technische universität dortmund

Einführung in die Programmierung

Wintersemester 2020/21

Kapitel 9: Elementare Datenstrukturen (Teil 4)

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

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 einen 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**

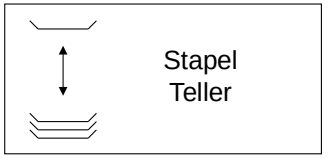
Lineare Datenstrukturen: Keller bzw. **Stapel** (*engl. stack*)

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

Aufräumen:
Kiste in den Keller, **oben** auf Haufen.

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

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



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
};
```

Alternative: anonymer enum („*the enum trick*“)

```
  enum { maxSize = 100 };
```


Implementierung: (Version 1)

```
template<typename T>
Stapel<T>::Stapel() {
  sz = -1;
}
```

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

Idee:
unzulässiger Arrayindex -1 kennzeichnet leeren Stapel

Problem: Arraygrenzen!

Wann können Probleme auftreten?

Bei **pop**, falls Stapel leer ist:

→ Stapelzeiger 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 (erlaubt: 0 bis maxSize – 1)

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

```
void error(char const *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**)

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


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

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

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

```
template<typename T>
void Stapel<T>::error(char const * info) {
  std::cerr << info << std::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<int> 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*)

FIFO:
First in, first out.

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

Schlange an der Supermarktkasse:

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

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)

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

Klassendefinition: (Version 1; schon mit Fehlerbehandlung)

```
template<typename 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 const *info);  // Fehlermeldung
};
```


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

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

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

```
template<typename T>
void Schlange<T>::error(char const *info) {
  std::cerr << info << std::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<int> 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**

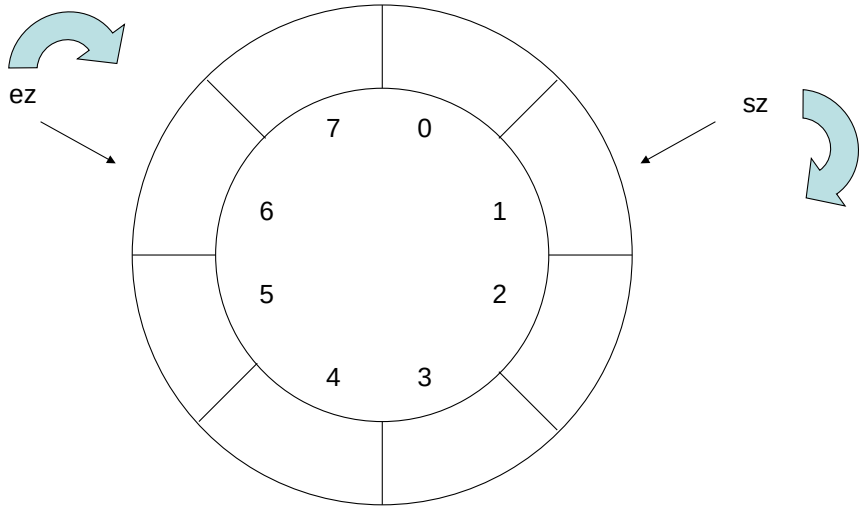
```
template<typename T>
void Schlange<T>::deq() {
  if (empty()) error("leer");
  for (int i = 0; i < ez; i++)
    data[i] = data[i+1];
  ez--;
}
```

ez = Anzahl Elemente in Schlange

Insgesamt **ez** Datenverschiebungen

Worst case: (**maxSize** – 1) mal

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



Implementierung: (Version 2; mit Ringspeicher)

```
template<typename 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 const *info);
};
```


Implementierung: (Version 2; mit Ringspeicher)

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

Implementierung: (Version 2; mit Ringspeicher)

