technische universität
dortmund

Einführung in die Programmierung

Wintersemester 2020/21

Klausurvorbereitung

M.Sc. Roman Kalkreuth
Lehrstuhl für Algorithm Engineering (LS11)
Fakultät für Informatik

Klausurvorbereitung

Inhalt

- Zeiger
- Klassen/Vererbung/virtuelle Methoden
- Datenstrukturen

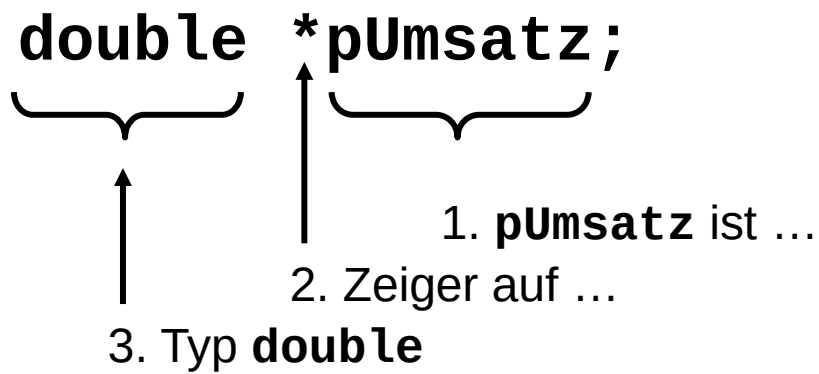
Zwei Operatoren: * und &

- *-Operator:
  - mit Datentyp: Erzeugung eines Zeigers `double *pUmsatz;`
  - mit Variable: Inhalt des Ortes, an den Zeiger zeigt `*pUmsatz = 10.24;`
- &-Operator:

  ermittelt Adresse des Datenobjektes `pUmsatz = &Umsatz;`

Wie interpretiert man Datendefinition richtig?

Man lese **von rechts nach links**!

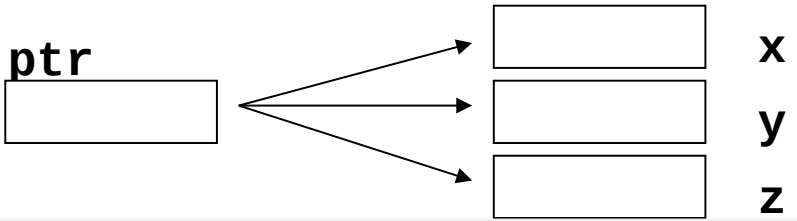


Beispiele:

```
double a = 4.0, b = 5.0, c;
c = a + b;
double *pa = &a, *pb = &b, *pc = &c;
*pc = *pa + *pb;
```

```
double x = 10.;
double y = *&x;
```

Unterscheidung



- **Veränderliche Zeiger**

```
double x = 2.0, y = 3.0, z = 7.0, s = 0.0, *ptr;
ptr = &x;
s += *ptr;
ptr = &y;
s += *ptr;
ptr = &z;
s += *ptr;
```

ptr nimmt nacheinander verschiedene Werte (Adressen) an,
s hat am Ende den Wert 12.0

Zeigerarithmetik

Achtung:

```
T val;
T *ptr = &val;
ptr = ptr + 2;
```

In der letzten Zeile werden **nicht** 2 Bytes zu **`ptr`** hinzugezählt,

sondern **2 mal die Speichergröße des Typs `T`**.

Das wird auch dann durchgeführt, wenn **`ptr`** nicht auf Array zeigt.

