

# Einführung in die Programmierung

Wintersemester 2009/10

Prof. Dr. Günter Rudolph

Lehrstuhl für Algorithm Engineering

Fakultät für Informatik

TU Dortmund

## Inhalt

- Definition: Abstrakter Datentyp (ADT)
- ADT Stapel
- ADT Schlange
- ADT Liste
- ADT Binärer Suchbaum
- ADT Graph
- Exkurse:
  - Einfache Dateibehandlung
  - C++ Strings

**Definition:**

Abstrakter Datentyp (ADT) ist ein Tripel  $(T, F, A)$ , wobei

- T eine nicht leere Menge von Datenobjekten
- F eine Menge von Operationen,
- A eine nicht leere Menge von Axiomen,  
die die Bedeutung der Operationen erklären. ■


**Abstrakt?**

- Datenobjekte brauchen keine konkrete Darstellung (Verallgemeinerung).
- Die Wirkung der Operationen wird beschrieben,  
nicht deren algorithmische Ausprägung.  
→ „**WAS, nicht WIE!**“

**Beispiel: ADT bool**

F: Operationen


true	:		→ bool
false	:		→ bool
not	:	bool	→ bool
and	:	bool x bool	→ bool
or	:	bool x bool	→ bool



Festlegung, welche  
Methoden es gibt

A: Axiome

not(false)	=	true
not(true)	=	false
and(false, false)	=	false
and(false, true)	=	false
and(true, false)	=	false
and(true, true)	=	true
or(x, y)	=	not(and(not(x), not(y)))



Festlegung, was die  
Methoden bewirken

### Eigenschaften

- Wenn man ADT kennt, dann kann man ihn überall verwenden.
- Implementierung der Funktionen für Benutzer nicht von Bedeutung.
- Trennung von Spezifikation und Implementierung.
- Ermöglicht späteren Austausch der Implementierung, ohne dass sich der Ablauf anderer Programme, die ihn benutzen, ändert!

Nur Operationen geben Zugriff auf Daten!

→ Stichwort: Information Hiding!

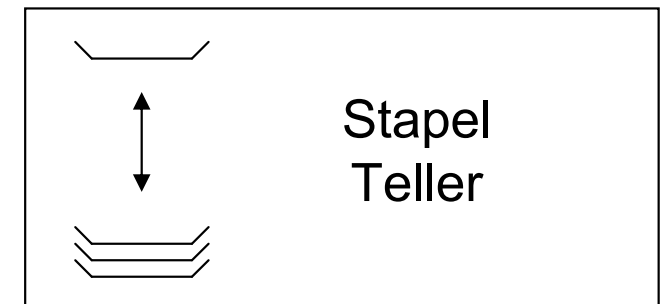
### Lineare Datenstrukturen: Keller bzw. Stapel (*engl. stack*)

create :  $\rightarrow$  Stapel  
 push : Stapel  $\times$  T  $\rightarrow$  Stapel  
 pop : Stapel  $\rightarrow$  Stapel  
 top : Stapel  $\rightarrow$  T  
 empty : Stapel  $\rightarrow$  bool

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

Aufräumen:  
Kiste in Keller,  
oben auf Haufen.

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



LIFO:  
Last in, first out.

Klassendefinition: (Version 1)

```
typedef int 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
};
```

Implementierung: (Version 1)

```
Stapel::Stapel() {  
    sz = -1;  
}
```

**Idee:**  
unzulässiger Arrayindex  
kennzeichnet leeren Stapel

```
void Stapel::push(T &x) {  
    data[++sz] = x;  
}  
void Stapel::pop() {  
    sz--;  
}  
T Stapel::top() {  
    return data[sz];  
}  
bool Stapel::empty() {  
    return (sz == -1);  
}
```

**Probleme:**  
**Arraygrenzen!**





## Wann können Probleme auftreten?

Bei `pop`, falls Stapel leer ist:

→ Stackpointer wird -2, anschließendes `push` versucht auf `data[-1]` zu schreiben

Bei `top`, falls Stapel leer ist:

→ es wird undefinierter Wert von `data[-1]` zurückgegeben

Bei `push`, falls Stapel voll ist:

→ es wird versucht auf `data[maxSize]` zu schreiben

⇒ **diese Fälle müssen abgefangen werden! Fehlermeldung!**

```
void error(char *info) {  
    cerr << info << endl;  
    exit(1);  
}
```

gibt Fehlermeldung `info` aus und bricht das Programm durch `exit(1)` **sofort** ab und liefert den Wert des Arguments (hier: 1) an das Betriebssystem zurück

Klassendefinition: (Version 2; Ergänzungen in **rot**)

```
typedef int 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?
    bool full();      // Stapel voll?
private:
    static unsigned int const maxSize = 100;
    int sz;           // Stapelzeiger
    T data[maxSize]; // Speichervorrat für Nutzdaten
    void error(char *info); // Fehlermeldung + Abbruch
};
```

Implementierung: (Version 2, Änderungen und Zusätze in rot)

```
Stapel::Stapel() {
    sz = -1;
}
void Stapel::push(T &x) {
    if (full()) error("voll");
    data[++sz] = x;
}
void Stapel::pop() {
    if (empty()) error("leer");
    sz--;
}
```

```
T Stapel::top() {
    if (empty()) error("leer");
    return data[sz];
}
bool Stapel::empty() {
    return (sz == -1);
}
bool Stapel::full() {
    return (sz == maxSize - 1);
}
```

```
void Stapel::error(char* info) {
    cerr << info << endl;
    exit(1);
}
```

← private Methode:  
kann nur innerhalb der Klasse  
aufgerufen werden!

## Erster Test ...

```
#include <iostream>
#include "Stapel.h"
using namespace std;

int main() {
    Stapel s;
    for (int i = 0; i < 100; i++) s.push(i);
    cout << s.top() << endl;
    for (int i = 0; i < 90; i++) s.pop();
    while (!s.empty()) {
        cout << s.top() << endl;
        s.pop();
    }
    return 0;
}
```

**Ausgabe:** 99  
9  
8  
7  
6  
5  
4  
3  
2  
1  
0

### Lineare Datenstrukturen: Schlange (*engl. queue*)

create :  $\rightarrow$  Schlange  
 enq : Schlange  $\times$  T  $\rightarrow$  Schlange  
 deq : Schlange  $\rightarrow$  Schlange  
 front : Schlange  $\rightarrow$  T  
 empty : Schlange  $\rightarrow$  bool

empty(create) = true  
 empty(enq(s, x)) = false  
 deq(enq(s, x)) = empty(s) ? s : enq(deq(s), x)  
 front(enq(s, x)) = empty(s) ? x : front(s)

FIFO:  
First in, first out.

Schlange an der Supermarktkasse:  
Wenn Einkauf fertig, dann **hinten** anstellen. Der nächste Kunde an der Kasse steht ganz **vorne** in der Schlange.

Eingehende Aufträge werden „geparkt“, und dann nach und nach in der Reihenfolge des Eingangs abgearbeitet.

Klassendefinition: (Version 1; schon mit Fehlerbehandlung)

```
typedef int T;

class Schlange {
public:
    Schlange(); // Konstruktor
    void enq(T &x); // Element anhängen
    void deq(); // erstes Element entfernen
    T front(); // erstes Element ansehen
    bool empty(); // Schlange leer?
    bool full(); // Schlange voll?
private:
    static unsigned int const maxSize = 100;
    int ez; // Endezeiger
    T data[maxSize]; // Array für Nutzdaten
    void error(char *info); // Fehlermeldung
};
```

Implementierung: (Version 1; Fehler bei Arraygrenzen werden abgefangen)

```
Schlange::Schlange():ez(-1) {  
}  
T Schlange::front() {  
    if (empty()) error("leer");  
    return data[0];  
}  
bool Schlange::empty() {  
    return (ez == -1);  
}  
bool Schlange::full() {  
    return (ez == maxSize - 1);  
}
```

```
void Schlange::enq(T &x) {  
    if (full()) error("voll");  
    data[++ez] = x;  
}  
  
void Schlange::deq() {  
    if (empty()) error("leer");  
    for (int i = 0; i < ez; i++)  
        data[i] = data[i+1];  
    ez--;  
}
```

```
void Schlange::error(char *info) {  
    cerr << info << endl;  
    exit(1);  
}
```

← private Methode:  
kann nur innerhalb der Klasse  
aufgerufen werden!

## Erster Test ...

```
#include <iostream>
#include "Schlange.h"
using namespace std;

int main() {
    Schlange s;

    for (int i = 0; i < 100; i++) s.enq(i);
    cout << s.front() << endl;
    for (int i = 0; i < 90; i++) s.deq();
    while (!s.empty()) {
        cout << s.front() << endl;
        s.deq();
    }
    return 0;
}
```

**Ausgabe:** 0  
90  
91  
92  
93  
94  
95  
96  
97  
98  
99



Benutzer des (abstrakten) Datentyps `Schlange` wird feststellen, dass

1. fast alle Operationen schnell sind, aber
2. die Operation `deq` vergleichsweise langsam ist.

