Grundgebiete der Informatik 2: Algorithmen und Programmiertechniken

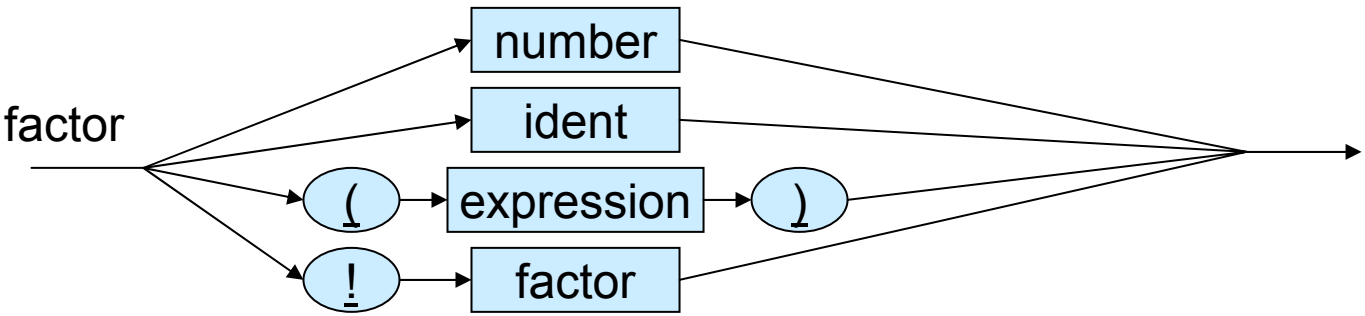
Tutorium zur Klausurvorbereitung

Fr 30.07.2004, 14:00 – 15:30 AH I

Mo 02.08.2004, 14:00 – 15:30 AH I

EBNF

- Gibt mögliche Programmstruktur vor
- Darstellungsform für *kontextfreie Grammatiken*
- Äquivalent zu *Syntaxdiagrammen*
- Sequenz, Option [ ], Wiederholung{ }, Alternative |, Rekursion



Rekursion

factor ::= number | ident | (expression) | ! factor

Nichtterminal-Symbol

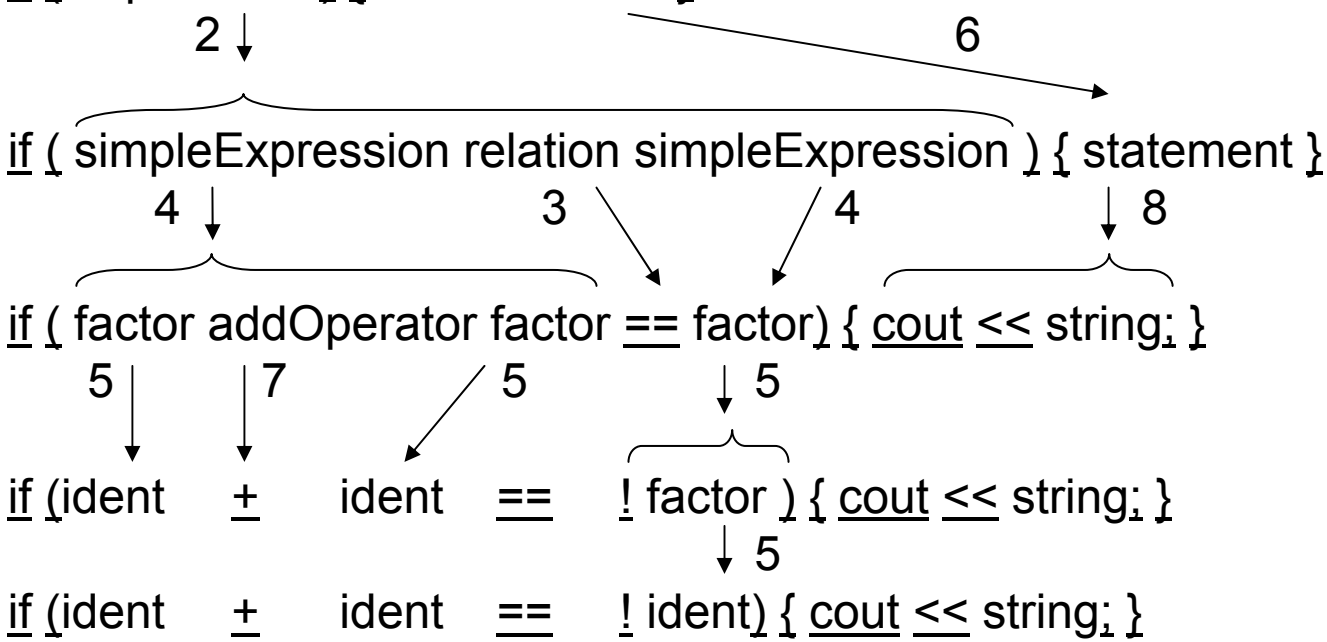
Metasymbole der EBNF

Terminal-Symbol

Ableiten eines Worts aus einer EBNF

- Wort: Folge von Terminalsymbolen, die durch Anwendung von Regeln einer Grammatik/EBNF erzeugt wurde
- Beispiel bezieht sich auf EBNF aus Aufgabe 1

```
if ( a + b == !c) { cout << "test"; }
```



Ebenen der Syntax & Semantik

- Lexikalische Syntax:
  Schlüsselwörter, Bezeichner, Literale, Begrenzer, ...
- Kontextfreie Syntax (Aufbausyntax):
  Beschreibt Aufbau eines Programms
  Ausdrücke, Anweisungen, Deklarationen, Programmstruktur

EBNF

- Kontextsensitive Syntax:
  Beschreibt Beziehungen zwischen Teilen eines Programms

Umgangs-sprachlich

- Semantik:
  Beschreibt die Bedeutung der Elemente, aus denen ein Programm aufgebaut ist

Umgangs-sprachlich

Beispiel für Ebenen der Syntax

```
#include <iostream.h>

int summiere(int a, int b) {
    return a + b ;
}

int main() {
    int a, b, summe ;

    cin >> a >> b;
    summe = summiere(a,b);

    cout << summe << endl;
}
```

alte Klausur: Aufgabe mit Fehlersuche

Lexikalische Syntax

- Literale: `1.8e6` ; ``` ``Hallo`` ```
- Schlüsselworte: `int`, `return`
- Bezeichner: `summiere`, `a`, `b`
- Begrenzer: `;` `{` `}` `(` `)` space tab

- Kommentare: Eine lexikalische Einheit, die aber ignoriert wird

Kontextfreie Syntax

- Ausdrücke
- Deklarationen
- Anweisungen
- Programmstruktur

Kontextsensitive Syntax

→ Zur Deklaration

Kontrollstrukturen

- Fallunterscheidungen

```
if ( zahl > 0 ) {
    // Anweisungsfolge
}
```

Einseitig bedingte Anweisung

```
if ( zahl > 0 ) {
    // Anweisungsfolge
} else {
    // Alternative A.
}
```

Zweiseitig bedingte Anweisung

Kontrollstrukturen

- Auswahlanweisung

```
enum Wochentag { Mo, Di, Mi, Do, Fr, Sa, So };
Wochentag Tag;
```

```
switch ( Tag ) {
    case Mo:
        // Anweisungen
        break;
     case Di:
     case Mi:
        // Anweisungen
        break;
    default:
        // Anweisungen
        break;
}
```