Zeigerarithmetik (Beispiele)

```
int a[] = { 100, 110, 120, 130 }, *pa, sum = 0;
pa = &a[0];
sum += *pa + *(pa + 1) + *(pa + 2) + *(pa + 3);
```

```
struct KundeT {
  double umsatz;
  float skonto;
};
KundeT Kunde[5], *pKunde;
pKunde = &Kunde[0];
int i = 3;
*pKunde = *(pKunde + i);
```

Größe des Datentyps **KundeT**:

8 + 4 = 12 Byte

Sei **pKunde == 10000**

Dann **(pKunde + i) == 10036**

prozedural

```
struct Punkt {
  double x, y;
};
void setzeX(Punkt &p, double w);
void setzeY(Punkt &p, double w);
double leseX(Punkt const &p);
double leseY(Punkt const &p);
```

objektorientiert

```
class Punkt {
  double x, y;
public:
  void setzeX(double w);
  void setzeY(double w);
  double leseX();
  double leseY();
};
```

⇒ Schlüsselwort **`public`** : alles Nachfolgende ist **öffentlich zugänglich**

```
struct Punkt {
  double x, y;
};
```

```
void verschiebe(Punkt &p,
                double dx, double dy);
bool gleich(Punkt &a, Punkt& b);
double norm(Punkt &a);
```

```
class Punkt {
private:
  double x, y;
public:
  void   setzeX(double w);
  void   setzeY(double w);
  double leseX();
  double leseY();
  void   verschiebe(double dx, double dy);
  bool   gleich(Punkt const &p);
  double norm();
};
```

Methoden

Klasse = Beschreibung von **Eigenschaften** und **Operationen:**

- Eine Klasse ist also die Beschreibung des **Bauplans** (Konstruktionsvorschrift) für konkrete (mit Werten belegte) **Objekte**
- Eine Klasse ist **nicht** das Objekt selbst
- Ein Objekt ist eine **Instanz** / Ausprägung einer Klasse

Objekt = Zusammenfassung von Daten / Eigenschaften und Operationen

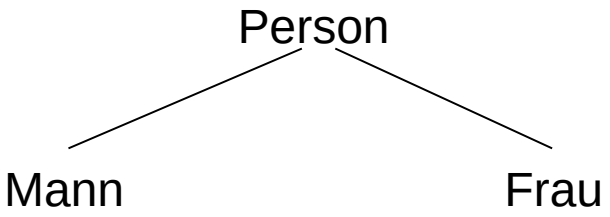
Zugriff auf Daten **nur über Operationen** der Klasse;
man sagt auch: *„dem Objekt wird eine Nachricht geschickt“*

Objektname.Nachricht(Daten)

Methode = Operation, die sich auf ein Objekt einer Klasse anwenden lässt

(Synonyme: Element- oder Klassenfunktion)

Beispiel Klassenhierachie:

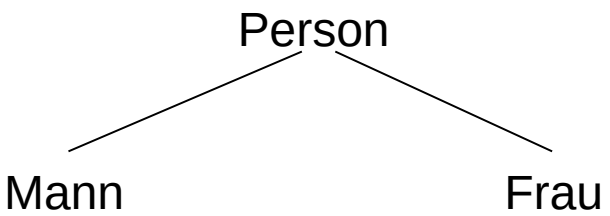


Die abgeleiteten Klassen **Mann** und **Frau** enthalten alle Attribute und Methoden, die geschlechtsspezifisch sind.

```
class Mann : public Person {
private:
  Frau *Ehefrau;
public:
  Mann(char const *vn);
  Mann(Person *p);
  void NimmZurFrau(Frau *f);
  Frau *EhemannVon();
class Frau : public Person {
private:
  Mann *Ehemann;
public:
  Frau(char const *vn);
  Frau(Person *p);
  void NimmZumMann(Mann *m);
  Mann *EhefrauVon();
};
```

Beispiel:

Einfache Klassenhierarchie

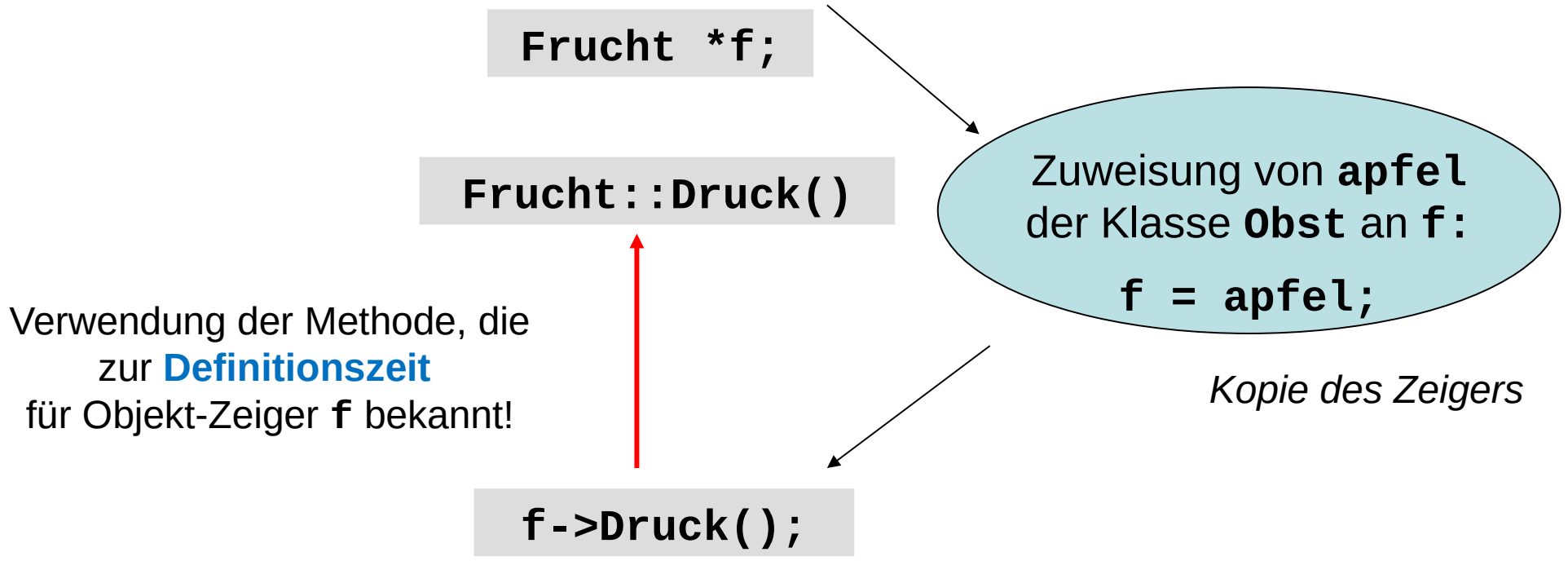


Klasse **`Person`** enthält alle Attribute und Methoden, die geschlechtsunspezifisch sind.