### Laufzeit / Effizienz der Operation `deq`

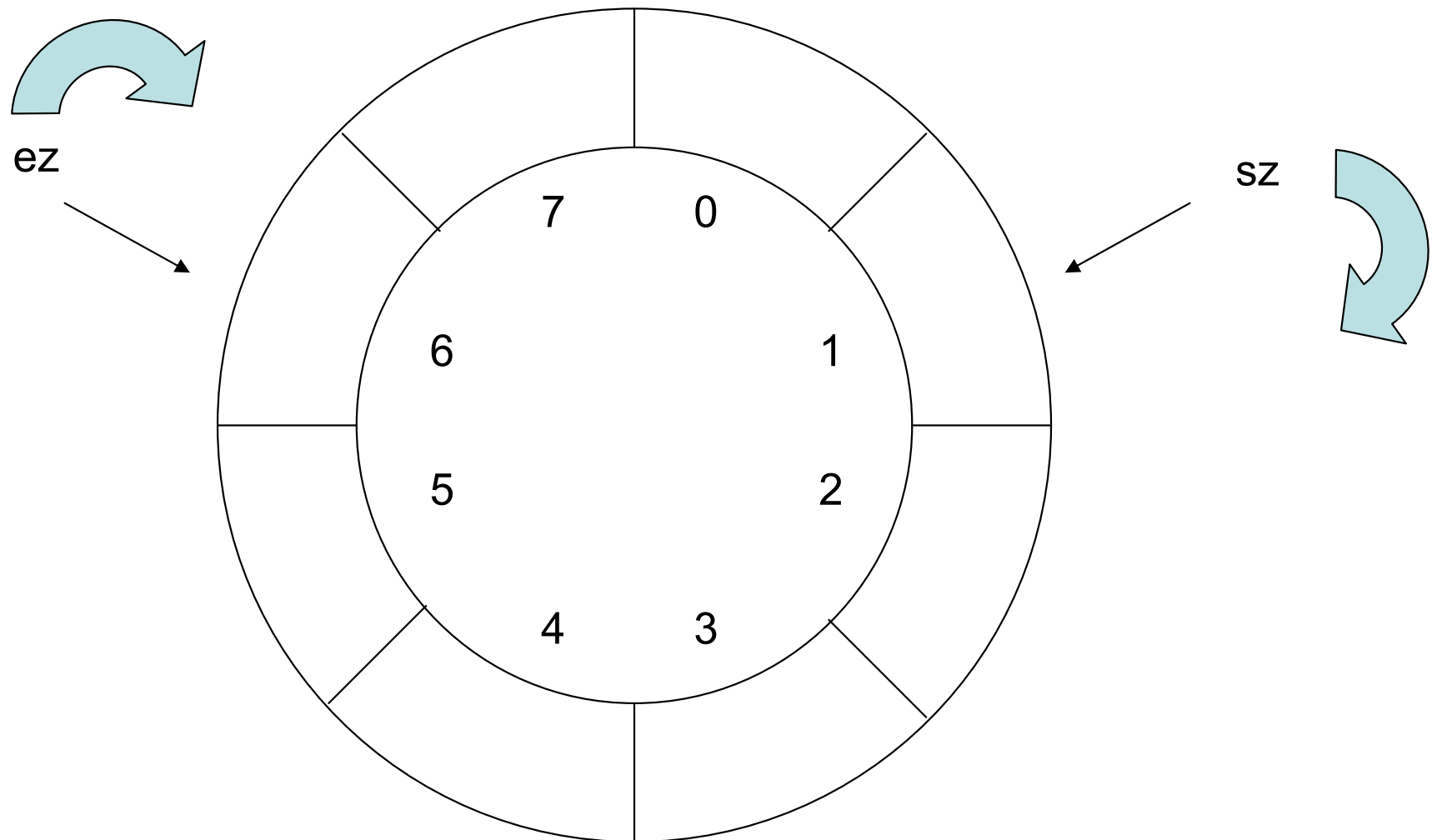
```
void Schlange::deq() {  
    if (empty()) error("leer");  
    for (int i = 0; i < ez; i++)  
        data[i] = data[i+1];  
    ez--;  
}
```

`ez` = Anzahl Elemente in Schlange

`ez` viele Datenverschiebungen

Worst case: (`maxSize` - 1) mal

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



Implementierung: (Version 2; mit Ringspeicher)

```
typedef int T;

class Schlange {
public:
    Schlange();
    void enq(T &x);
    void deq();
    T front();
    bool empty();
    bool full();
private:
    static unsigned int const maxSize = 100;
    int ez;           // Endezeiger
    int sz;           // Startzeiger
    T data[maxSize];
    void error(char *info);
};
```

Implementierung: (Version 2; mit Ringspeicher)

```
Schlange::Schlange() {
    sz = 0;
    ez = -1;
}
T Schlange::front() {
    if (empty()) error("leer");
    return data[sz];
}
bool Schlange::empty() {
    return (ez == -1);
}
bool Schlange::full() {
    if (empty()) return false;
    return ((ez + 1) % maxSize) == sz;
}
```

Implementierung: (Version 2; mit Ringspeicher)

```
void Schlange::enq(T &x) {  
    if (full(s)) error("full");  
    ez = (ez + 1) % maxSize;  
    data[ez] = x;  
}
```

Laufzeit:

unabhängig von Größe  
der Schlange

```
void Schlange::deq() {  
    if (empty(s)) error("leer");  
    if (sz == ez) { sz = 0; ez = -1; }  
    else sz = (sz + 1) % maxSize;  
}
```

Laufzeit:

unabhängig von Größe  
der Schlange

Unbefriedigend bei der Implementierung:

Maximale festgelegte Größe des Stapels bzw. der Schlange!

→ Liegt an der unterliegenden Datenstruktur Array:

Array ist **statisch**, d.h.

**Größe** wird zur Übersetzungszeit **festgelegt**

und ist **während** der **Laufzeit** des Programms **nicht veränderbar!**

Schön wären **dynamische** Datenstrukturen, d.h.

**Größe** wird zur Übersetzungszeit **nicht festgelegt**

und ist **während** der **Laufzeit** des Programms **veränderbar!**

⇒ **Dynamischer Speicher!** (Stichwort: `new / delete` )

### Lineare Datenstrukturen: Schlange (*engl. queue*)

create : → Schlange  
enq : Schlange x T → Schlange  
deq : Schlange → Schlange  
front : Schlange → T  
empty : Schlange → bool

create : erzeugt leere Schlange  
enq : hängt Element ans Ende der Schlange  
deq : entfernt Kopf der Schlange  
front : gibt im Kopf der Schlange gespeichertes Element zurück  
empty : prüft, ob Schlange leer ist

→ Implementierung mit statischen Speicher ersetzen durch dynamischen Speicher

### Bauplan:

Datentyp \*Variable = **new** Datentyp; (Erzeugen)

**delete** Variable; (Löschen)

### Bauplan für Arrays:

Datentyp \*Variable = **new** Datentyp[Anzahl]; (Erzeugen)

**delete[ ]** Variable; (Löschen)

### Achtung:

Dynamisch erzeugte Objekte müssen auch wieder gelöscht werden!  
Keine automatische Speicherbereinigung!



Vorüberlegungen für ADT Schlange mit dynamischen Speicher:

Wir können bei der Realisierung der Schlange statt statischen (Array) nun dynamischen Speicher verwenden ...

Ansatz: `new int[oldsize+1]` ... bringt uns das weiter?

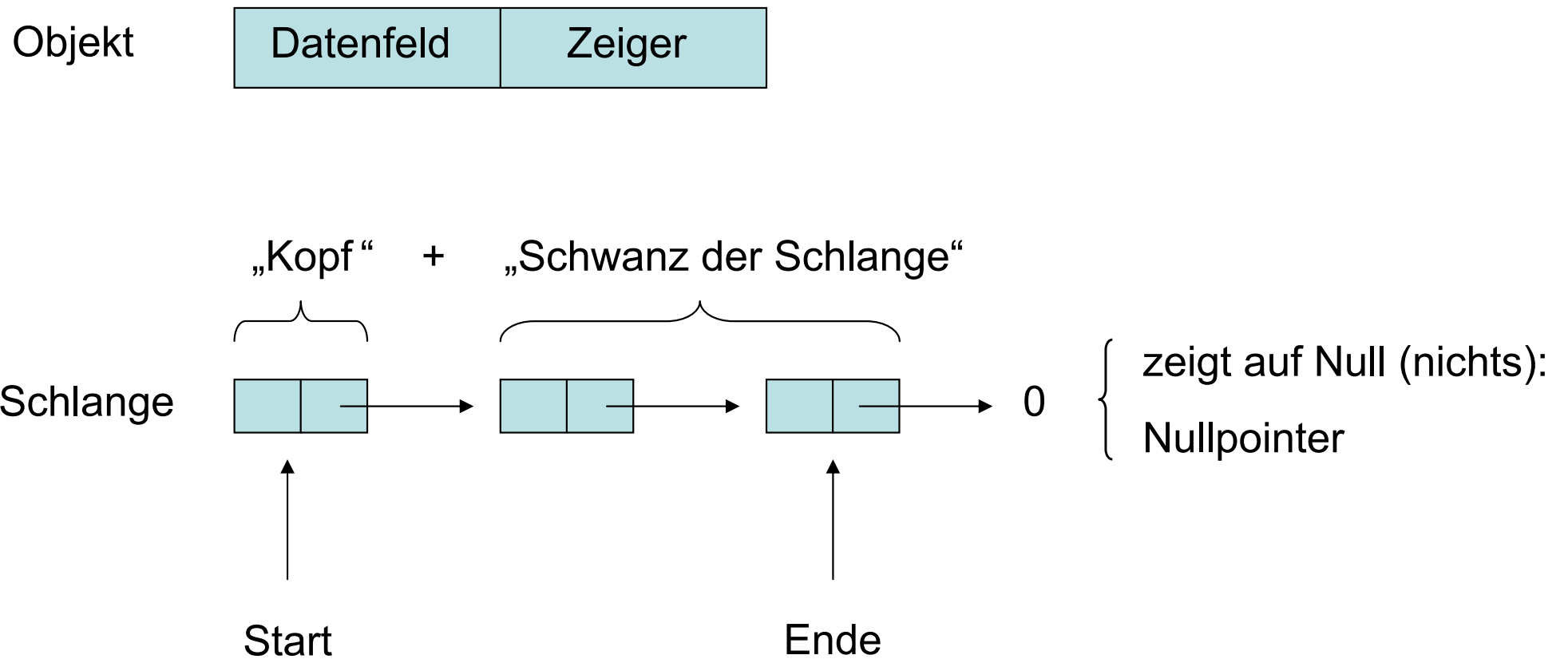
→ Größe kann zwar zur Laufzeit angegeben werden, ist aber dann fixiert!

Falls maximale Größe erreicht, könnte man

1. größeres Array anlegen
2. Arraywerte ins größere Array **kopieren** und
3. kleineres Array löschen.

} **ineffizient!**

Vorüberlegungen für ADT Schlange mit dynamischen Speicher:



Klassendefinition: (Version 3; mit dynamischem Speicher)

```
typedef int 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 *info); // Hilfsmethode: Fehlermeldung
};
```

Implementierung: (Version 3)

```
Schlange::Schlange() {
    ez = NULL;
}
T Schlange::front() {
    if (empty()) error("leer");
    return sz->data;
}
bool Schlange::empty() {
    return (ez == NULL);
}
void Schlange::clear() {
    while (!empty()) deq();
}
void Schlange::error(char *info) {
    cerr << info << endl;
    exit(1);
}
```

NULL ist der Nullzeiger!

```
Schlange::~~Schlange() {
    clear();
}
```

Implementierung: (Version 3)