Fall „Mo“

Fälle , „`Di`“ und „`Mi`“

Alle anderen Fälle (optional)

Kontrollstrukturen

- Schleifen
  - Wiederholte Ausführung einer Anweisungsfolge (Block)
  - Unterscheidung
    - Anzahl der Iterationen bekannt
      ➔ **`for`**-Schleife

```
// Berechnung der Fakultät
int eingabe, fakultaet = 1;
cin >> eingabe;

for ( int i=1; i < eingabe; i++ ) {
     fakultaet *= i;
}
```

    - Anzahl der Iterationen unbekannt
      ➔ **`while`**-Schleife

```
// Reaktion auf Messfuehler
while ( Messfuehler.GibWert() > 0 ) {
     cout << "Aktueller Wert: "
          << Messfuehler.GibWert()
          << endl;
}

cout << "Wert ist nicht mehr positiv!" << endl;
```

Prozedur vs. Funktion vs. Methode

- Prozedur
  - Führt eine Anweisungsfolge aus
  - Basiert auf Parametern, verändert diese gegebenenfalls
  - *Kein* Rückgabewert
  - Beenden der Prozedur durch **`return;`** (optional)

```
void Ausgabe( int a ) {
    cout << a << endl;
}
```

- Funktion
  - Wie Prozedur, aber:
  - Hat Rückgabewert *ungleich* **`void`**
  - Beenden der Funktion & Rückgabe durch **`return`**-Anweisung

```
int Eingabe() {
    int eingabe;      // Auf Groß-/Kleinschreibung achten!
    cin >> eingabe;

    return eingabe;
}
```

- Methode
  - Gehört zu einer Klasse (bspw. ADT) "member function"
  - Kann Prozedur *oder* Funktion sein
  - Durch Klassenzugehörigkeit: Zugriff auf private Klassen-Elemente

Variablen, Pointer, Referenzen

Prinzipiell `varTyp varBezeichner;`

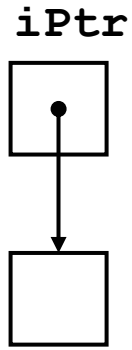
- Pointer
  - Deklaration: Vor jedem Var.-Bezeichner *
- Referenzen
  - Deklaration: Vor jedem Var.-Bezeichner &
  - Muss direkt initialisiert werden
- Dereferenzierung: * "Folgt dem Zeiger"
- Adresse von: & "Liefert Zeiger auf"

```
int a;        // Integer-Variable, Name:"a"
int *pa;      // Pointer/Zeiger auf Integer-Variable, Name:"b"
int &ra = a; // Referenz auf Integer-Variable, Name:"c"

pa = &a;      // Pointer "b" zeigt nun auf "a"
a = 10;  cout << a;
*pa = 20; cout << a;
ra = 30;  cout << a;      // Frage: Was wird jeweils ausgegeben?
```

Speicherallokation und Freigabe

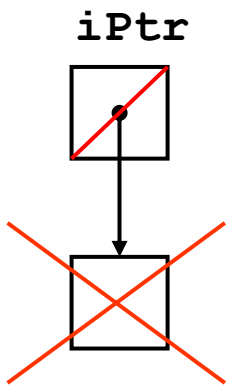
- Speicher auf dem Heap allokieren:

```
int* iPtr;

iPtr=new int;

*iPtr=42;

...
```



- Allokierten Speicher wieder freigeben:

```
delete iPtr;

iPtr=NULL;
```



Parameterübergabe: Beispiel

```
#include "iostream.h"

void f(int &x, int y, int *z) {
   y++;
   x=*z+y;
   x++;
   ++(*z);
}

int main()
{
   int a=1, b=2, c=3;
   f(a, b, &c); //1. Aufruf
   cout << "1. Aufruf: " << a << b << c << char(13) << char(10);
   f(a, b, &c); //2. Aufruf
   cout << "2. Aufruf: " << a << b << c << char(13) << char(10);
   return 0;
}
```

Call by Value

Call by Reference

	a	b	c
vor 1. Aufruf	1	2	3
nach 1. Aufruf	7	2	4
nach 2. Aufruf	8	2	5

Datentypen: vordefinierte, Typdeklaration

- Vordefinierte Typen (Beispiele):

```
[unsigned] int, char,
[unsigned] double,
short, float, ...
```

```
double dbl;
```

Objekt-deklaration

- Arrays:

```
int i[41];          int *i[41];
```

Objekt-deklarationen

- Typdeklaration:

```
typedef int *pIntT;
```

- Verwendung:

```
pIntT pi;
```

Objekt-deklaration

Datentypen: Arrays, Strukturen, ...

- Aufzählungstypen

```
[typedef] enum Werktag {di, mi};
```

- Arrays

```
typedef int* IntPtrFeld[41];
```

- Strukturen

```
[typedef] struct MyStruct {
    int i;
    MyStruct* next;
};
```

Typ-deklaration

- Verwendung

```
MyStruct aStruct;
IntPtrFeld anArray;
```

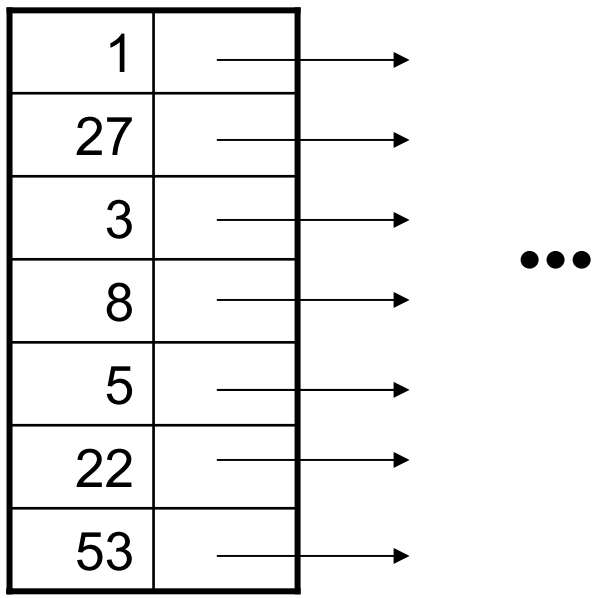
Objekt-deklarationen

Datentypen: komplexes Beispiel

```
struct MyStruct {
      int i;
      SomeType* info;
};
```

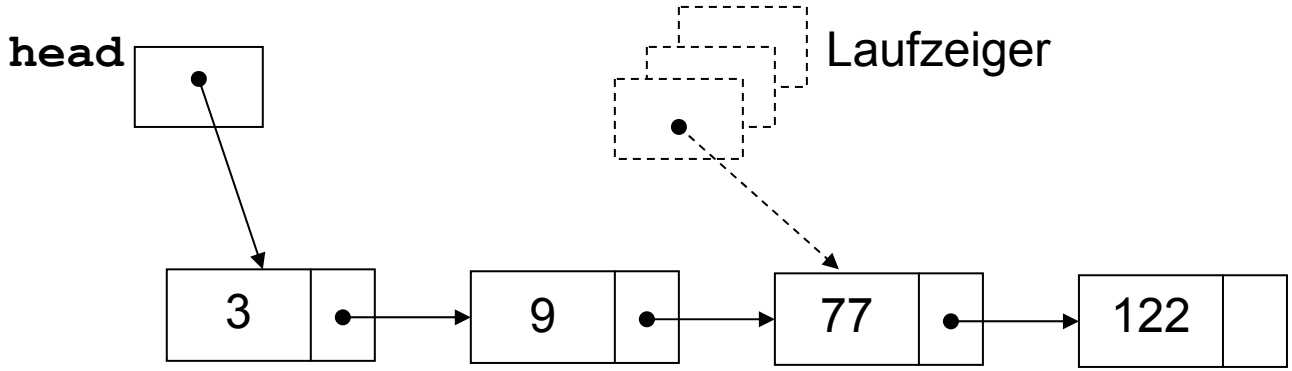
```
typedef MyStruct ComplexArray[7];
```

```
ComplexArray anArray;
```