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

Laufzeit:

unabhängig von Größe der Schlange

```
template<typename T>
void Schlange<T>::deq() {
  if (empty()) 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 statischem 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 dynamischem Speicher:

Wir können bei der Realisierung der Schlange
statt statischem (Array) nun **dynamischen Speicher** verwenden …

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

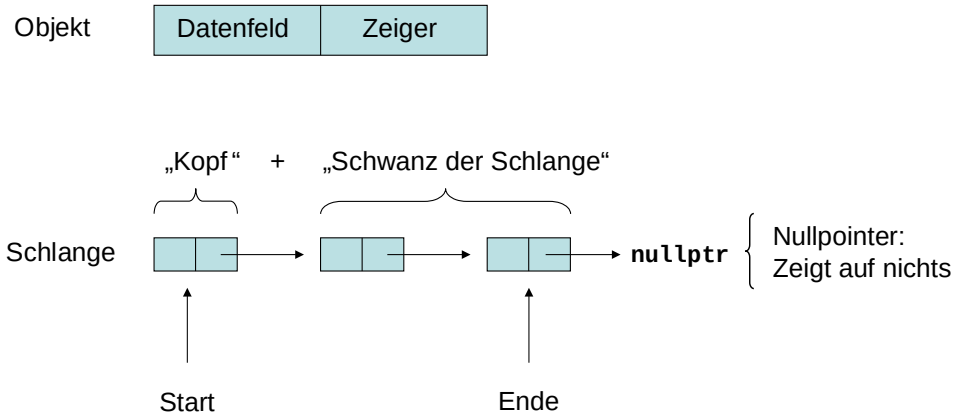
→ Größe kann zwar zur Laufzeit angegeben werden, ist aber dann fixiert!

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 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
};
```


Implementierung: (Version 3)

```
template<typename T>
Schlange<T>::Schlange() {
  ez = sz = nullptr;
}
template<typename T>
T Schlange<T>::front() {
  if (empty()) error("leer");
  return sz->data;
}
template<typename T>
bool Schlange<T>::empty() {
  return (ez == nullptr);
}
template<typename T>
void Schlange<T>::clear() {
  while (!empty()) deq();
}
```

`nullptr` ist der Nullzeiger!

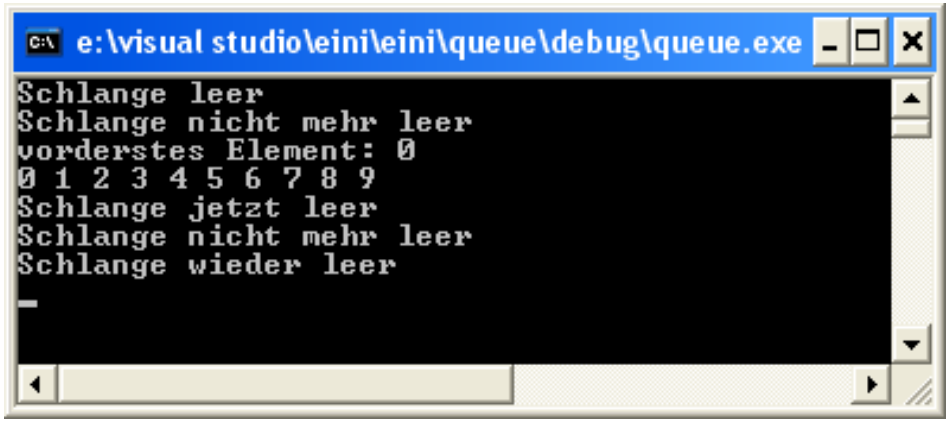
```
template<typename T>
Schlange<T>::~Schlange() {
  clear();
}
template<typename T>
void Schlange<T>::error(
    char const *info){
  cerr << info << endl;
  exit(1);
}
```