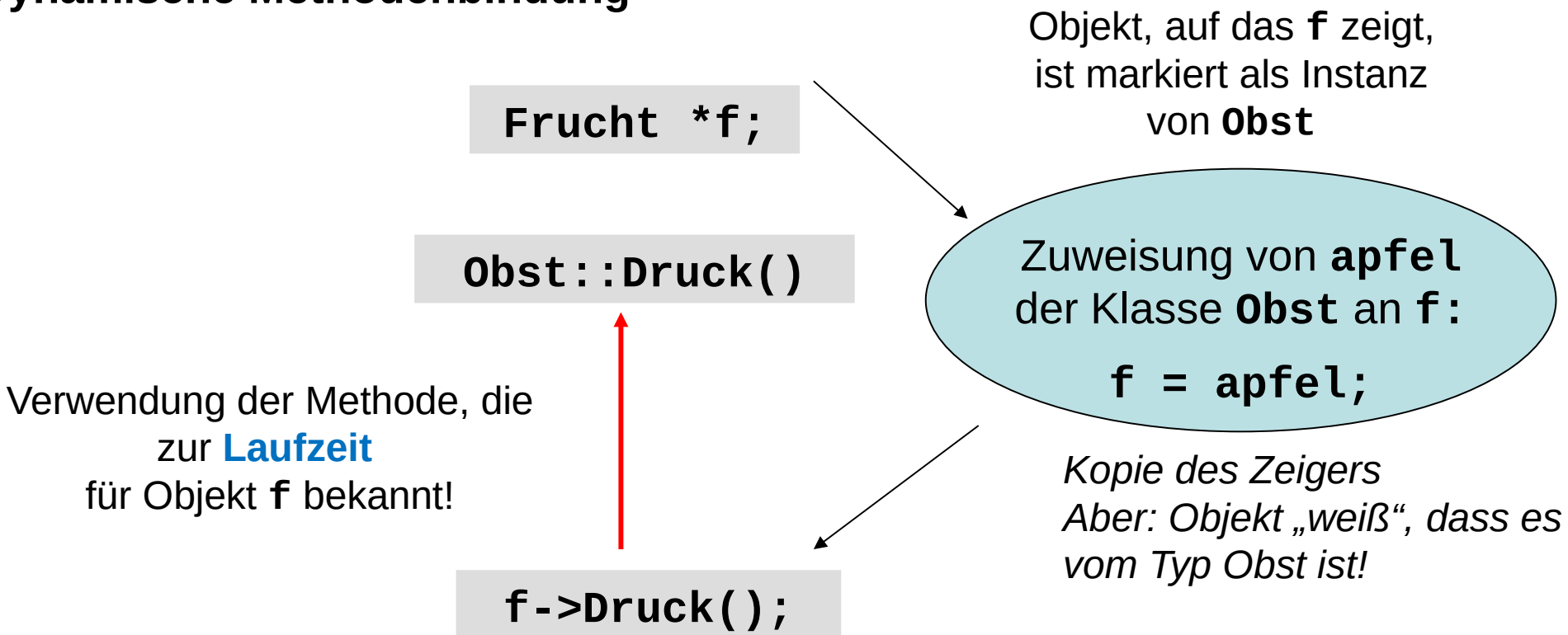
```
class Person {
private:
  KString *Vorname;
  Frau    *Mutter;
  Mann    *Vater;
public:
  Person(char const *vorname);
  Person(KString *vorname);

  char *Name();
  void SetzeVater(Mann *m);
  void SetzeMutter(Frau *f);
  void Druck(char const *s);
  ~Person();
};
```

Statische Methodenbindung



Dynamische Methodenbindung



Version mit virtuellen Funktionen

```
class Frucht {
protected:
  string dieFrucht;
public:
  Frucht(char const *name);
  Frucht(string &name);
  virtual void Druck();
};
```

Ansonsten keine Änderungen im Code der konventionellen Version!

Kennzeichnung als **virtuelle Methode**:

Instanzen von abgeleiteten Klassen suchen dynamisch die entsprechende Methode aus.

Konsequenzen: Testprogramm mit virtuellen Methoden (nur 2. Teil)

```
  Frucht *f = new Frucht("Frucht");
  f->Druck();

  f = apfel;  // jedes Obst ist auch Frucht
  f->Druck();

  Obst *o = new Obst("Obst");
  o->Druck();

  o = banane; // Suedfrucht ist auch Obst
  o->Druck();
}
```

2. Teil

Ausgabe:
(dynamisch, also mit **`virtual`**)

```
(F) Frucht
(O) Apfel
(O) Obst
(S) Banane
```

Ausgabe:
(statisch, also ohne **`virtual`**)

```
(F) Frucht
(F) Apfel
(O) Obst
(O) Banane
```

Rein virtuelle Methoden

```
class AusgabeGeraet {
protected:
  bool KannFarben;
  Data data;
public:
  virtual void Farbdruck() = 0;
  void Drucke();
};
```

abstrakte Klasse

```
void AusgabeGeraet::Drucke() {
  if (KannFarben) Farbdruck();
  else cout << data;
}
```

Man kann rein virtuelle Methode verwenden, ohne dass Code vorhanden ist!

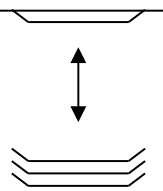
Lineare Datenstrukturen: Keller bzw. **Stapel** (*engl. stack*)

create	:		→	Stapel
push	:	Stapel x T	→	Stapel
pop	:	Stapel	→	Stapel
top	:	Stapel	→	T
empty	:	Stapel	→	bool

empty(create) = true
empty(push(k, x)) = false
pop(push(k, x)) = k
top(push(k, x)) = x

Aufräumen:
Kiste in den Keller,
oben auf Haufen.