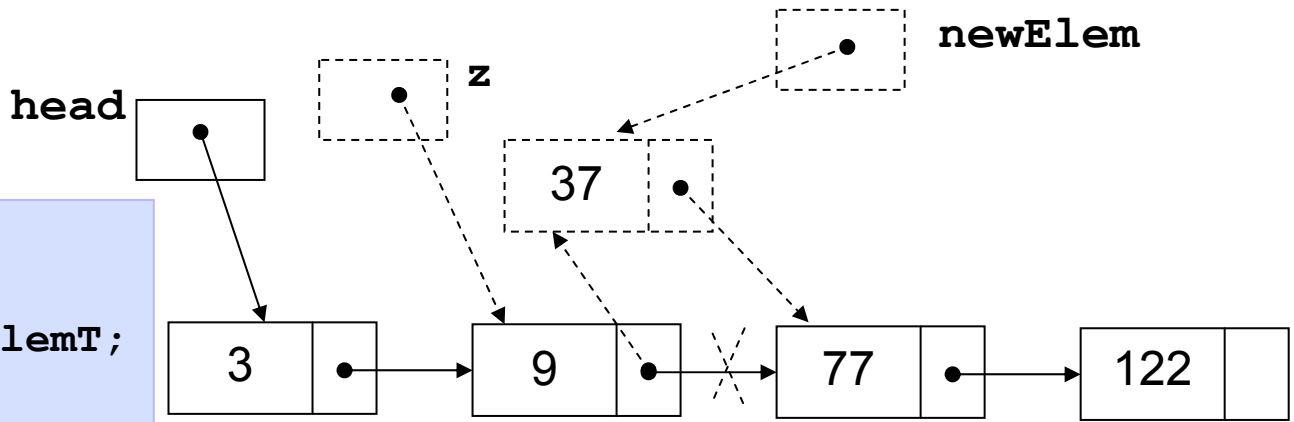


Datenstrukturen: Verkettete Liste

```
struct lElemT{
   int value;
   lElemT *next;
};

lElemT* head;
...
head=NULL;
```



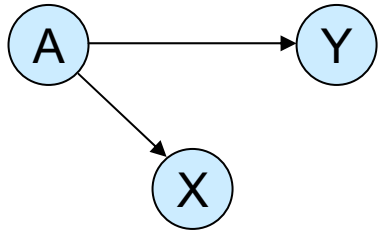
z.B. Element einfügen




```
// z: Einfuegestelle

lElemT *newElem=new lElemT;
newElem->value=37;

newElem->next=z->next;
z->next=newElem;
```

Datenstrukturen: Graphen

- Knotenmenge, Kantenmenge



- Repräsentations-Möglichkeiten:
  - knotenorientiert - Adjazenzliste

$A \rightarrow \{ X, Y \}$

  - knotenorientiert - Adjazenzmatrix

	...	X	...	Y	
...					
A		1		1	
...					

  - kantenorientiert - Kanten explizit gespeichert

{ (A, X), ..., (A, Y) }

- Mögliche Realisierungen:
  - sequentiell (Arrays)
  - verkettete Listen

Aufwandsabschätzung in O-Notation

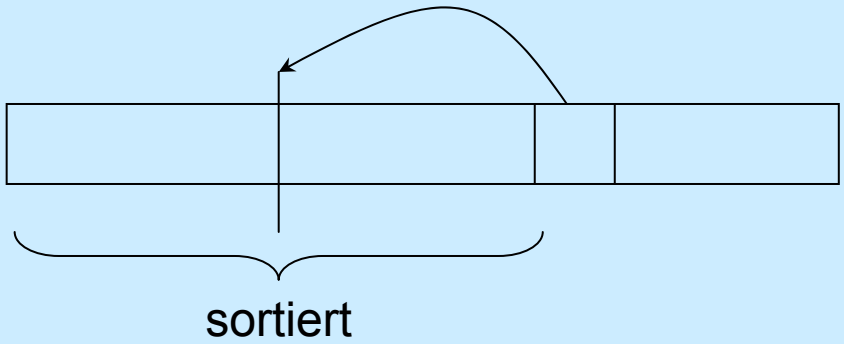
- O-Notation
  - Gibt Größenordnung an
  - Abstrahiert von "unwichtigen" Details/Faktoren
  - Oft Abschätzung für besten, mittleren und schlechtesten Fall

z.B.:

- Quadratischer Aufwand: $a \cdot n^2 + b \cdot n + c$ ➔ $O(n^2)$
- Logarithmischer Aufwand: $a \cdot log(n) + c$ ➔ $O(log(n))$
- Linearer Aufwand $a \cdot n + c$ ➔ $O(n)$
- Konstanter Aufwand c ➔ $O(1)$

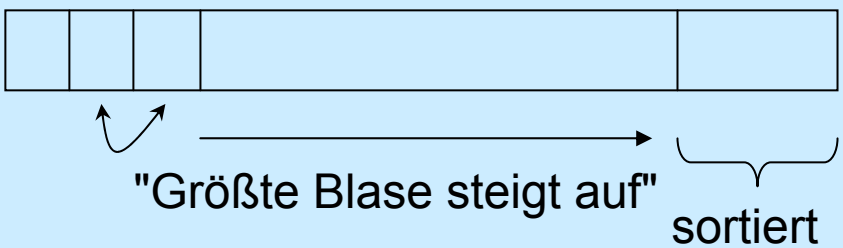
Sortierverfahren (1)

Sortieren durch Einfügen (Straight Insertion)



Bester Fall: O(n)
Mittlerer Fall: O(n²)
Schlechtester Fall: O(n²)