Implementierung: (Version 3)

```
template<typename T>
void Schlange<T>::enq(T &x) {
  Objekt *obj = new Objekt; // neues Objekt anlegen
  obj->data = x;            // Nutzdaten speichern
  obj->tail = nullptr;
  if (empty()) sz = obj;    // falls leer nach vorne,
  else ez->tail = obj;      // sonst hinten anhängen
  ez = obj;                 // Endezeiger aktualisieren
}
template<typename T>
void Schlange<T>::deq() {
  if (empty()) error("leer");
  Objekt *obj = sz;         // Zeiger auf Kopf retten
  sz = sz->tail;            // Start auf 2. Element
  if (sz == nullptr) ez = nullptr;  // Schlange leer!
  delete obj;               // ehemaliges 1. Element
}                           //               löschen
```


Testprogramm!

```
int main() {
  Schlange<int> 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 (int 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;
}
```

Kopieren von Klassenobjekten

```
template<typename T>
class Schlange {
  T data[100];
  int sz, ez;
};
```

→

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

statischer Speicher:
byteweises Speicherabbild
⇒ **OK**

```
template<typename
T>
class Schlange {
  struct Objekt {
    Objekt *tail;
    T       data;
  } *sz, *ez;
};
```

→

```
Schlange<int> s1;
for (int i=0;i<10;i++)
  s1.enq(i);
Schlange<int> 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. Ressource) → 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<int> stack1;
  Schlange<int> queue1;
  for (int i = 0; i < 10; i++) {
    stack1.push(i);
    queue1.enq(i);
  }
  Stapel<int> stack2 = stack1;
  Schlange<int> 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*)

```
template<typename T>
class Schlange {
public:
  Schlange();   // Konstruktor
  Schlange(const Schlange<T>& s);
  ~Schlange();  // Destruktor
};
```

Kann wie eine Zuweisung **interpretiert** werden

← **Kopierkonstruktor**

```
template<typename T>
Schlange<T>::Schlange(const Schlange<T>& s){
  ez = nullptr;
  Objekt *ptr = s.sz;
  while (ptr != nullptr) {
    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 als Parameter an eine Funktion übergeben 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**. (Das ist die sogenannte *„Rule of Three“*.)

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 = `operator+ (a, b)`

  Funktionsname — 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?

Objekttyp& operator☝(const ObjektTyp& bezeichner)

Objekttyp operator☝ (const ObjektTyp& bezeichner)

Überladen von Operatoren: Zuweisung

```
template<typename T>
Schlange<T>& Schlange<T>::operator= (const Schlange<T>& s) {
  if (this == &s) return *this;  // falls Selbstzuweisung
  clear(); // Speicher freigeben
  Objekt *ptr = s.sz;
  while (ptr != nullptr) {
    enq(ptr->data);
    ptr = ptr->tail;
  }
  return *this;
}
```

this ist ein Zeiger auf das Objekt selbst.

Bei der Zuweisung wird ja keine neue Instanz erzeugt; tatsächlich wird eine vorhandene Instanz verändert.

Deshalb ist Rückgabewert eine **Referenz auf sich selbst**!

Überladen von Operatoren: Test auf Gleicheit

```
template<typename T>
bool Schlange<T>::operator== (const Schlange<T>& s) {
  if (this == &s) return true;       // Selbstvergleich?
  Objekt *ptr1 = sz;      // this->sz
  Objekt *ptr2 = s.sz;
  while (ptr1 != nullptr && ptr2 != nullptr) {
    if (ptr1->data != ptr2->data) return false;
    ptr1 = ptr1->tail;
    ptr2 = ptr2->tail;
  }
  return (ptr1 == ptr2);
}
```

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

Automatisch erzeugte Methoden *erzwingen* (C++11)

- Schlüsselwort **`default`** ist Anweisung an Compiler, die Standardimplementierung zu erzeugen
- z.B. für den parameterlosen Standard-Konstruktor

```
class Punkt {
private:
  double x, y;
public:
  Punkt() = default;

  Punkt(double ax, double ay) : x(ax), y(ax){}
};
```

Der Compiler soll seine Standardimplementierung verwenden

Automatisch erzeugte Methoden *verhindern* (C++11)