Etwas aus Keller holen:
Zuerst **oberste** Kiste, weil oben auf Haufen.



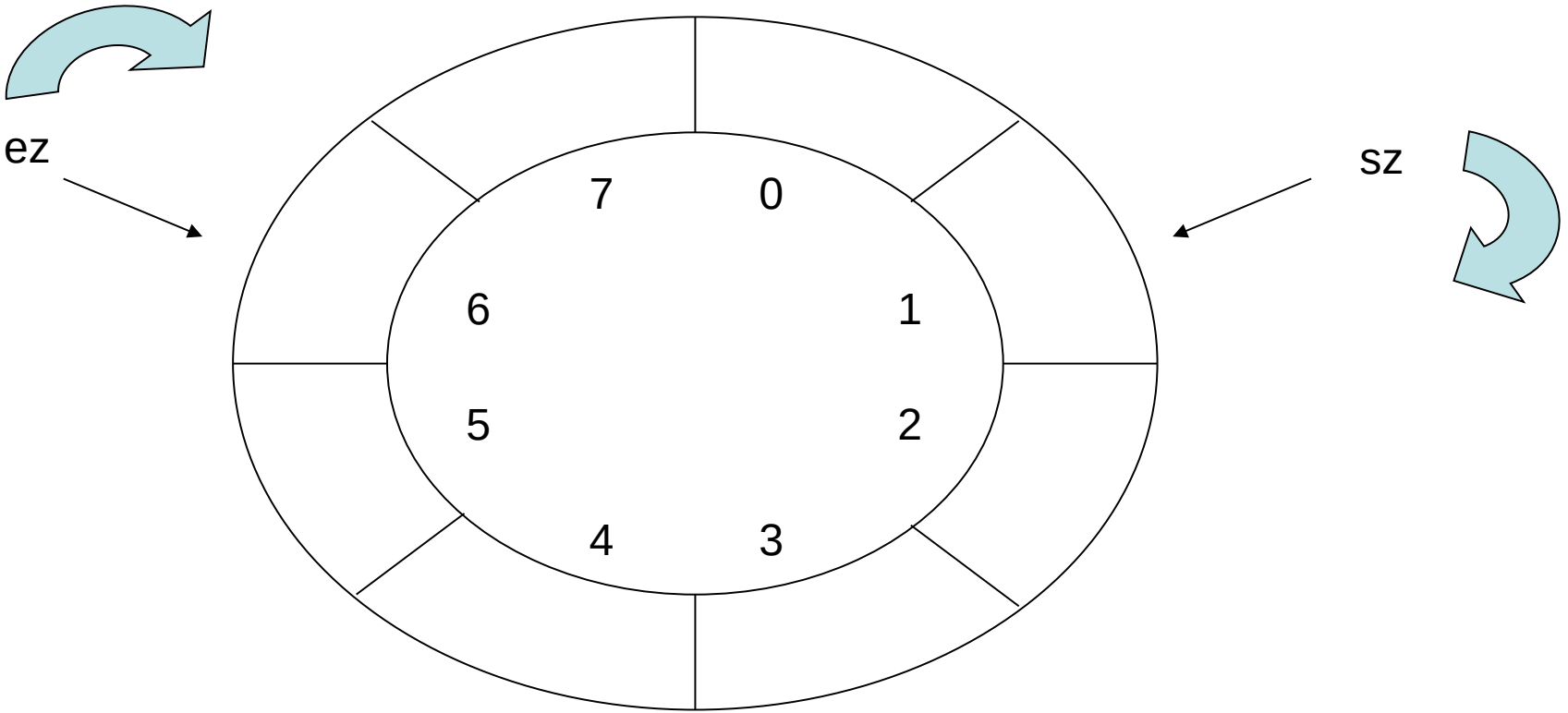
Stapel
Teller

LIFO:
Last in, first out.