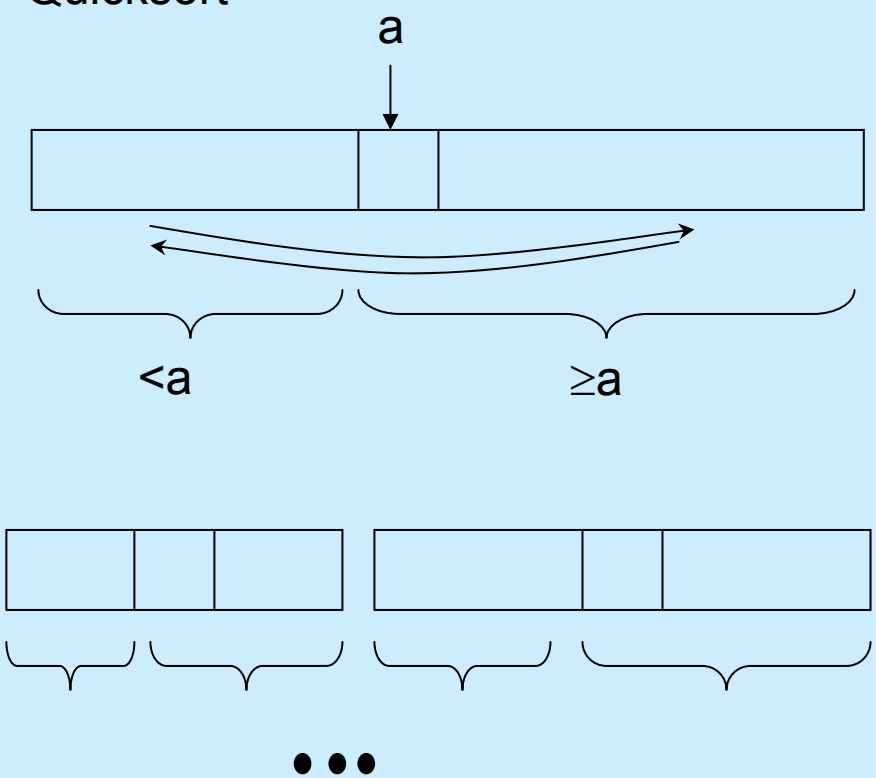
Bubblesort



Bester Fall: O(n)
Mittlerer Fall: O(n²)
Schlechtester Fall: O(n²)

Sortierverfahren (2)

Quicksort



Bester Fall: $O(n \log_2 n)$
Mittlerer Fall: $O(n \log_2 n)$
Schlechtester Fall: $O(n^2)$

Rekursion!

Suchverfahren

- Lineare Suche:
  - Laufe von vorne nach hinten bis Element gefunden

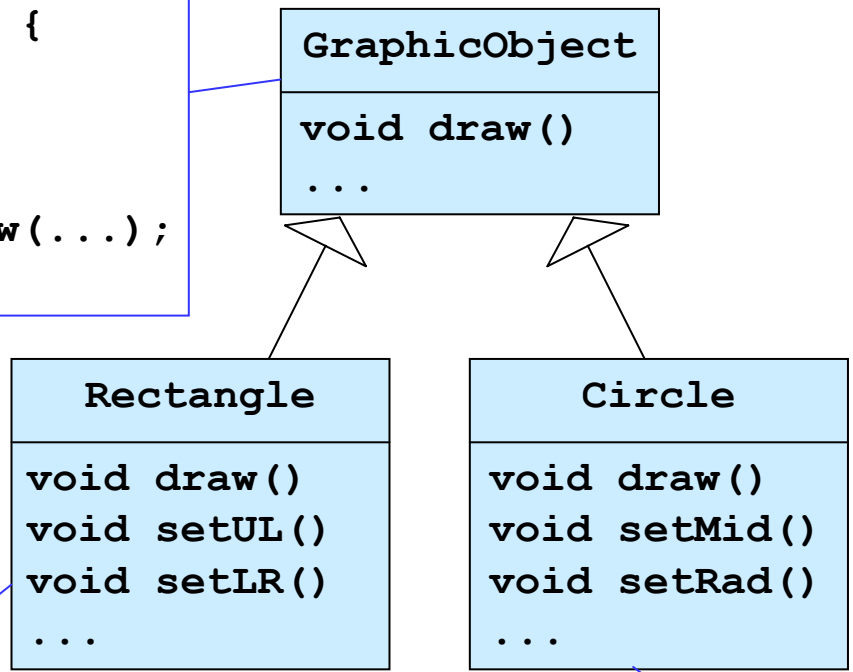
Bester Fall: O(1)
Mittlerer Fall: O(n)
Schlechtester Fall: O(n)

- Binäre Suche:
  - Sortierung vorausgesetzt !
  - vergleiche mit mittlerem Element:
    - gleich: gefunden
    - kleiner: wiederhole Suche in linker Hälfte
    - größer: wiederhole Suche in rechter Hälfte
  - wenn Hälfte leer: nicht gefunden

Bester Fall: O(1)
Mittlerer Fall: $O(\log_2 n)$
Schlechtester Fall: $O(\log_2 n)$

Objektorientierung: Klassen und Vererbung

```
class GraphicObject {
private:
   ...
public:
   virtual void draw(...);
}
```




```
class Rectangle:
   public GraphicObject {
private:
   ...
public:
   void draw(...);
   void setUL(int x, int y);
   void setLR (int x, int y);
}
```

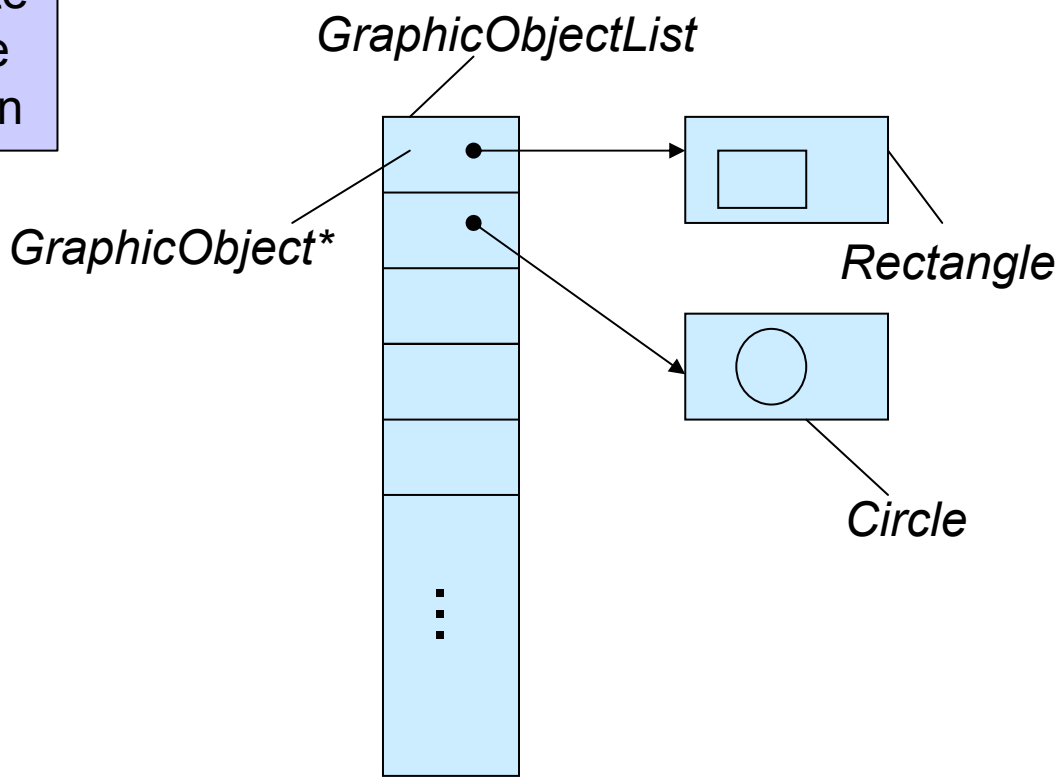
```
class Circle:
   public GraphicObject {
private:
   ...
public:
   void draw(...);
   void setMid(int x, int y);
   void setRad(int x, int y);
}
```