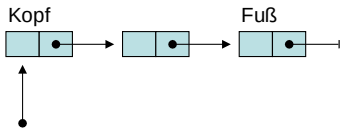
- Schlüsselwort **`delete`** verhindert die Erzeugung von Methoden
- z.B. für Klassen, deren Instanzen nicht kopiert werden können

```
class Punkt {
private:
  double x, y;
public:
  Punkt(const Punkt& p) = delete;

  Punkt& operator=(const Punkt& p) = delete;
};
```

Der Compiler soll keinen Kopierkonstruktor erzeugen

Der Compiler soll keinen Zuweisungsoperator erzeugen

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 Liste (1. Version)

```
template<typename T>
Liste<T>::Liste() {
  sz = nullptr;
}
template<typename T>
void Liste<T>::clear(Objekt *obj) {
  if (obj == nullptr) return;
  clear(obj->next);
  delete obj;
}
template<typename T>
void Liste<T>::clear() {
  clear(sz);
  sz = nullptr;
}
template<typename T>
Liste<T>::~Liste() {
  clear();
}
```

Laufzeit:
unabhängig von Listenlänge

rekursives Löschen von „hinten“ nach „vorne“

Laufzeit:
proportional zur Listenlänge

ADT Liste (1. Version)

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

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

neuen Eintrag erzeugen

Liste leer? → Kopf = neuer Eintrag

iterativer Durchlauf von „vorne“ nach „hinten“

Laufzeit:
proportional zur Listenlänge

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

Laufzeit: quadratisch proportional zur Listenlänge!

ADT Liste (1. Version)

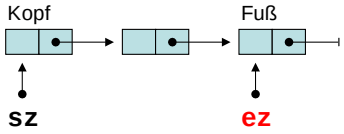
Zusammenfassung:

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)

```
template<typename T>
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 Listenfuß (Ende)



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

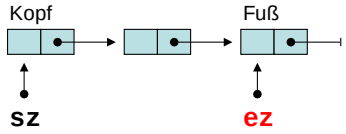
ADT Liste (2. Version)

```
template<typename T>
Liste<T>::Liste() {
  ez = sz = nullptr;
}
```

```
template<typename T>
bool Liste<T>::empty() {
  return (ez == nullptr);
}
```

```
template<typename T>
Liste<T>::~Liste() {
  clear();
}
```

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



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

ADT Liste (2. Version)