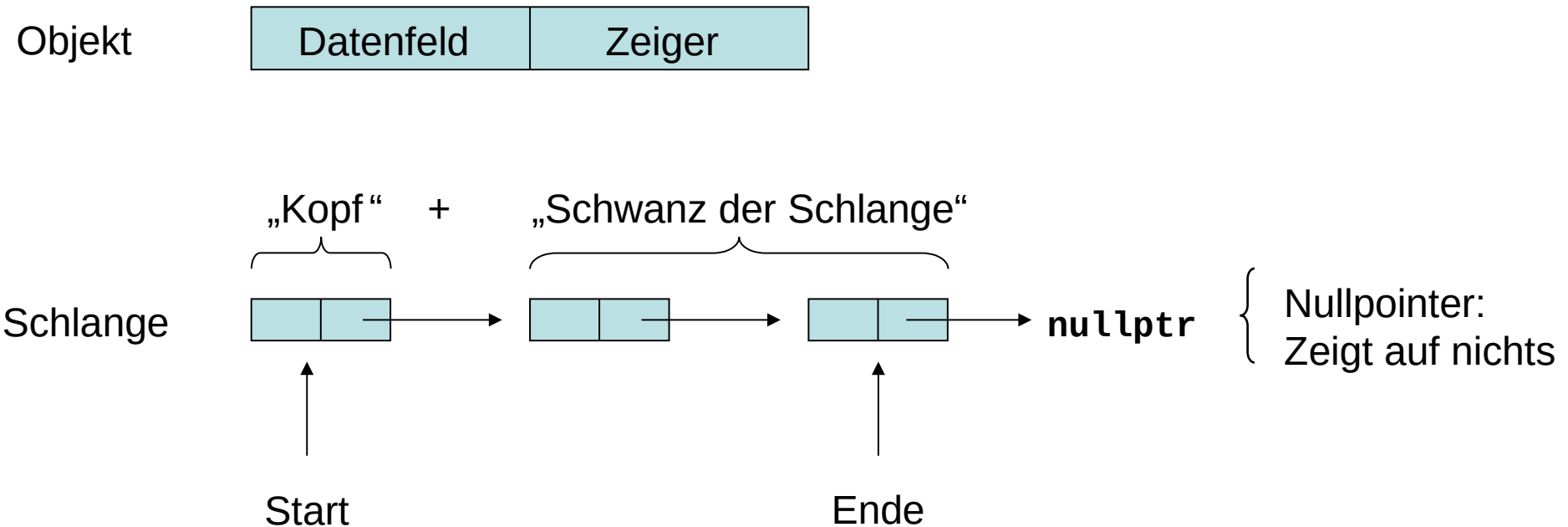
Klassendefinition: (Version 1)

```
template<typename T>
class Stapel {
public:
  Stapel();          // Konstruktor
  void push(T &x);   // Element auf den Stapel legen
  void pop();        // oberstes Element entfernen
  T top();           // oberstes Element ansehen
  bool empty();      // Stapel leer?
private:
  static unsigned int const maxSize = 100;
  int sz;            // Stapelzeiger
  T data[maxSize];   // Speichervorrat für Nutzdaten
};
```

Idee: Array zum Kreis machen; zusätzlich Anfang/Start markieren (sz)



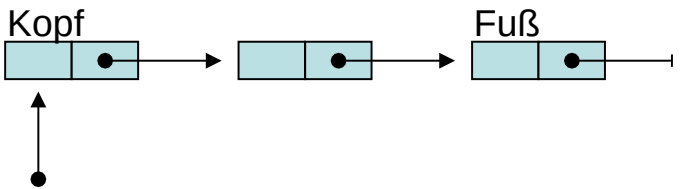
Vorüberlegungen für ADT Schlange mit dynamischem Speicher:



Klassendefinition: (Version 3; mit dynamischem Speicher)

```
template<typename T>
class Schlange {
public:
  Schlange();                  // Konstruktor
  void enq(T &x);
  void deq();
  T front();
  bool empty();
  void clear();                // löscht alle Einträge
  ~Schlange();                 // Destruktor
private:
  struct Objekt {              // interner Datentyp
    Objekt *tail;              // Zeiger auf Schlangenschwanz
    T       data;              // Datenfeld
  } *sz, *ez;                  // Zeiger auf Start + Ende
  void error(char const *info); // für Fehlermeldungen
};
```

ADT Liste (1. Version)



Liste wird nur durch **einen Zeiger** auf ihren Listenkopf repräsentiert

Operationen:

create	:		→ Liste	
empty	:	Liste	→ bool	
append	:	T x Liste	→ Liste	hängt am Ende an
prepend	:	T x Liste	→ Liste	vor Kopf einfügen
clear	:		→ Liste	
is_elem	:	T x Liste	→ bool	ist Element enthalten?