Objektorientierung: Dynamische Bindung

```
GraphicObject *go;
GraphicObjectList gl;

go=new Rectangle();
go->setUL(10,10);
go->setLR(15,25);

gl.add(go);

go=new Circle();
go->setMid(25,30);
go->setRad(10);

gl.add(go);

gl.gotoFirst();
while (gl.hasCurrent()) {
   go=gl.getCurrent();
   go->draw();
   gl.gotoNext();
}
```

```
class GraphicObject {
private:
   ...
public:
   virtual void draw(...);
};
```

speziellste Methode ausführen



Objektorientierung: Vorsicht mit `virtual`!

```
class A {
public:
   void f();
   virtual void g();
};
```

```
class B : public A {
public
   void f();
   void g();
};
```

```
void A::f() {
   cout << "f von A";
}
void A::g() {
   cout << "g von A";
}
```

```
void B::f() {
   cout << "f von B";
}
void B::g() {
   cout << "g von B";
}
```

```
void main() {
  A *a;
  B *b;

  b=new B();
  b->f();
  b->g();

  a=b;
  a->f();
  a->g();
}
```

f und g von B

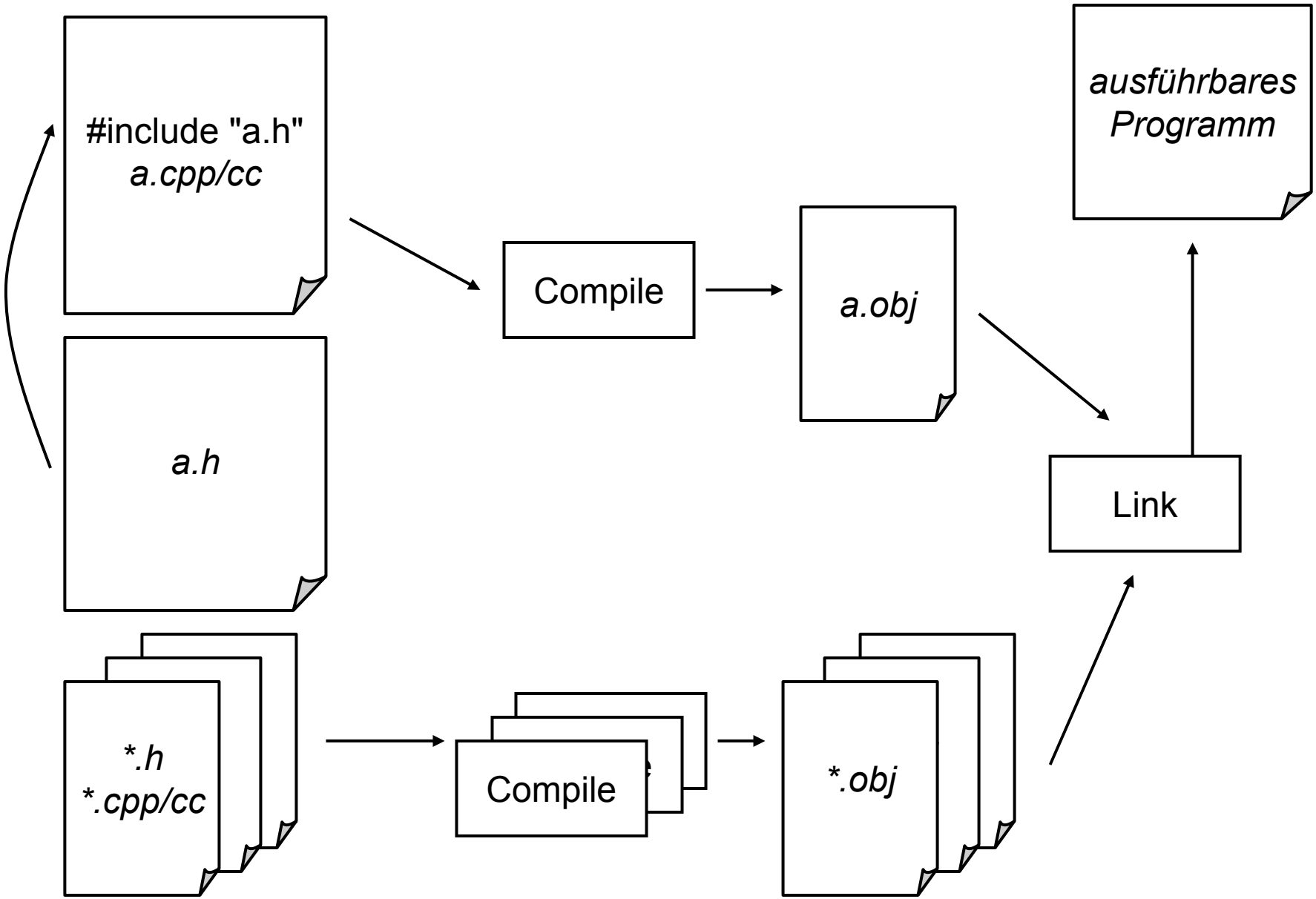
kein virtual ➔ f von A

virtual ➔ g von B

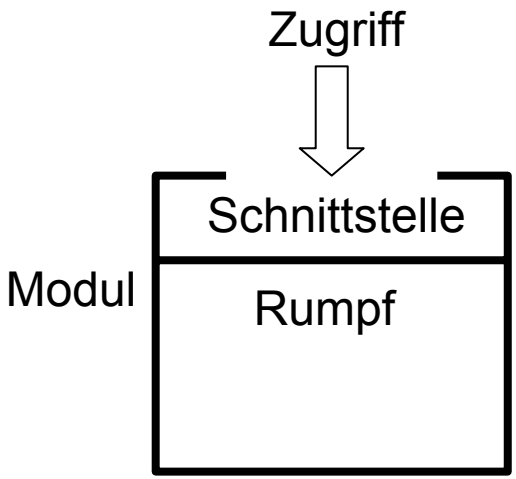
Ausgabe:

```
f von B
g von B
f von A
g von B
```

Getrennte Übersetzung



Module



- Datenabstraktion
  - abstrahiert von Realisierung der Datenstruktur
- Funktionale Abstraktion
  - abstrahiert vom Algorithmus

- Ziele:
  - Änderungen im Rumpf ohne Auswirkungen auf Verwender, Wartbarkeit
  - Zergliederung in Teile
    - Arbeitsteilung
    - Modul-Test
  - Entwurf vor Realisierung, Softwarearchitektur