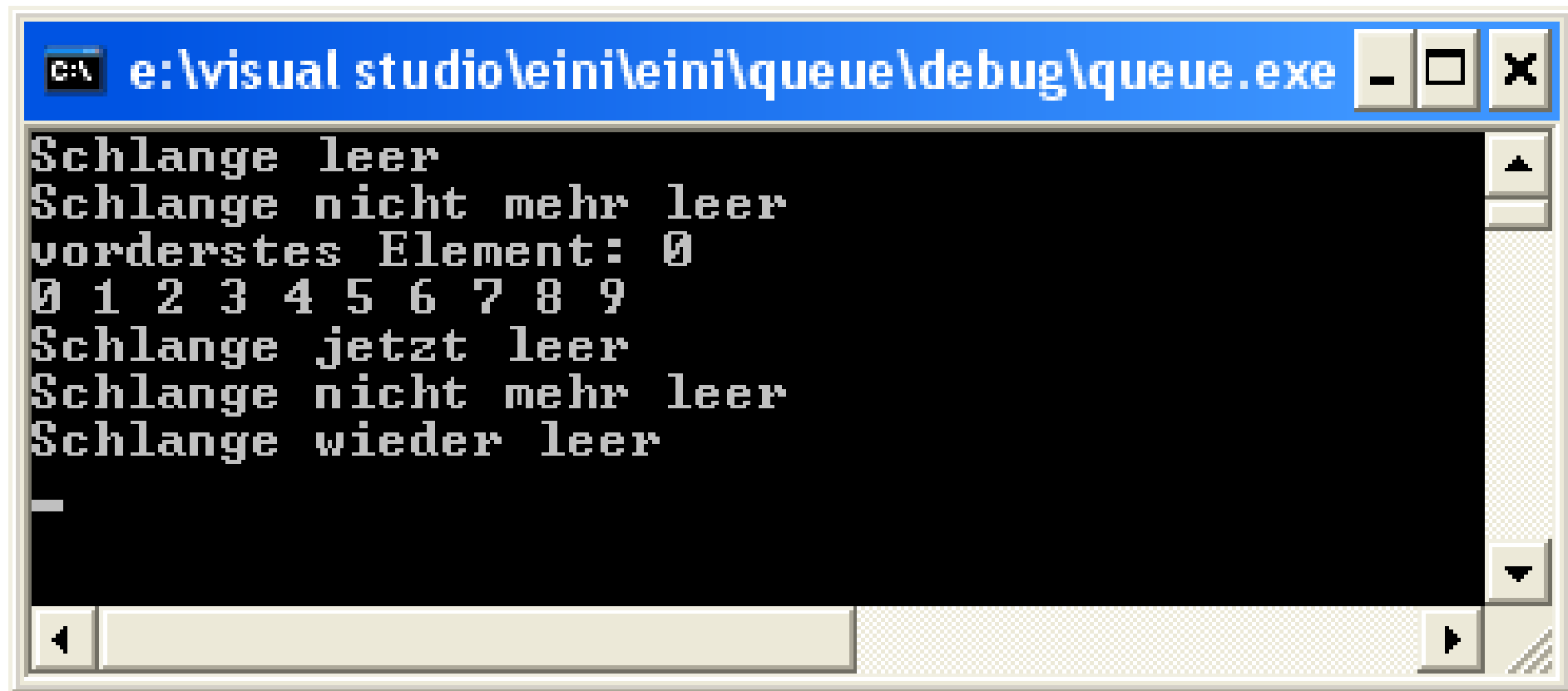
```
void Schlange::enq(T &x) {
    Objekt *obj = new Objekt;           // neues Objekt anlegen
    obj->data = x;                        // Nutzdaten speichern
    obj->tail = NULL;
    if (empty()) sz = obj;               // falls leer nach vorne,
    else ez->tail = obj;                 // sonst hinten anhängen
    ez = obj;                            // Endezeiger aktualisieren
}

void Schlange::deq() {
    if (empty()) error("leer");
    Objekt *obj = sz;                   // Zeiger auf Kopf retten
    sz = sz->tail;                       // Start auf 2. Element
    if (sz == NULL) ez = NULL;          // Schlange leer!
    delete obj;                          // ehemaliges 1. Element
}                                       // löschen
```

**Testprogramm!**

```
int main() {
    Schlange s;
    if (s.empty()) cout << "Schlange leer" << endl;
    for (int i = 0; i < 10; i++) s.enq(i);
    if (!s.empty()) cout << "Schlange nicht mehr leer" << endl;
    cout << "vorderstes Element: " << s.front() << endl;
    while (!s.empty()) {
        cout << s.front() << " ";
        s.deq();
    }
    cout << endl;
    if (s.empty()) cout << "Schlange jetzt leer" << endl;

    for (i = 0; i < 100; i++) s.enq(i);
    if (!s.empty()) cout << "Schlange nicht mehr leer" << endl;
    s.clear();
    if (s.empty()) cout << "Schlange wieder leer" << endl;
    return 0;
}
```



```
e:\visual studio\ini\ini\queue\debug\queue.exe  
Schlange leer  
Schlange nicht mehr leer  
vorderstes Element: 0  
0 1 2 3 4 5 6 7 8 9  
Schlange jetzt leer  
Schlange nicht mehr leer  
Schlange wieder leer  
-
```

## Kopieren von Klassenobjekten

```
class Schlange {  
    T data[100];  
    int sz, ez;  
};
```

→

```
Schlange s1;  
for (int i=0;i<10;i++)  
    s1.enq(i);  
Schlange s2 = s1;
```

**statischer Speicher:**  
byteweises  
Speicherabbild!  
⇒ **OK!**

```
class Schlange {  
    struct Objekt {  
        Objekt *tail;  
        T data;  
    } *sz, *ez;  
};
```

→

```
Schlange s1;  
for (int i=0;i<10;i++)  
    s1.enq(i);  
Schlange s2 = s1;
```

**dynam. Speicher:**  
byteweises  
Speicherabbild!  
⇒ **Problem!**

Es werden nur die Inhalte der Zeiger kopiert!

Bei Verwendung von dynamischem Speicher muss auch dieser kopiert werden!

⇒ In C++ kann das durch den **Kopierkonstruktor** realisiert werden!



## Kopierkonstruktor (*copy constructor*)

Wird für eine Klasse **kein** Kopierkonstruktor implementiert, dann erzeugt ihn der Compiler **automatisch!**

### Achtung!

Es wird dann ein **byteweises Speicherabbild** des Objektes geliefert!

⇒ „flache Kopie“ (engl. *shallow copy*)

### Problem:

- Konstruktor fordert dynamischen Speicher an → nur Kopie des Zeigers
- Konstruktor öffnet exklusive Datei (o.a. Resource) → nicht teilbar! Crash!

⇒ dann „tiefe Kopie“ (engl. *deep copy*) nötig!

⇒ man **muss** Kopierkonstruktor (und Destruktor) implementieren!

```

#include <iostream>
#include "Stapel.h"      // statischer Speicher
#include "Schlange.h"   // dynamischer Speicher
using namespace std;
int main() {
    Stapel stack1;
    Schlange queue1;
    for (int i = 0; i < 10; i++) {
        stack1.push(i);
        queue1.enq(i);
    }
    Stapel stack2 = stack1;
    Schlange queue2 = queue1;
    while (!stack1.empty()) stack1.pop();
    while (!queue1.empty()) queue1.deq();
    while (!stack2.empty()) {
        cout << stack2.top() << endl;
        stack2.pop();
    }
    while (!queue2.empty()) {
        cout << queue2.front() << endl;
        queue2.deq();
    }
    return 0;
}

```

### Programmiertes Unheil:

Stapel1 / Schlange1 mit Daten belegen.

Stapel1 / Schlange1 kopieren

Stapel1 / Schlange1 löschen

kopierten Stapel ausgeben

kopierte Schlange ausgeben ...



**crash!**

## Kopierkonstruktor (*copy constructor*)

```
class Schlange {  
public:  
    Schlange(); // Konstruktor  
    Schlange(const Schlange & s); ←  
    ~Schlange(); // Destruktor  
};
```

Kann wie eine Zuweisung  
interpretiert werden!

Kopierkonstruktor

```
Schlange::Schlange(const Schlange& s){  
    ez = NULL;  
    Objekt *ptr = s.sz;  
    while (ptr != NULL) {  
        enq(ptr->data);  
        ptr = ptr->tail;  
    }  
}
```

Entstehendes  
Objekt wird mit einem  
**bestehenden** Objekt  
initialisiert!

## Kopierkonstruktor (*copy constructor*)

### Bauplan:

ObjektTyp (`const` ObjektTyp & bezeichner);

→ Kopierkonstruktor liefert / soll liefern byteweises Speicherabbild des Objektes

Wird **automatisch** aufgerufen, wenn:

1. ein neues Objekt erzeugt und mit einem bestehenden initialisiert wird;
2. ein Objekt per Wertübergabe an eine Funktion gereicht wird;
3. ein Objekt mit `return` als Wert zurückgegeben wird.

```
Punkt a(1.2, 3.4); // Neu
```

```
Punkt b(a); // Kopie: direkter Aufruf des Kopierkonstruktors
```

```
Punkt c = b; // Kopie: bewirkt Aufruf des Kopierkonstruktors
```

```
b = a; // Zuweisung! Keine Kopie! → gleiche Problematik!
```

Wenn für eine Klasse der Zuweisungsoperator **nicht** überschrieben wird, dann macht das der Compiler **automatisch!**

### **Vorsicht!**

Speicher des Objektes wird **byteweise** überschrieben!

### **Problem:**

z.B. wenn Objekt dynamischen Speicher verwendet

⇒ gleiche Problematik wie beim Kopierkonstruktor

### **Merke:**

Wenn die Implementierung eines **Kopierkonstruktors** nötig ist, dann höchstwahrscheinlich auch **Destruktor** und überschriebene **Zuweisung!**

## Überladen von Operatoren

- Operator ist eine Verknüpfungsvorschrift!
- Kann man auffassen als Name einer Funktion:

**Bsp:** Addition  $a + b$  interpretieren als  $+(a, b)$

- in C++ als:  $c = \underbrace{\text{operator+}}_{\text{Funktionsname}} (\underbrace{a, b}_{\text{Argumente}})$

### Zweck:

eine Klasse mit Funktionalität ausstatten,  
die vergleichbar mit elementarem Datentyp ist!

insbesondere bei  
**Zuweisung** und **Gleichheit**

### Vorteil:

Quellcode wird übersichtlicher

## Überladen von Operatoren

Welche?

+	^	==	+=	^=	!=	<<	()
-	&	>	-=	&=	&&	<<=	new
*		>=	*=	=		>>	delete
/	~	<	/=	++	->	>>=	=
%	!	<=	%=	--	->*	[]	

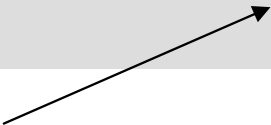
Wie?

Objektyp& operator@(const ObjektTyp& bezeichner)

Objektyp operator@ (const ObjektTyp& bezeichner)

## Überladen von Operatoren

```
Schlange& operator= (const Schlange& s) {  
    clear(); // Speicher freigeben  
    Objekt *ptr = s.sz;  
    while (ptr != NULL) {  
        enq(ptr->data);  
        ptr = ptr->tail;  
    }  
    return *this;  
}
```



Zuweisung

**this** ist ein Zeiger auf das Objekt selbst!

bei der Zuweisung wird ja keine neue Instanz erzeugt;  
tatsächlich wird vorhandene Instanz verändert;

deshalb ist Rückgabewert eine Referenz auf sich selbst!