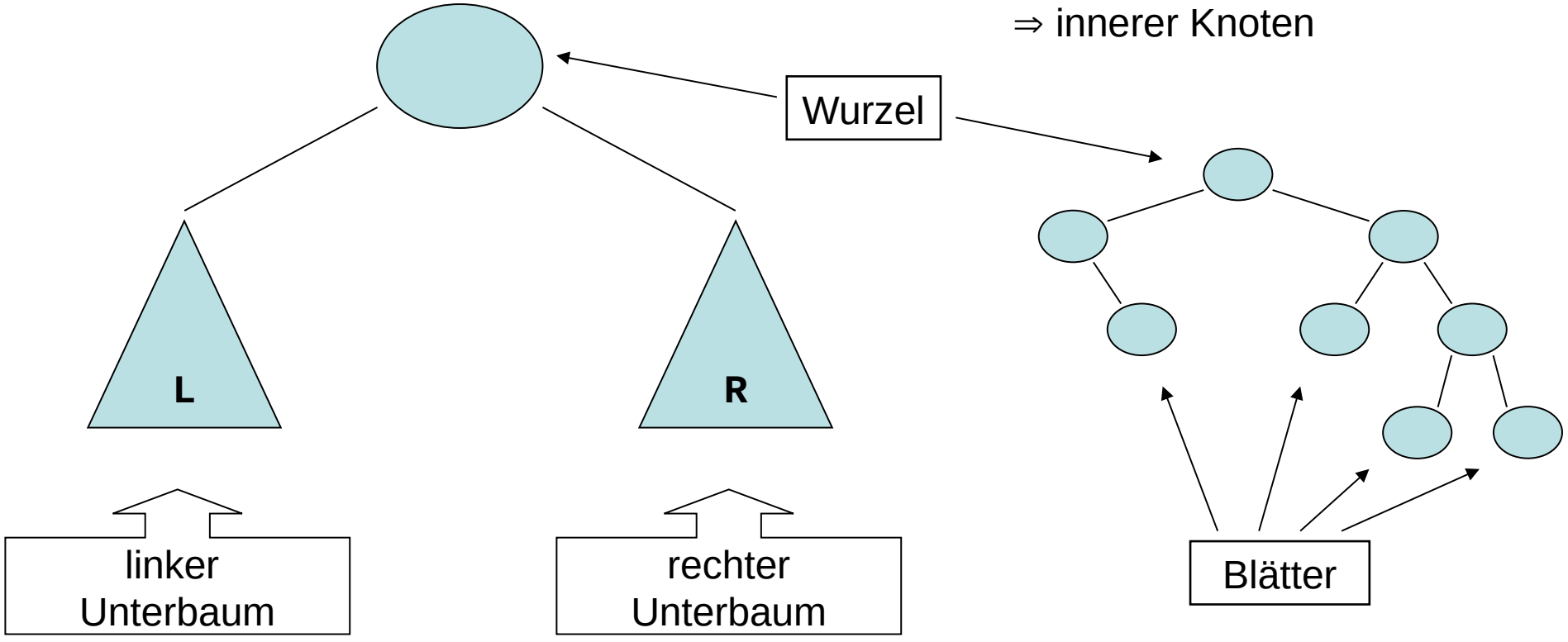
ADT Liste (1. Version)

```
template<typename T>
class Liste {
public:
  Liste();                      // Konstruktor
  Liste(const Liste<T>& liste);// Kopierkonstruktor
  void append(const T& x);  // hängt hinten an
  void prepend(const T& x); // fügt vorne ein
  bool empty();             // Liste leer?
  bool is_elem(const T& x); // ist Element x in Liste?
  void clear();             // Liste leeren
  ~Liste();                 // Destruktor
private:
  struct Objekt {           // privater Datentyp
    T       data;           // Nutzdaten
    Objekt *next;           // Zeiger auf nächstes Objekt
  } *sz;                    // Startzeiger auf Listenkopf
  void clear(Objekt *obj);  // Hilfsmethode zum Leeren
```