Modularten

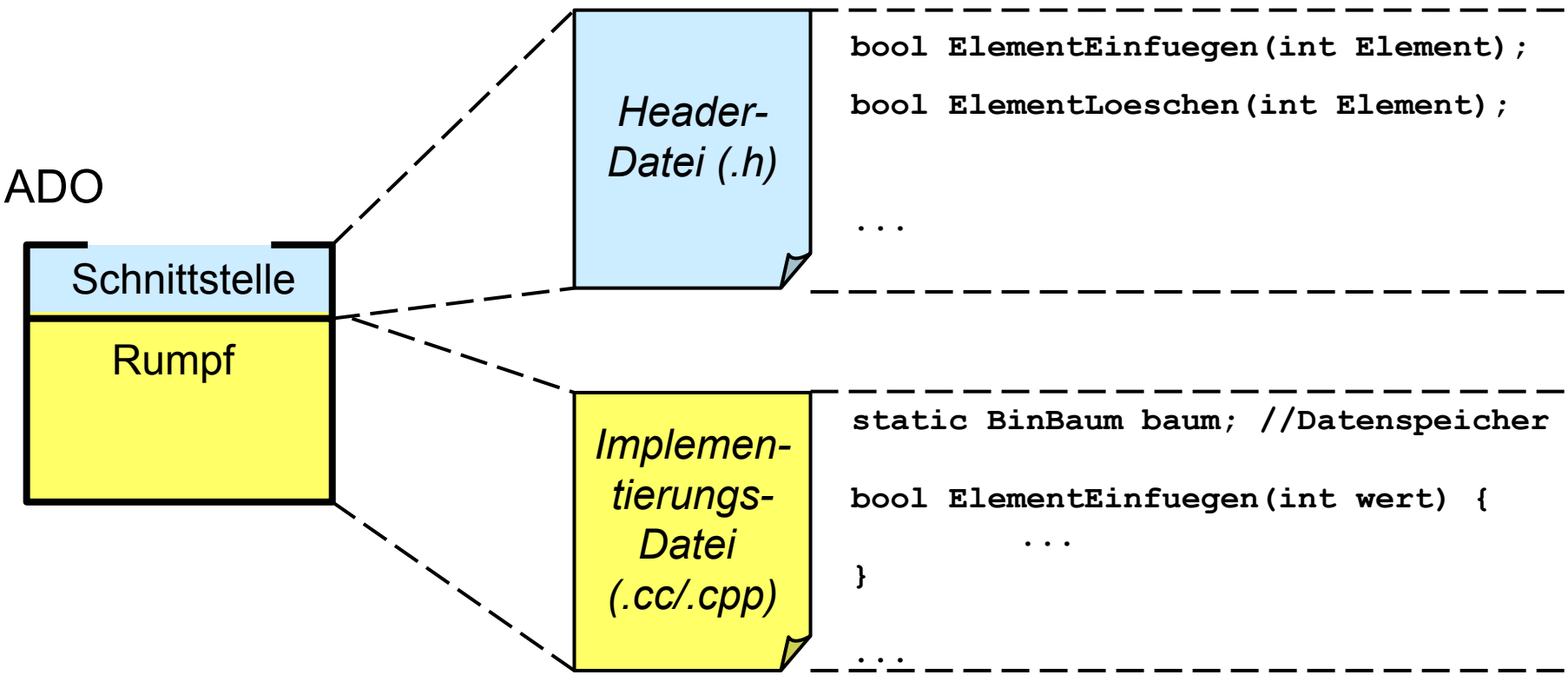
DA

- ADO : **A**bstraktes **D**aten-**O**bjekt
  - Datenabstraktion
  - Exakt ein Objekt zur Laufzeit
  - Alle Operationen auf diesem Objekt
  - Allokation/Erzeugung und De-Allokation/Zerstörung durch Laufzeitsystem
- ADT : **A**bstrakter **D**aten-**T**yp
  - Datenabstraktion
  - Schablone, beliebig viele Instanzen zur Laufzeit erzeugbar
  - Instanz bei jeder Operation angeben
  - Allokation/Erzeugung und Deallokation/Zerstörung in Verantwortung des Programmierers

FA

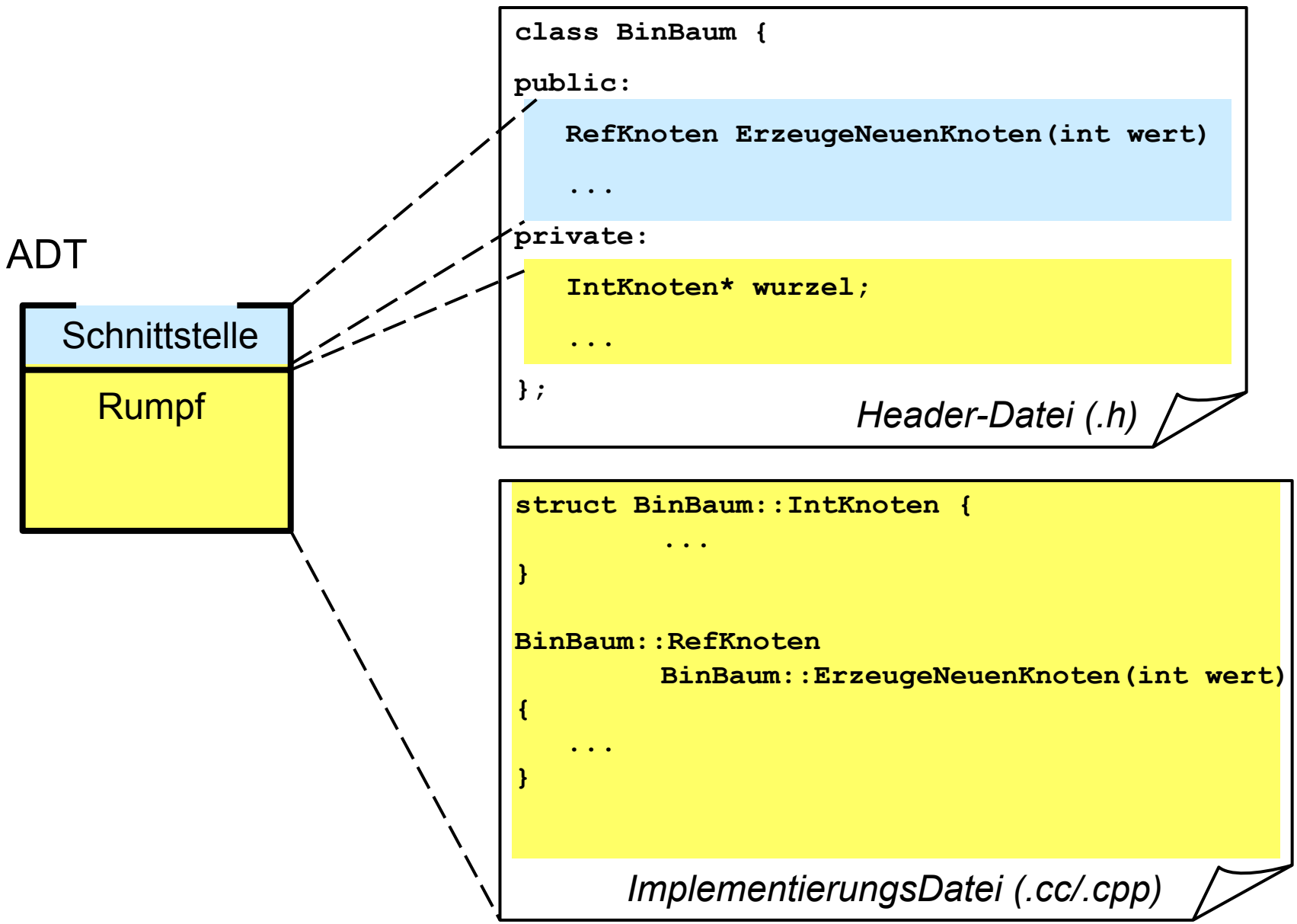
- FM : **F**unktionales **M**odul
  - Funktionale Abstraktion
  - Transformierendes Modul → Funktionsbibliothek
  - Kein "Gedächtnis" → Keine Seiteneffekte