## Überladen von Operatoren

```
bool operator== (const Schlange& s) {  
    Objekt *ptr1 = sz; // this->sz  
    Objekt *ptr2 = s.sz;  
    while (ptr1 != NULL && ptr2 != NULL) {  
        if (ptr1->data != ptr2->data) return false;  
        ptr1 = ptr1->tail;  
        ptr2 = ptr2->tail;  
    }  
    return (ptr1 == ptr2);  
}
```

Test auf Gleichheit

Zwei Schlangen sind gleich genau dann, wenn sie

1. gleich viele Elemente haben und
2. die Inhalte in gleicher Reihenfolge paarweise gleich sind.

### Unterschied zwischen Kopierkonstruktor und Zuweisung

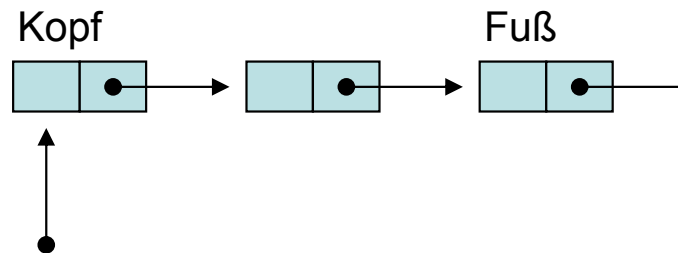
#### Kopierkonstruktor:

Initialisierung einer **neu** deklarierten Variable von **existierender** Variable

#### Zuweisung:

- wirkt zwar wie Kopierkonstruktor (flache Kopie bzw. tiefe Kopie), überschreibt jedoch Speicher der **existierenden** Variable mit dem Speicher der zuweisenden, **existierenden** Variable
- zusätzlich ggf. Aufräumen: Freigabe dynamischer Speicher!
- außerdem: Rückgabe einer Referenz auf sich selbst

## 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
prepend	:	T x Liste	→ Liste
clear	:		→ Liste
is_elem	:	T x Liste	→ bool

hängt am Ende an  
vor Kopf einfügen

ist Element enthalten?

## ADT Liste (1. Version)

```
typedef int T;

class Liste {
public:
    Liste(); // Konstruktor
    Liste(const Liste& 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 Liste (1. Version)

```
Liste::Liste() {
    sz = NULL;
}

void Liste::clear(Objekt *obj) {
    if (obj == NULL) return;
    clear(obj->next);
    delete obj;
}

void Liste::clear() {
    clear(sz);
    sz = NULL;
}

Liste::~~Liste() {
    clear();
}
```

**Laufzeit:**

unabhängig von Listenlänge

rekursives Löschen von  
„hinten“ nach „vorne“**Laufzeit:**

proportional zur Listenlänge



## ADT Liste (1. Version)

```
bool Liste::empty() {  
    return (sz == NULL);  
}
```

```
bool Liste::is_elem(const T& x) {  
    Objekt *ptr = sz;  
    while (ptr != NULL) {  
        if (ptr->data == x) return true;  
        ptr = ptr->next;  
    }  
    return false;  
}
```

```
void Liste::prepend(const T& x){  
    Objekt *obj = new Objekt;  
    obj->data = x;  
    obj->next = sz;  
    sz = obj;  
}
```

**Laufzeit:**

unabhängig von Listenlänge

iterativer Durchlauf von  
„vorne“ nach „hinten“

**Laufzeit:**

proportional zur Listenlänge

**Laufzeit:**

unabhängig von Listenlänge

## ADT Liste (1. Version)

```
void Liste::append(const T& x) {
    Objekt *obj = new Objekt;
    obj->data = x;
    obj->next = NULL;
    if (empty()) sz = obj;
    else {
        Objekt *ptr = sz;
        while (ptr->next != NULL)
            ptr = ptr->next;
        ptr->next = obj;
    }
}
```

} neuen Eintrag erzeugen

Liste leer? → Kopf = neuer Eintrag

} iterativer Durchlauf von  
„vorne“ nach „hinten“

**Laufzeit:**

proportional zur Listenlänge

```
Liste::Liste(const Liste& liste) : sz(NULL) {
    for (Objekt *ptr = liste.sz; ptr != NULL; ptr = ptr->next)
        append(ptr->data);
}
```

**Laufzeit:** quadratisch proportional zur Listenlänge!

### ADT Liste (1. Version)

#### Zusammenfassung:

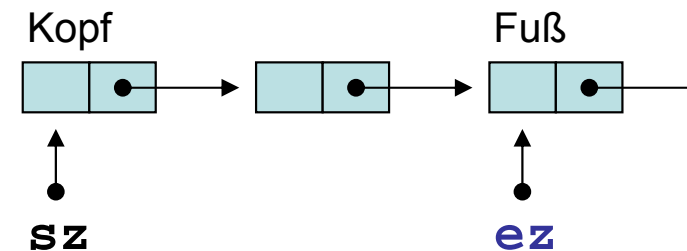
1. Laufzeit von **clear** proportional zur Listenlänge
  - kann nicht verbessert werden, weil ja jedes Element gelöscht werden muss
  - unproblematisch, weil nur selten aufgerufen
2. Laufzeit des **Kopierkonstruktors** quadratisch proportional zur Listenlänge
  - kann nur verbessert werden, wenn **append** verbessert werden kann
  - bestenfalls Laufzeit proportional zur Listenlänge: muss alle Elemente kopieren!
3. Laufzeit von **is\_elem** proportional zur Listenlänge
  - kann bei dieser **Datenstruktur** nicht verbessert werden
  - später verbessert durch ADT BinärerSuchbaum
4. Laufzeit von **append** proportional zur Listenlänge
  - kann durch Veränderung der **Implementierung** verbessert werden
  - zusätzlicher Zeiger auf das Ende der Liste



## ADT Liste (2. Version)

```
class Liste {  
public:  
    // keine Änderungen  
private:  
    struct Objekt {  
        T data;  
        Objekt *next;  
    } *sz, *ez;  
    // sonst keine Änderungen  
};
```

Liste besteht aus 2 Zeigern:  
Zeiger auf Listenkopf (Start)  
Zeiger auf Listenuß (Ende)



Kennzeichnung der leeren Liste jetzt durch Nullzeiger bei **ez**.

```
Liste::Liste() {  
    ez = NULL;  
}
```

```
bool Liste::empty() {  
    return (ez == NULL);  
}
```

```
Liste::~~Liste() {  
    clear();  
}
```

## ADT Liste (2. Version)

```
void Liste::clear(Objekt *obj) {
    if (obj == NULL) return;
    clear(obj->next);
    delete obj;
}
void Liste::clear() {
    clear(sz);
    ez = NULL;
}
```

keine Änderungen!

**Laufzeit:**

proportional zur Listenlänge

→ keine Verbesserung (OK)

```
bool Liste::is_elem(const T& x) {
    Objekt *ptr = sz;
    while (ptr != NULL) {
        if (ptr->data == x) return true;
        ptr = ptr->next;
    }
    return false;
}
```

keine Änderungen!

**Laufzeit:**

proportional zur Listenlänge

→ keine Verbesserung (OK)

## ADT Liste (2. Version)

```
void Liste::prepend(const T& x) {  
    Objekt *obj = new Objekt;  
    obj->data = x;  
    obj->next = sz;  
    sz = obj;  
}
```

keine Änderungen!

**Laufzeit:**  
unabhängig von Listenlänge

```
void Liste::append(const T& x) {  
    Objekt *obj = new Objekt;  
    obj->data = x;  
    obj->next = NULL;  
    if (empty()) sz = obj;  
    else ez->next = obj;  
    ez = obj;  
}
```

**Laufzeit:**  
unabhängig von Listenlänge

→ Verbesserung!

## ADT Liste (2. Version)