ADT Binärer Suchbaum: Terminologie

keine Wurzel und kein Blatt

⇒ innerer Knoten



ADT Binärer Suchbaum: Klassendefinition

```
template<typename T>
class BinTree {
private:
  struct Node {
    T data;
    Node *left, *right;
  } *root;
  Node *insert(Node *node, T key);
  bool isElem(Node *node, T key);
  void clear(Node *node);
public:
  BinTree()         { root = nullptr; }
  void insert(T x) { root = insert(root, x); }
  bool isElem(T x) { return isElem(root, x); }
  void clear()      { clear(root); root = nullptr; }
  ~BinTree()        { clear(); }
};
```

leerer Unterbaum → Nullzeiger

ADT Binärer Suchbaum: Element suchen

```
template<typename T>
bool BinTree<T>::isElem(Node *node, T key) {
  if (node == nullptr) return false;
  if (node->data == key) return true;
  if (node->data < key) return isElem(node->right, key);
  return isElem(node->left, key);
}
```

Rekursionsverankerung (Abbruchbedingung)

Rekursive Suche:

Falls kein Erfolg im aktuellen Knoten, dann Frage an den Unterbaum weiterreichen, der das Element enthalten müsste.

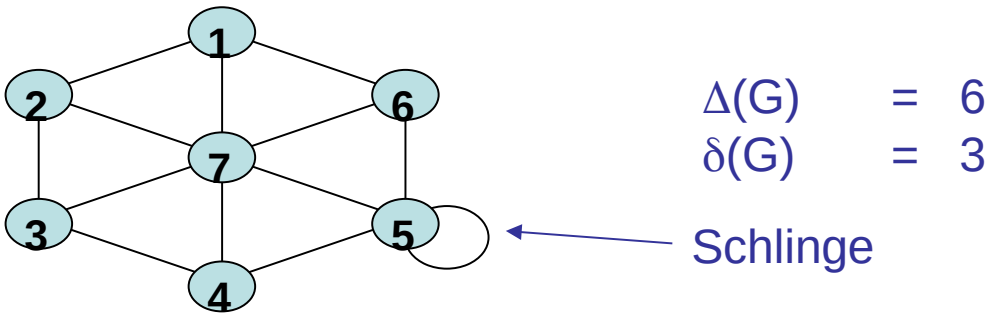
Falls Knoten Element enthält: Erfolg!

Falls Unterbaum leer, dann Element nicht vorhanden.

Definition

Ein **Graph** G = (V, E) besteht aus einer Menge von **Knoten** V („vertex, pl. vertices“)

und einer Menge von **Kanten** E („edge, pl. edges“) mit E ⊆ V x V.



Eine Kante (u, v) heißt **Schlinge** (“loop“), wenn u = v.

Der **Grad** („degree“) eines Knotens v ∈ V ist die Anzahl der zu ihm inzidenten Kanten: deg(v) = | { (a, b) ∈ E : a = v oder b = v } | . Schlingen zählen doppelt.

Maxgrad von G ist Δ(G) = max { deg(v) : v ∈ V }

Mingrad von G ist δ(G) = min { deg(v) : v ∈ V }

Mögliche Funktionalität

```
typedef unsigned int uint;
```

→ **typedef** Datentyp TypName;

```
class Graph {
public:
  Graph(uint NoOfNodes);
  void addEdge(uint Node1, uint Node2);
  bool hasEdge(uint Node1, uint Node2);
  uint noOfEdges();
  uint noOfNodes();
  void printGraph();
  ~Graph();
private:
  uint mNoOfNodes;
  Liste<uint> *mAdjList;
};
```

mAdjList: Array von Zeigern auf **Liste<uint>**