Realisierung der Modularten (ADO, FM)



FM: ohne Datenspeicher, sonst gleiche Realisierung

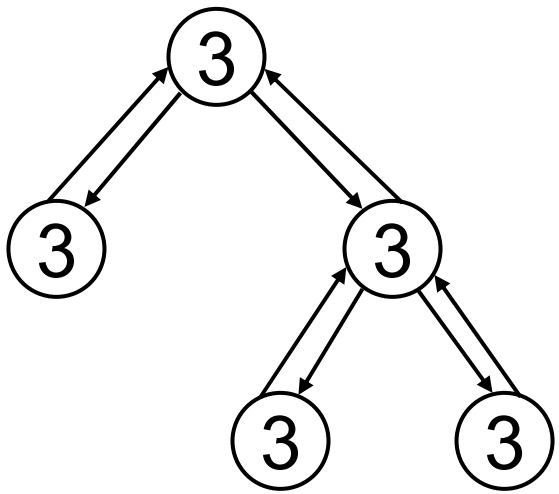
Realisierung der Modularten (ADT)



Definieren von Datenstrukturen

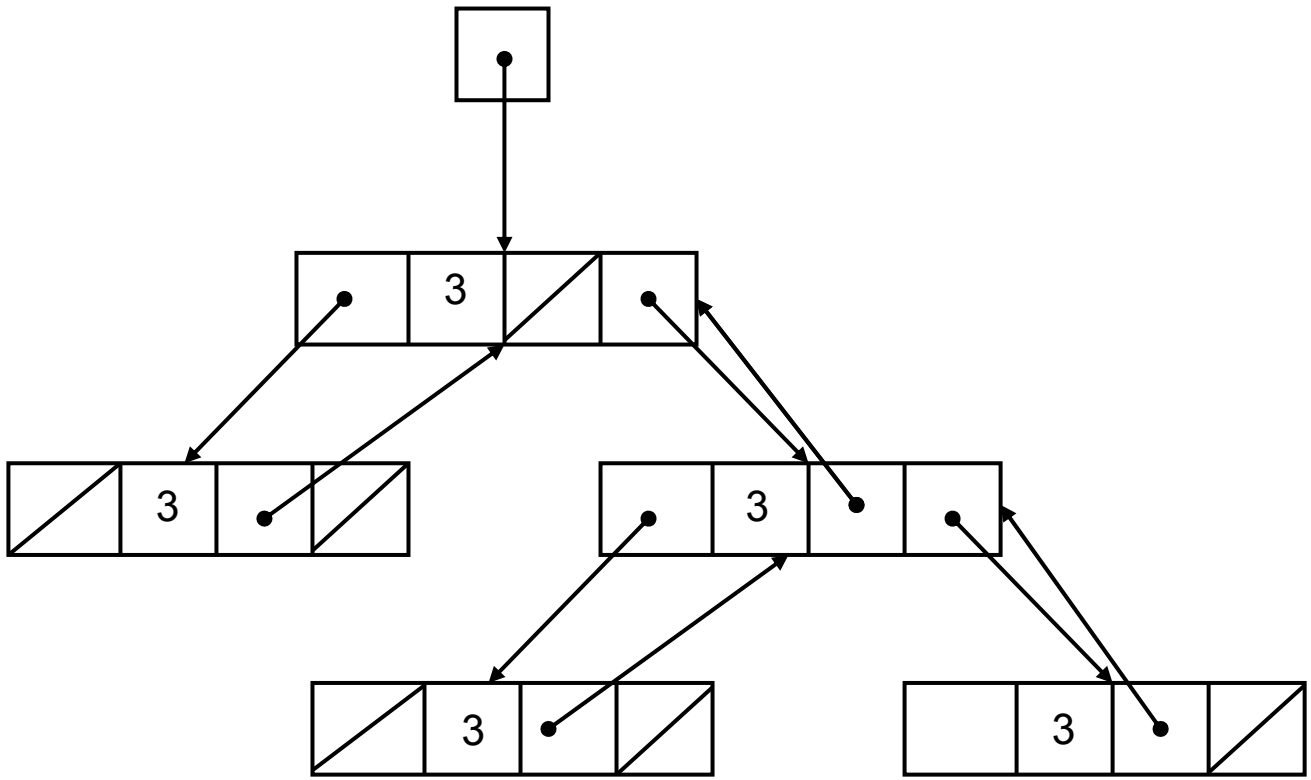
Ziel: Binärer Baum mit Zeiger auf Vater

Idee:



Definieren von Datenstrukturen

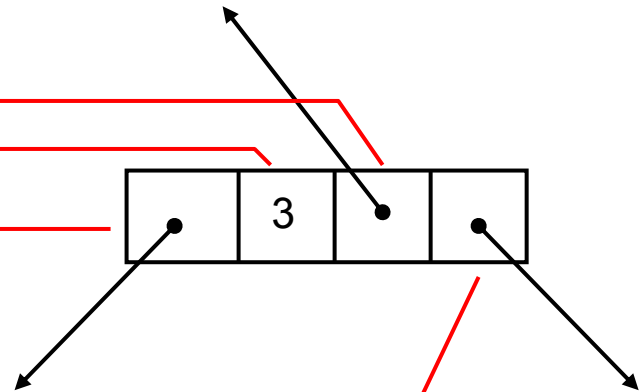
Konkreteres Modell:



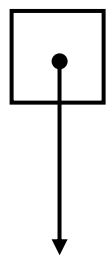
Definieren von Datenstrukturen

Umsetzung in Programmiersprache C++:

```
struct BinBaum::IntKnoten {
        IntKnoten* vater;
        int inhalt;
        IntKnoten* linkesKind;
        IntKnoten* rechtesKind;
};
```




```
IntKnoten* wurzel;
```



ADT-Umsetzung: Header-Datei

```
// BinBaum.h: Schnittstelle für den ADT BinBaum.
class BinBaum { // Der außen verwendete opake Datentyp für Bäume

private:
   struct IntKnoten;
   IntKnoten* m_Wurzel;

public:
   typedef IntKnoten *RefKnoten;

   BinBaum(); // Konstruktor

   RefKnoten GibWurzel();
   RefKnoten GibLinkesKind(RefKnoten vater);
   RefKnoten GibRechtesKind(RefKnoten vater);
   RefKnoten GibVater(RefKnoten aktuellerKnoten);

   RefKnoten ErzeugeNeuenKnoten(int wert);
   void SetzeWurzel( RefKnoten neueWurzel );
   void EinhaengenKnotenLinks( RefKnoten vater, RefKnoten neuesKind );
   void EinhaengenKnotenRechts( RefKnoten vater, RefKnoten neuesKind );

   void SetzeWert( RefKnoten zuAendernderKnoten, int wert );
   int GibWert( RefKnoten zuLesenderKnoten );

   ...

};
```

private Datenstruktur zum Aufnehmen eines Knotens und Wurzel des Baumes

zusätzlicher "opaker" Datentyp zur gezielten Identifikation einzelner Knoten

Navigation

Manipulation

Zugriff auf Inhalt

Schnittstelle

ADT-Umsetzung: .cpp/.cc-Datei

```
#include "BinBaum.h"

struct BinBaum::IntKnoten {
         int inhalt;
         IntKnoten* vater;
         IntKnoten* linkesKind;
         IntKnoten* rechtesKind;
}

BinBaum::RefKnoten BinBaum::ErzeugeNeuenKnoten(int wert)
{
   IntKnoten *knoten = new IntKnoten;

   knoten->inhalt = wert;
   knoten->vater = NULL;
   knoten->linkesKind = NULL;
   knoten->rechtesKind = NULL;

   return knoten;
}

BinBaum::RefKnoten BinBaum::GibLinkesKind(RefKnoten vater)
{
   return vater->linkesKind;
}
```