```
Liste::Liste(const Liste& liste) {
    ez = NULL;
    for (Objekt *ptr = liste.sz; ptr != NULL; ptr = ptr->next)
        append(ptr->data);
}
```

### Laufzeit:

proportional zur Listenlänge, weil append verbessert wurde → **Verbesserung!**

	Version 1		Version 2	
Elemente	Debug	Release	Debug	Release
<b>5000</b>	145469	32107	9504	1627
<b>10000</b>	566812	125605	19491	3279
<b>20000</b>	2234480	495467	38610	6444

Anzahl Elemente mal 4 ⇒  
 Laufzeit mal  $4^2=16$  (Version 1)  
 Laufzeit mal 4 (Version 2)

Laufzeit in  $\mu$ sek. für Kopieroperation

### ADT Liste (2. Version)

#### Zusammenfassung:

1. Laufzeit von `clear` proportional zur Listenlänge
  - kann nicht verbessert werden, weil ja jedes Element gelöscht werden muss
  - unproblematisch, weil nur selten aufgerufen
2. Laufzeit von `is_elem` proportional zur Listenlänge
  - kann bei dieser **Datenstruktur** nicht verbessert werden
  - verbessern wir gleich durch ADT BinärBaum
3. Laufzeit von `append` unabhängig von Listenlänge
  - war proportional zur Listenlänge in 1. Version
  - Verbesserung erzielt durch Veränderung der **Implementierung**
4. Laufzeit des Kopierkonstruktors proportional zur Listenlänge
  - war quadratisch proportional zur Listenlänge in 1. Version
  - Verbesserung erzielt durch Verbesserung von `append`

### ADT Binärer Suchbaum

Vorbemerkungen:

Zahlenfolge (z. B. 17, 4, 36, 2, 8, 19, 40, 6, 7, 37) soll gespeichert werden, um später darin suchen zu können

Man könnte sich eine Menge  $A$  vorstellen mit Anfrage: Ist  $40 \in A$  ?

Mögliche Lösung: Zahlen in einer Liste speichern und nach 40 suchen ...

... aber: **nicht effizient**,  
weil im schlechtesten Fall alle Elemente überprüft werden müssen!

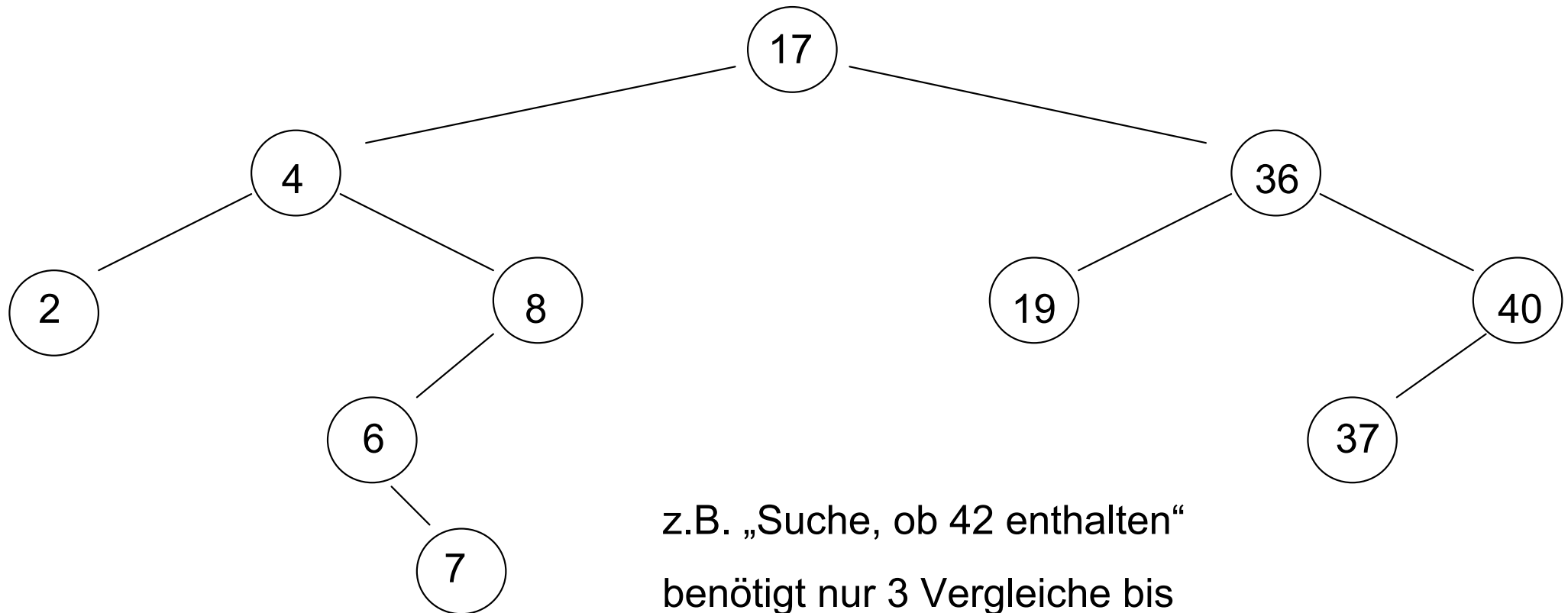
Bessere Lösungen?

## ADT Binärer Suchbaum

Beispiel:

Zahlenfolge 17, 4, 36, 2, 8, 19, 40, 6, 7, 37

kleiner : nach links  
größer : nach rechts

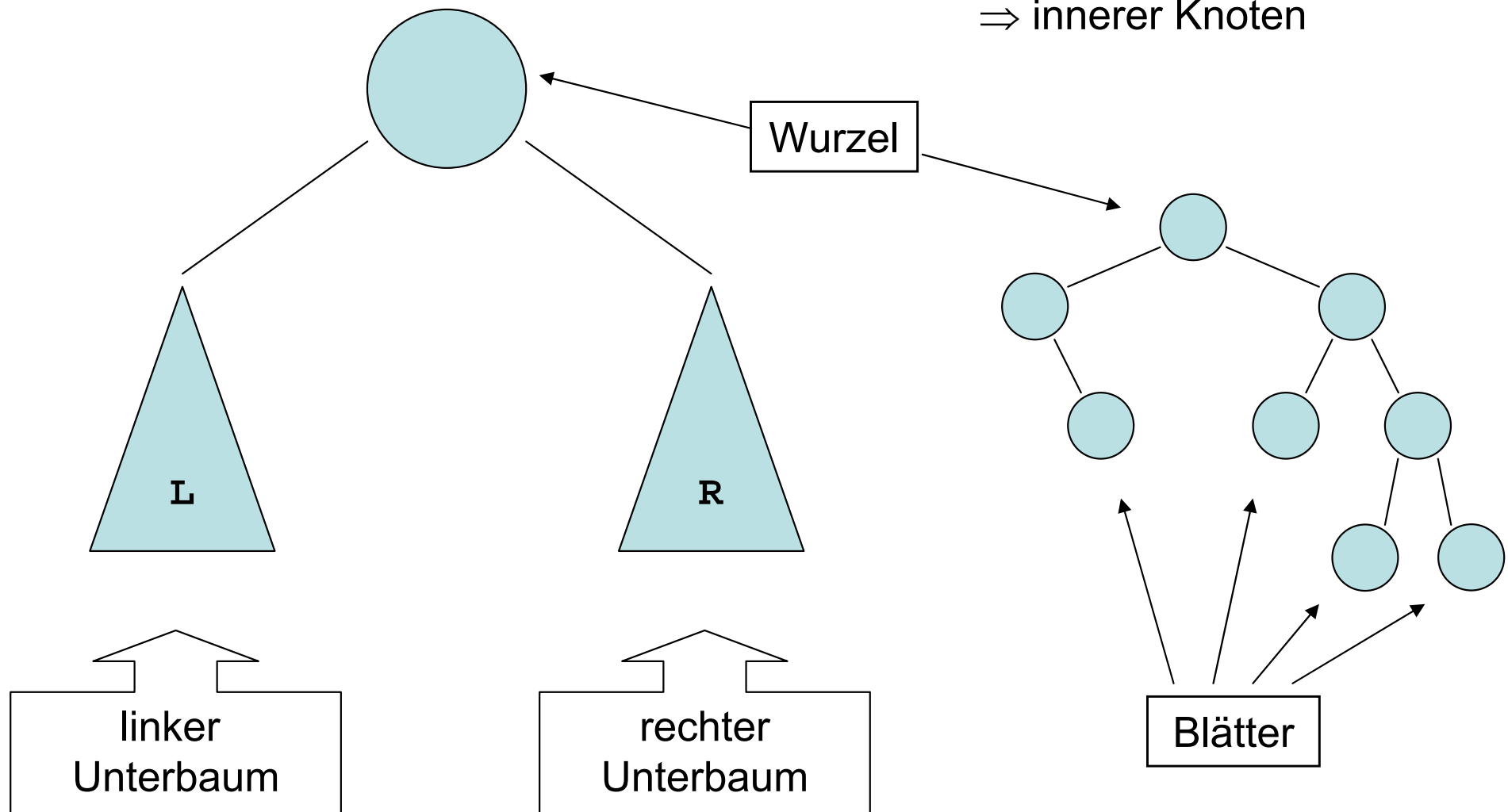


z.B. „Suche, ob 42 enthalten“

benötigt nur 3 Vergleiche bis zur Entscheidung **false**

## ADT Binärer Suchbaum: Terminologie

keine Wurzel und kein Blatt  
⇒ innerer Knoten





## ADT Binärer Suchbaum: Klassendefinition

```
typedef int 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 = 0; }
    void insert(T x)    { root = insert(root, x); }
    bool isElem(T x)    { return isElem(root, x); }
    void clear()        { clear(root); root = 0; }
    ~BinTree()          { clear(); }
};
```

leerer Unterbaum  
→ Nullzeiger

## ADT Binärer Suchbaum: Element suchen

```
bool BinTree::isElem(Node *node, T key) {  
    if (node == 0) return false;  
    if (node->data == key) return true;  
    if (node->data < key) return isElem(node->right, key);  
    return isElem(node->left, key);  
}
```


Rekursive Suche:

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

Falls Knoten Element enthält: Erfolg!

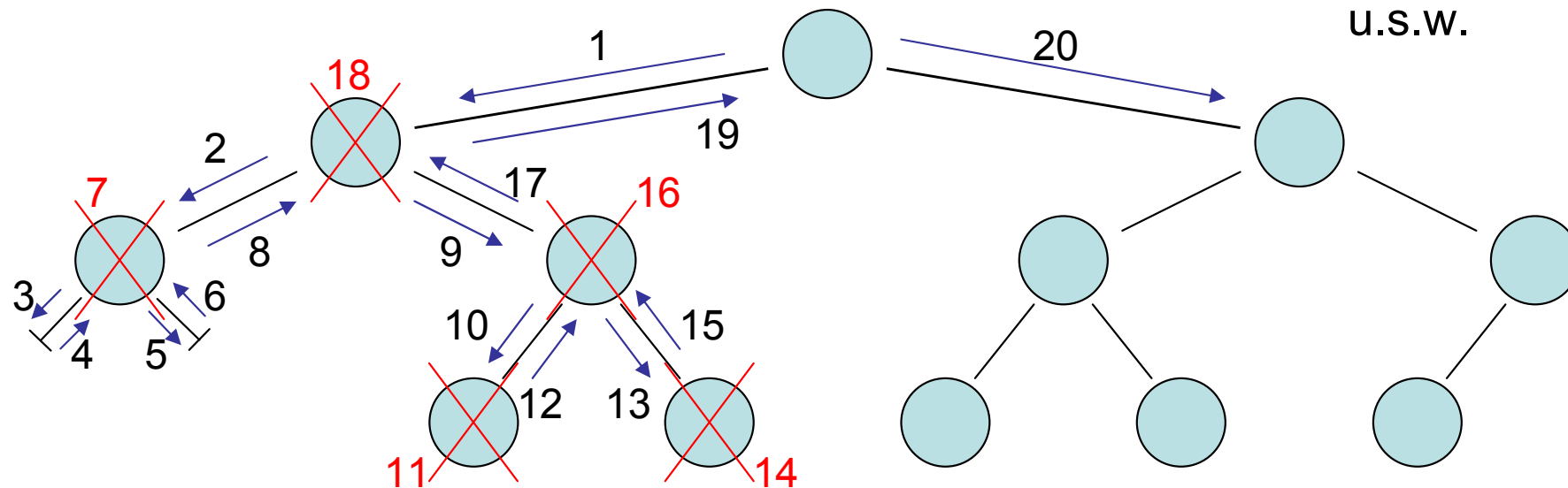
Falls Unterbaum leer, dann Element  
nicht vorhanden.

Rekursionsverankerung  
(Abbruchbedingung)



## ADT Binärer Suchbaum: Aufräumen

