e);
    double evaluate();
private:
    tnode* root;
};
```

741

## Werte zu Bäumen!

```
using number_type = expression ;
```

```
// term = factor { "*" factor | "/" factor }
number_type term (std::istream& is){
    number_type value = factor (is);
    while (true) {
        if (consume (is, '*'))
            value *= factor (is);
        else if (consume (is, '/'))
            value /= factor (is);
        else
            return value;
    }
}
```

```
double_calculator.cpp
(Ausdruckswert)
→
texpression_calculator.cpp
(Ausdrucksbaum)
```

742

## Abschliessende Bemerkung

- Wir haben in dieser Vorlesung die Knoten für Liste und Baum bewusst ohne Memberfunktionen implementiert. Wir betrachten sie nämlich als reine Datencontainer ohne eigene Intelligenz.<sup>8</sup>
- Wenn Vererbung und Polymorphie im Spiel ist, ist die Implementation der Funktionalität wie `evaluate`, `print`, `clear`, `copy` (etc.) mit Memberfunktionen vorzuziehen.
- In jedem Falle implementiert man die Speicherverwaltung der zusammengesetzten Datenstruktur Liste / Baum nicht in den Knotenklassen.

<sup>8</sup>Teile der Implementation waren so sogar einfacher, da der Fall `n==nullptr` einfacher abgefangen werden kann

743