"Geheime" Definition von IntKnoten

Methode gehört zur Klasse **BinBaum**

Knoten erzeugen

Initialisieren, vgl. Definition der Struktur

Zeiger auf Knoten ist **RefKnoten**

aussen ist Struktur von **vater** nicht zugreifbar

intern darf die Struktur zugegriffen werden

ADT-Umsetzung: Verwendung

```
#include "BinBaum.h"

BinBaum baum;

void main() {
   BinBaum::RefKnoten kn=baum.ErzeugeNeuenKnoten(42);
   baum.SetzeWurzel(kn);
   ...
   kn=baum.GibLinkesKind(baum.GibWurzel());
   cout << baum.GibWert(kn);
}
```

Instanz des ADTs (der Klasse)

Aufruf von Methoden der Instanz

Anmerkung:

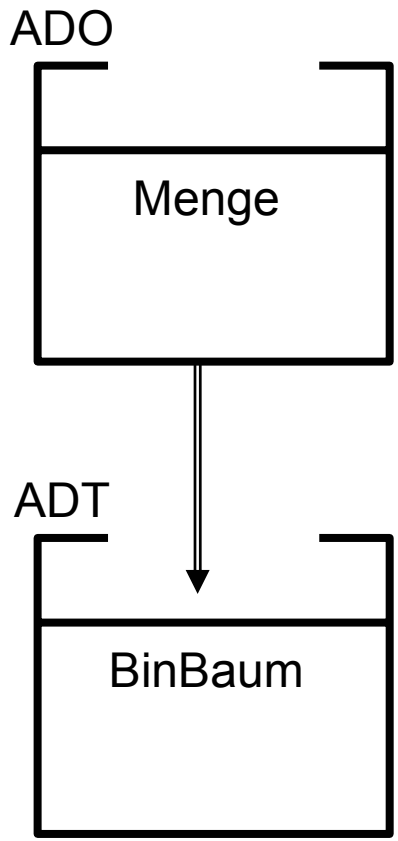
ADT in der Klausur immer als Klasse

in der Praxis (und auch in Vorlesung und Übung) auch anders

ADO-Umsetzung: Header Datei

Bei ADO: nur öffentliche Schnittstelle = Header-Datei

```
//Menge.h
//Realisierung einer Menge basierend auf einem Binärbaum
//Fuegt das Element item in die Menge ein
//Rueckgabewert: true, wenn erfolgreich
//                false, wenn item schon Element der Menge
bool ElementEinfuegen(int Element);

//Entfernt das Element item aus der Menge
//Rueckgabewert: true, wenn erfolgreich
//                false, wenn item nicht Element der Menge
bool ElementLoeschen(int Element);

//Testet, ob item in der Menge enthalten ist
//Rueckgabewert: true, wenn ja
//                false, sonst
bool istElement(int Element);

//Testet, ob die Menge leer ist
//Rueckgabewert: true, wenn ja
                 false, sonst
bool istLeer();
```



ADO-Umsetzung: .cpp/.cc-Datei

private Deklarationen (static)

```
#include "Menge.h"
#include "BinBaum.h"
static BinBaum baum;

bool ElementEinfuegen(int wert) {
   BinBaum::RefKnoten knoten;
   BinBaum::RefKnoten neuerKnoten;

   if (baum.GibWurzel()==NULL) {
      //ist der Baum leer?
      neuerKnoten=baum.ErzeugeNeuenKnoten(wert);
      baum.SetzeWurzel(neuerKnoten);
      return true;
   } else {
   ...
   }
   return false;
}

...
```

Implementierung von Funktionen

ADO-Umsetzung (mit Typ): Header-Datei

```
// BinBaum.h: Schnittstelle für den ADO BinBaum.

struct IntKnoten;
typedef IntKnoten *RefKnoten;

RefKnoten GibWurzel();
RefKnoten GibLinkesKind(RefKnoten vater);
RefKnoten GibRechtesKind(RefKnoten vater);
RefKnoten GibVater(RefKnoten aktuellerKnoten);

RefKnoten ErzeugeNeuenKnoten(int wert);
void SetzeWurzel( RefKnoten neueWurzel );
void EinhaengenKnotenLinks( RefKnoten vater, RefKnoten neuesKind );
void EinhaengenKnotenRechts( RefKnoten vater, RefKnoten neuesKind );

void SetzeWert( RefKnoten zuAenderndenKnoten, int wert );
int GibWert( RefKnoten zuLesendenKnoten );

...
```

Vorabdeklaration des Privaten Typs, Definition folgt im Rumpf

Verwender kann diesen Typ zum gezielten Zugriff auf einzelne Knoten nutzen

Methoden-Deklarationen

Auch ein ADO kann einen Typ exportieren:

- Die eigentliche Datenstruktur (der Baum) existiert nur ein mal => ADO.
- Zusätzlich gibt es einen weiteren Typ zur Identifikation von Knoten.

ADO-Umsetzung(mit Typ): .cpp/.cc-Datei

```
#include "BinBaum.h"

struct IntKnoten {
    int inhalt;
    IntKnoten *vater, *linkesKind, *rechtesKind;
};

static IntKnoten* m_Wurzel;


RefKnoten ErzeugeNeuenKnoten(int wert){
   IntKnoten *knoten = new IntKnoten;

   knoten->inhalt = wert;
   knoten->vater = NULL;
   knoten->linkesKind = NULL;
   knoten->rechtesKind = NULL;

   return knoten;
}
...
```

Vollständige Definition des Typs

Wurzel des Baumes, private Variable

Methoden-Implementierungen

ADO-Umsetzung: Verwendung

```
#include "BinBaum.h"

void main() {
   RefKnoten kn=ErzeugeNeuenKnoten(42);
   SetzeWurzel(kn);
   ...
   kn=GibLinkesKind(GibWurzel());
   cout << GibWert(kn);
}
```

Es muss keine Instanz angelegt werden, da genau eine Instanz existiert

Daher muss auch keine Instanz beim Zugriff angegeben werden

Der Typ **`RefKnoten`** wird zum Zugriff auf einzelne Knoten genutzt

Funktionales Modul (FM) - Umsetzung

```
//trig.h

double sinus(double x);
double tan(double x);
double cos(double x);
...
```

Header-Datei=Schnittstelle

```
#include "trig.h"

double sinus(double x) {
    ...
}

double tan(double x) {
    ...
double cos(double x) {
    ...
}
```

.cpp/.cc-Datei=Rumpf

Klausurtermin

Donnerstag, 05.08.2004
14:00 Uhr
und
Dienstag, 21.09.2004
14:00 Uhr

Hörsaalverteilung wird an den Lehrstühlen ausgehängt

Viel Erfolg!