```
void BinTree::clear(Node *node) {
    if (node == 0) return; // Rekursionsabbruch
    clear(node->left); // linken Unterbaum löschen
    clear(node->right); // rechten Unterbaum löschen
    delete node; // Knoten löschen
}
```

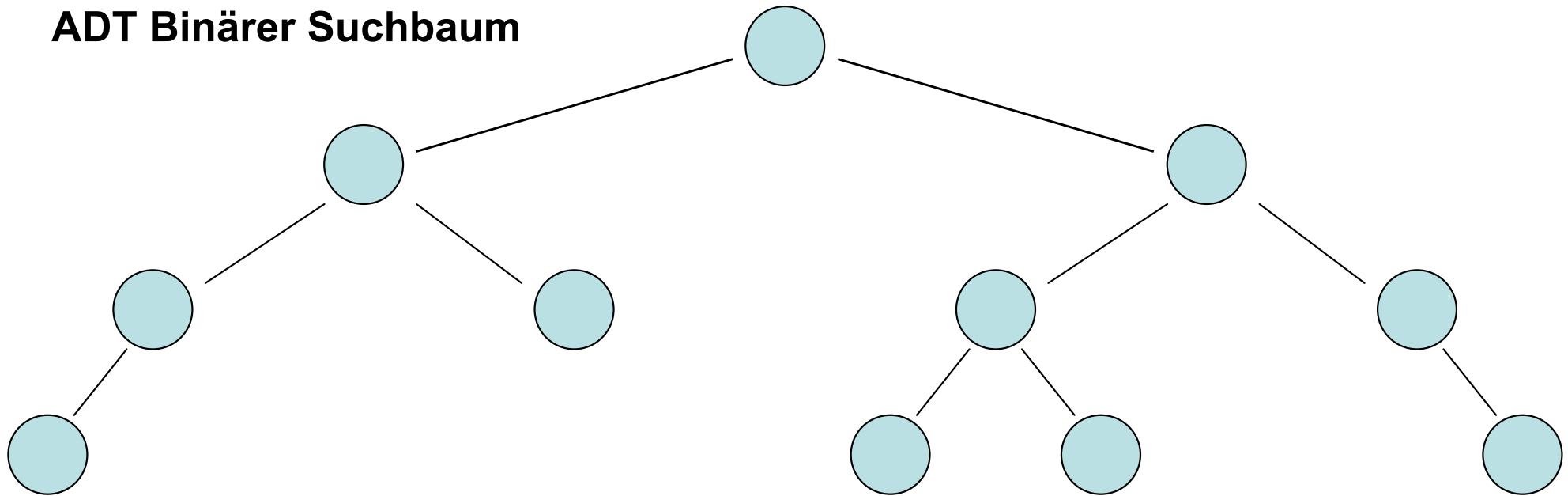


## ADT Binärer Suchbaum: Einfügen

```
BinTree::Node *BinTree::insert(Node *node, T key) {  
    if (node == 0) {  
        node = new Node;  
        node->data = key;  
        node->left = node->right = 0;  
        return node;  
    }  
    if (node->data < key)  
        node->right = insert(node->right, key);  
    else if (node->data > key)  
        node->left = insert(node->left, key);  
    return node;  
}
```

Rekursives Einfügen

## ADT Binärer Suchbaum



Höhe := Länge des längsten Pfades von der Wurzel zu einem Blatt.

Höhe(leerer Baum) = 0

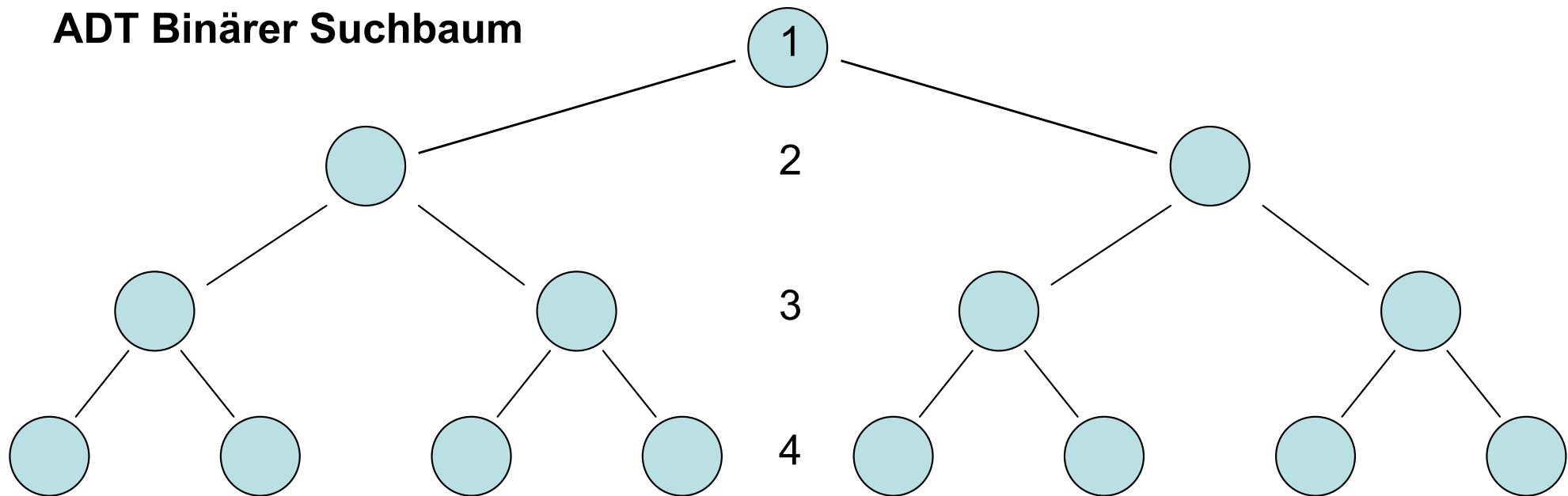
Höhe(nicht leerer Baum) =  $1 + \max \{ \text{Höhe}(\text{linker U-Baum}), \text{Höhe}(\text{rechter U-Baum}) \}$



Anmerkung: rekursive Definition!

(U-Baum = Unterbaum)

## ADT Binärer Suchbaum

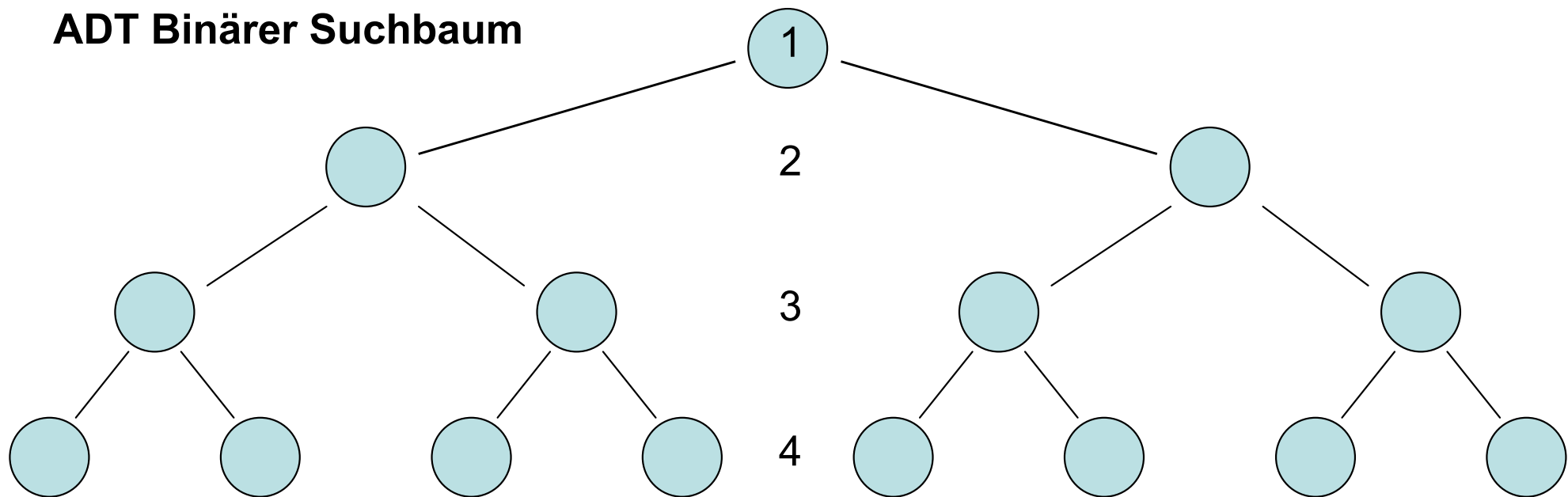


Auf Ebene  $k$  können jeweils zwischen 1 und  $2^{k-1}$  Elemente gespeichert werden.

⇒ In einem Baum der Höhe  $h$  können also zwischen  $h$  und

$$\sum_{k=1}^h 2^{k-1} = 2^h - 1 \quad \text{Elemente gespeichert werden!}$$

## ADT Binärer Suchbaum



- Ein **vollständiger Baum** der Höhe  $h$  besitzt  $2^h - 1$  Knoten.  
Man braucht maximal  $h$  Vergleiche, um Element (ggf. nicht) zu finden.  
Bei  $n = 2^h - 1$  Elementen braucht man  $\log_2(n) < h$  Vergleiche!
- Ein **degenerierter Baum** der Höhe  $h$  besitzt  $h$  Knoten (= lineare Liste).  
Man braucht maximal  $h$  Vergleiche, um Element (ggf. nicht) zu finden.  
Bei  $n = h$  braucht man also  $n$  Vergleiche!

**Datei** := speichert Daten in linearer Anordnung

Zwei Typen:

- **ASCII-Dateien**

- sind mit Editor les- und schreibbar
- Dateiendung („suffix“ oder „extension“) meist `.txt` oder `.asc`
- betriebssystem-spezifische Übersetzung von Zeichen bei Datentransfer zwischen Programm und externem Speicher

- **Binär-Dateien**

- werden byteweise beschrieben und gelesen
- lesen / schreiben mit Editor ist keine gute Idee
- schnellerer Datentransfer, da keine Zeichenübersetzung



Hier: einfache Dateibehandlung!

- Dateien können **gelesen** oder **beschrieben** werden.
- Vor dem ersten Lesen oder Schreiben muss **Datei geöffnet** werden.
- Man kann prüfen, ob das Öffnen funktioniert hat.
- Nach dem letzten Lesen oder Schreiben muss **Datei geschlossen** werden.
- Bei zu lesenden Dateien kann gefragt werden, ob **Ende der Datei** erreicht ist.
- Beim Öffnen einer zu schreibenden Datei wird vorheriger Inhalt gelöscht!
- Man kann noch viel mehr machen ...

wir benötigen:

```
#include <fstream>           bzw.           <fstream.h>
```

- Eingabe-Datei = input file

```
ifstream Quelldatei;
```

↑                    ↑  
Datentyp            Bezeichner

- Öffnen der Datei:

```
Quelldatei.open(dateiName);
```

ist Kurzform von  
`Quelldatei.open(dateiName, modus);`

wobei fehlender modus bedeutet:  
ASCII-Datei,  
Eingabedatei (weil ifstream)

- Ausgabe-Datei = output file

```
ofstream Zieldatei;
```

↑                    ↑  
Datentyp            Bezeichner

- Öffnen der Datei:

```
Zieldatei.open(dateiName);
```

ist Kurzform von  
`Quelldatei.open(dateiName, modus);`

wobei fehlender modus bedeutet:  
ASCII-Datei,  
Ausgabedatei (weil ofstream)

### modus:

<code>ios::binary</code>	binäre Datei
<code>ios::in</code>	öffnet für Eingabe (implizit bei ifstream)
<code>ios::out</code>	öffnet für Ausgabe (implizit bei ofstream)
<code>ios::app</code>	hängt Daten am Dateiende an
<code>ios::nocreate</code>	wenn Datei existiert, dann nicht anlegen

**Warnung:** teilweise Compiler-abhängig  
(`nocreate` fehlt in MS VS 2003, dafür `trunc`)

Man kann diese Schalter / Flags miteinander kombinieren via:

`ios::binary | ios::app` (öffnet als binäre Datei und hängt Daten an)

- **Datei öffnen**

`file.open(fileName)` bzw. `file.open(fileName, modus)`

falls Öffnen fehlschlägt, wird Nullpointer zurückgegeben

- **Datei schließen**

`file.close()`

sorgt für definierten Zustand der Datei auf Dateisystem;  
bei nicht geschlossenen Dateien droht Datenverlust!

- **Ende erreicht?**

ja falls `file.eof() == true`

- **Lesen** (von ifstream)

`file.get(c);` liest ein Zeichen

`file >> x;` liest verschiedene Typen

- **Schreiben** (von ofstream)

`file.put(c);` schreibt ein Zeichen

`file << x;` schreibt verschiedene Typen

### Merke:

1. Auf eine geöffnete Datei darf immer nur einer zugreifen.
2. Eine geöffnete Datei belegt Ressourcen des Betriebssystems.  
⇒ Deshalb Datei nicht länger als nötig geöffnet halten.
3. Eine geöffnete Datei unbekannter Länge kann solange gelesen werden, bis das Ende-Bit (end of file, EOF) gesetzt wird.
4. Der Versuch, eine nicht vorhandene Datei zu öffnen (zum Lesen) oder eine schreibgeschützte Datei zu öffnen (zum Schreiben), führt zu einem Nullpointer.  
⇒ Das muss überprüft werden, sonst Absturz bei weiterer Verwendung!
5. Dateieingabe und -ausgabe (input/output, I/O) ist sehr langsam im Vergleich zu den Rechenoperationen.  
⇒ I/O Operationen minimieren.

**"The fastest I/O is no I/O."**

Nils-Peter Nelson, Bell Labs

```
#include <iostream>
#include <fstream>

using namespace std;

int main() { // zeichenweise kopieren
    ifstream Quelldatei;
    ofstream Zieldatei;

    Quelldatei.open("quelle.txt");
    if (!Quelldatei.is_open()) {
        cerr << "konnte Datei nicht zum Lesen öffnen\n";
        exit(1);
    }
    Zieldatei.open("ziel.txt");
    if (!Zieldatei.is_open()) {
        cerr << "konnte Datei nicht zum Schreiben öffnen\n";
        exit(1);
    }
}
```

```
while (!Quelldatei.eof()) {  
    char c;  
    Quelldatei.get(c);  
    Zieldatei.put(c);  
}  
  
Quelldatei.close();  
Zieldatei.close();  
}
```

offene  
Datei



Start



aktuelle Position



eof() == true

### Bisher:

Zeichenketten wie `char str[20];`

- Relikt aus C-Programmierung!
- bei größeren Programmen mühevoll, lästig, ...
- ... und insgesamt **fehlerträchtig!**

### Jetzt:

Zeichenketten aus C++

- sehr angenehm zu verwenden (keine 0 am Ende, variable Größe, ...)
- eingebaute (umfangreiche) Funktionalität

wie benötigen: `#include <string>` und `using namespace std;`



## Datendefinition / Initialisierung

```
string s1;           // leerer String
string s2 = "xyz";  // initialisieren mit C-String
string s3 = s2;     // vollständige Kopie!
string s4("abc");   // initialisieren mit C-String
string s5(s4);      // initialisieren mit C++-String
string s6(10, '*'); // ergibt String aus 10 mal *
string s7(1, 'x');  // initialisieren mit einem char
string sx('x');     // FEHLER!
string s8("");      // leerer String
```

## Eingebaute Funktionen

- Konvertierung C++-String nach C-String via `c_str()`

```
const char *Cstr = s2.c_str();
```

- Stringlänge `length()`

```
cout << s2.length();
```

- Index von Teilstring finden

```
int pos = s2.find("yz");
```

- Strings addieren

```
s1 = s2 + s3;
s4 = s2 + "hello";
s5 += s4;
```

- Strings vergleichen

```
if (s1 == s2) s3 += s2;
if (s3 < s8) flag = true;
```

- `substr()`,
- `replace()`,
- `erase()`,
- ...

### ADT Binäre Bäume: Anwendung

#### Aufgabe:

Gegeben sei eine Textdatei.

Häufigkeiten der vorkommenden Worte feststellen.

Alphabetisch sortiert ausgeben.

#### Strategische Überlegungen:

Lesen aus Textdatei → `ifstream` benutzen!

Sortierte Ausgabe → Binärbaum: schnelles Einfügen, sortiert „von selbst“

Worte vergleichen → C++ Strings verwenden!

#### Programmskizze:

Jeweils ein Wort aus Datei lesen und in Binärbaum eintragen.

Falls Wort schon vorhanden, dann Zähler erhöhen.

Wenn alle Wörter eingetragen, Ausgabe (Wort, Anzahl) via **Inorder**-Durchlauf.

## zusätzlicher Zähler im Knoten

```
struct Node {  
    std::string data;  
    unsigned int cnt;  
    BinBaum *left, *right;  
};
```

gelesenes Wort  
wie oft gelesen?

## zusätzlicher Konstruktor (zum Einlesen der Datei)

```
BinTree::BinTree(string &filename) {  
    ifstream source;  
    source.open(filename.c_str());  
    if (!source.is_open()) return 0;  
    string s;  
    while (!source.eof()) {  
        source >> s;  
        insert(s);  
    }  
    source.close();  
}
```

Datei öffnen

Worte einzeln  
auslesen und im  
Baum einfügen

Datei schließen

Einfügen (Änderungen in **rot**)

```
BinTree::Node *BinTree::insert(Node *node, T key) {
    if (node == 0) {
        node = new Node;
        node->data = key;
        node->cnt = 1;
        node->left = node->right = 0;
        return node;
    }
    if (node->data < key)
        node->right = insert(node->right, key);
    else if (node->data > key)
        node->left = insert(node->left, key);
    else
        node->cnt++;
    return node;
}
```

## Ausgabe (rekursiv)

```
void BinTree::print(Node *node) {  
    if (node == 0) return;  
    print(node->left);  
    cout << node->cnt << " " << node->data.c_str() << endl;  
    print(node->right);  
}
```



Dies ist die **Inorder**-Ausgabe.

### Präorder:

```
cout ...;  
Ausgabe(...);  
Ausgabe(...);
```

### Postorder:

```
Ausgabe(...);  
Ausgabe(...);  
cout ...;
```

### Hauptprogramm:

```
#include "BinTree.h"
using namespace std;

int main() {
    string s("quelle.txt");
    BinBaum b(s);
    b.print();
    return 0;
}
```

## Durchlaufstrategien:

- **Tiefensuche** („depth-first search“, DFS)

- Präorder

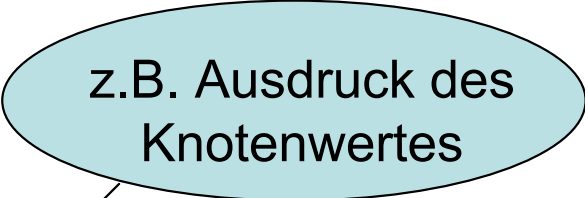
- Vor (*prä*) Abstieg in Unterbäume die „Knotenbehandlung“ durchführen

- Postorder

- Nach (*post*) Abstieg in bzw. Rückkehr aus Unterbäumen die „Knotenbehandlung“ durchführen

- Inorder

- Zwischen zwei Abstiegen „Knotenbehandlung“ durchführen



z.B. Ausdruck des Knotenwertes

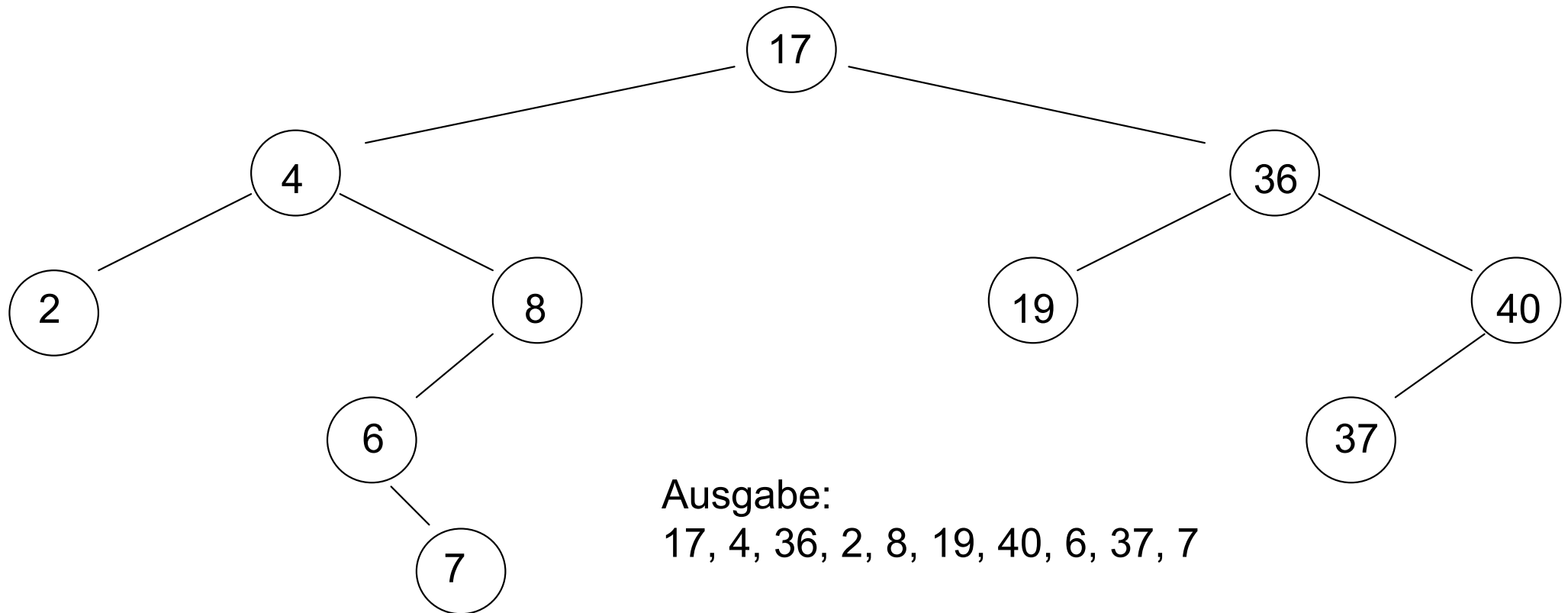
- **Breitensuche** („breadth-first search“, BFS; auch: „level search“)

auf jeder Ebene des Baumes werden Knoten abgearbeitet, bevor in die Tiefe gegangen wird



## Breitensuche

Beispiel: eingegebene Zahlenfolge 17, 4, 36, 2, 8, 19, 6, 7, 37



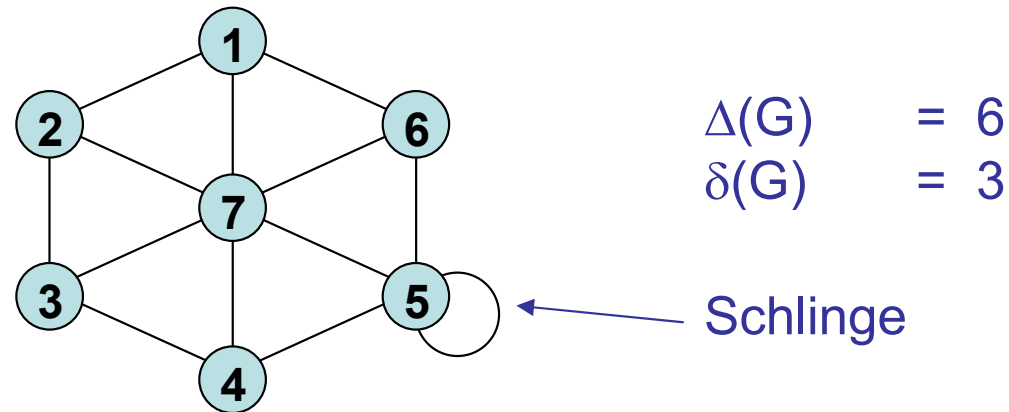
Ausgabe:  
17, 4, 36, 2, 8, 19, 40, 6, 37, 7

### ADT Graph

- Verallgemeinerung von (binären) Bäumen
- Wichtige Struktur in der Informatik
- Zahlreiche Anwendungsmöglichkeiten
  - Modellierung von Telefonnetzen, Versorgungsnetzwerken, Straßenverkehr, ...
  - Layout-Fragen bei elektrischen Schaltungen
  - Darstellung sozialer Beziehungen
  - etc.
- Viele Probleme lassen sich als Graphenprobleme „verkleiden“ und dann mit Graphalgorithmen lösen!

**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  $E \subseteq V \times V$ .



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

Der Grad („degree“) eines Knotens  $v \in V$  ist die Anzahl der zu ihm inzidenten Kanten:  $\deg(v) = |\{ (a, b) \in E : a = v \text{ oder } b = v \}|$ .

Maxgrad von  $G$  ist  $\Delta(G) = \max \{ \deg(v) : v \in V \}$

Mingrad von  $G$  ist  $\delta(G) = \min \{ \deg(v) : v \in V \}$

## Definition

Für  $v_i \in V$  heißt  $(v_0, v_1, v_2, \dots, v_k)$  ein Weg oder Pfad in  $G$ , wenn  $(v_i, v_{i+1}) \in E$  für alle  $i = 0, 1, \dots, k-1$ .

Die Länge eines Pfades ist die Anzahl seiner Kanten.

Ein Pfad  $(v_0, v_1, v_2, \dots, v_k)$  mit  $v_0 = v_k$  wird Kreis genannt.

Distanz  $\text{dist}(u, v)$  von zwei Knoten ist die Länge des kürzesten Pfades von  $u$  nach  $v$ .

Durchmesser  $\text{diam}(G)$  eines Graphes  $G$  ist das Maximum über alle Distanzen:

$$\text{diam}(G) = \max \{ \text{dist}(u, v) : (u, v) \in V \times V \}.$$

Graph ist zusammenhängend, wenn  $\forall u, v \in V$  mit  $u \neq v$  einen Pfad gibt.

$G$  heißt Baum gdw.  $G$  zusammenhängend und kreisfrei.

## Darstellung im Computer

- Adjazenzmatrix A mit  $a_{ij} = \begin{cases} 1 & \text{falls } (v_i, v_j) \in E \\ 0 & \text{sonst} \end{cases}$

Problem:

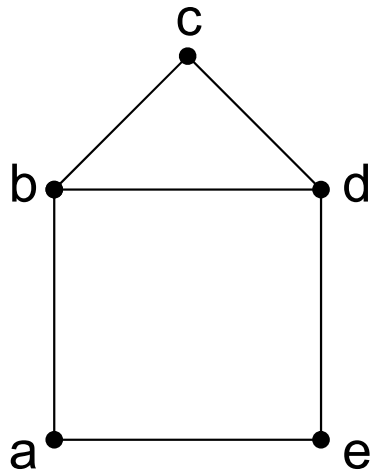
Da  $|E| \leq |V|^2 = n^2$  ist Datenstruktur ineffizient (viele Nullen)  
wenn  $|E|$  verschwindend klein.

- Adjazenzlisten:

Für jeden Knoten v eine (Nachbarschafts-) Liste L(v) mit

$$L(v) = \{ u \in V : (v, u) \in E \}$$

## Beispiel



## Adjazenzlisten

$$L(a) = ( b, e )$$

$$L(b) = ( a, c, d )$$

$$L(c) = ( b, d )$$

$$L(d) = ( b, c, e )$$

$$L(e) = ( a, d )$$

**ADT Liste**

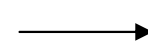
## Adjazenzmatrix

	a	b	c	d	e
a	0	1	0	0	1
b	1	0	1	1	0
c	0	1	0	1	0
d	0	1	1	0	1
e	1	0	0	1	0

**Array[ ][ ]**

## Mögliche Funktionalität

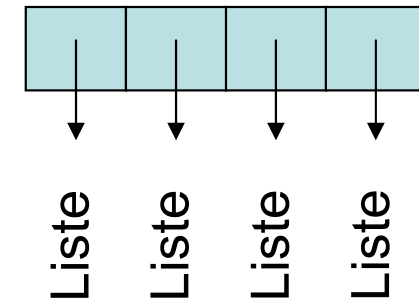
```
typedef unsigned int uint;
```



```
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 *mAdjList;  
};
```

mAdjList: Array  
von Zeigern auf  
Liste



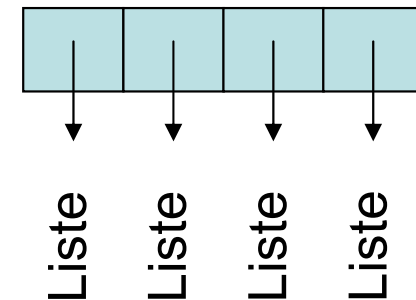
```
#include <iostream>
#include "Graph.h"
using namespace std;

Graph::Graph(uint NoOfNodes) {
    mNoOfNodes = NoOfNodes;
    if (mNoOfNodes > 0)
        mAdjList = new Liste[mNoOfNodes];
}

Graph::~~Graph() {
    if (mNoOfNodes > 0) delete[] mAdjList;
}

void Graph::printGraph() {
    for (uint i = 0; i < mNoOfNodes; i++) {
        cout << i << " : ";
        mAdjList[i].print();
    }
}
```

mAdjList: Array  
von Zeigern auf  
Liste





```
void Graph::addEdge(uint Node1, uint Node2) {
    if (!hasEdge(Node1, Node2)) {
        mAdjList[Node1].append(Node2);
        mAdjList[Node2].append(Node1);
    }
}

bool Graph::hasEdge(uint Node1, uint Node2) {
    if (mNoOfNodes < 1) return false;
    return mAdjList[Node1].is_elem(Node2);
}

uint Graph::noOfEdges() {
    uint cnt = 0;
    for (uint i = 0; i < mNoOfNodes; i++)
        cnt += mAdjList[i].size();
    return cnt / 2;
}

uint Graph::noOfNodes() {
    return mNoOfNodes;
}
```

**Ineffizient!**

Speicherung  
redundanter  
Information!

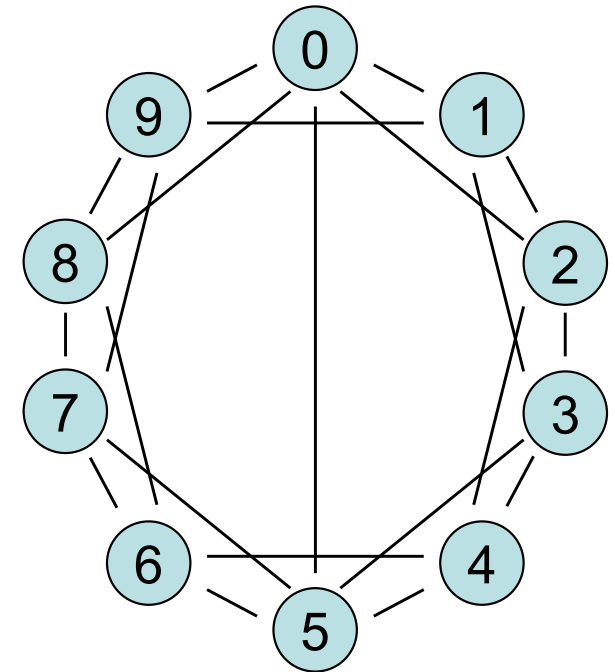
**Ineffizient!**

Falls häufig benutzt,  
dann besser Zähler  
`mNoOfEdges` in  
`class Graph`

## Test

```
#include <iostream>
#include "Graph.h"
using namespace std;

int main() {
    Graph g(10);
    uint n = g.noOfNodes();
    cout << "Knoten: " << n << endl;
    cout << "Kanten: " << g.noOfEdges() << endl;
    for (uint i = 0; i < n; i++)
        g.addEdge(i, (i+1) % n);
    for (uint i = 0; i < n; i++)
        g.addEdge(i, (i+2) % n);
    g.addEdge(5,0);
    if (g.hasEdge(0,5))
        cout << "Kante (0,5) existiert" << endl;
    g.printGraph();
    cout << "Kanten: " << g.noOfEdges() << endl;
    return 0;
}
```



```
d:\Code\EINI\debug\CGraph.exe
Knoten: 10
Kanten: 21
Kante (0,5) existiert
0 : 1 9 2 8 5
1 : 0 2 3 9
2 : 1 3 0 4
3 : 2 4 1 5
4 : 3 5 2 6
5 : 4 6 3 7 0
6 : 5 7 4 8
7 : 6 8 5 9
8 : 7 9 6 0
9 : 8 0 7 1
Kanten: 21
```

## Verbesserungsmöglichkeiten

- Überprüfung, ob Knotennummer zulässig ( $<$  Anzahl Knoten) !
- Zähler `mNoOfEdges` → wird erhöht, wenn neue Kante eingefügt wird
- Kanten sind bidirektional → nur einmal speichern!  
→ erfordert Anpassung in einigen Methoden!

```
void Graph::addEdge(uint Node1, uint Node2) {
    if (Node1 > Node2) Swap(&Node1, &Node2);
    if (!hasEdge(Node1, Node2))
        mAdjList[Node1].append(Node2);
}

bool Graph::hasEdge(uint Node1, uint Node2) {
    if (mNoOfNodes < 1) return false;
    if (Node1 > Node2) Swap(&Node1, &Node2);
    return mAdjList[Node1].is_elem(Node2);
}
```

### Idee:

Normierung, so  
dass kleinere  
Knotennummer  
zuerst

- Funktionalität erweitern: Hinzufügen Knoten; Löschen Knoten / Kanten, etc.