```
template<typename T>
void Liste<T>::clear(Objekt *obj) {
  if (obj == nullptr) return;
  clear(obj->next);
  delete obj;
}
template<typename T>
void Liste<T>::clear() {
  clear(sz);
  ez = sz = nullptr;
}
```

keine Änderungen!

Laufzeit:
proportional zur Listenlänge

→ keine Verbesserung (OK)

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

keine Änderungen!

Laufzeit:
proportional zur Listenlänge

→ keine Verbesserung (OK)

ADT Liste (2. Version)

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

keine Änderungen!

Laufzeit:
unabhängig von Listenlänge

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

Laufzeit:
unabhängig von Listenlänge

→ Verbesserung!

ADT Liste (2. Version)

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

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

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

Laufzeit in µsek. für Kopieroperation

Anzahl Elemente mal 4 ⇒

Laufzeit mal 4^2=16 (Version 1)

Laufzeit mal 4 (Version 2)

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 ∈ A ?

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

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

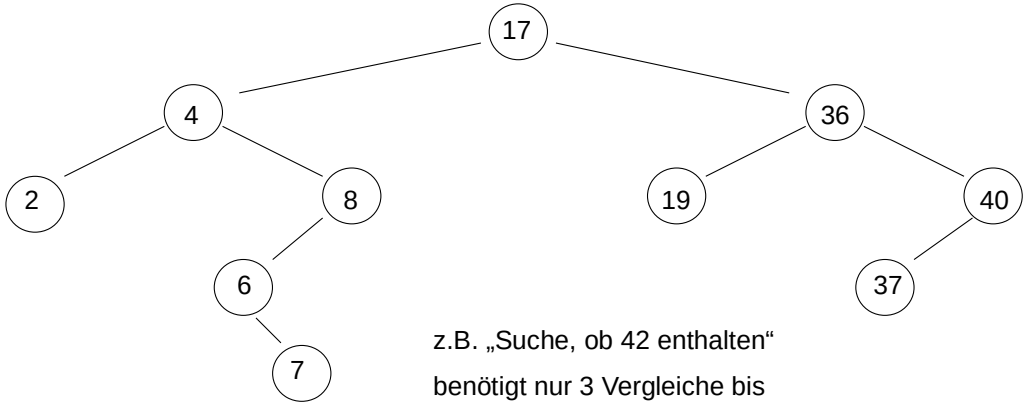
Bessere Lösungen?

ADT Binärer Suchbaum

Beispiel:

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

kleiner	: nach links
größer	: nach rechts



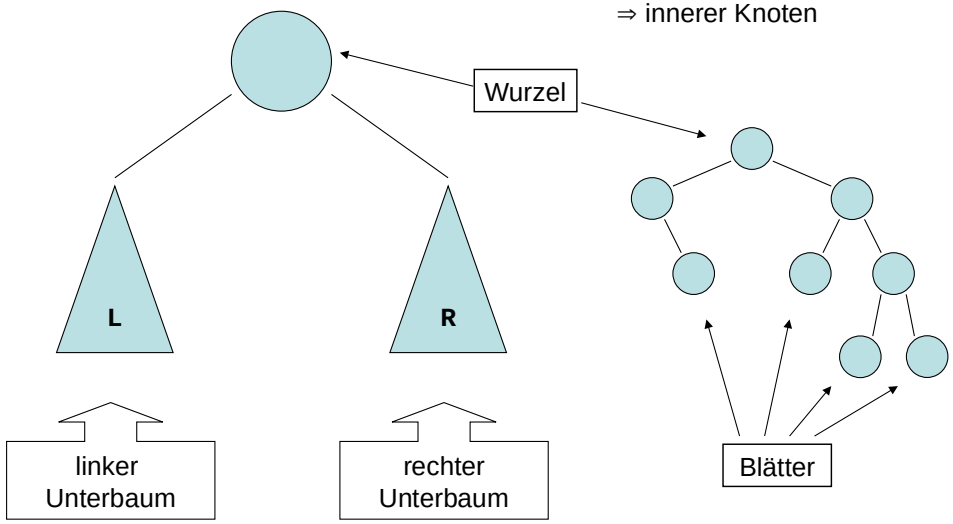
z.B. „Suche, ob 42 enthalten“

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

ADT Binärer Suchbaum: Terminologie

keine Wurzel und kein Blatt

⇒ innerer Knoten



ADT Binärer Suchbaum: Klassendefinition

```
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:

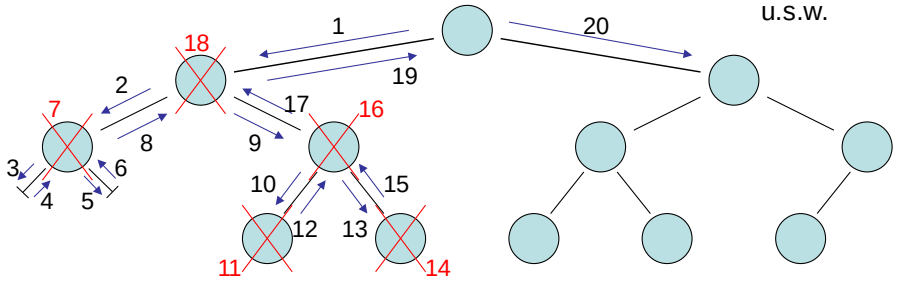
Falls kein Erfolg im aktuellen Knoten, dann Frage an den Unterbaum weiterreichen, der das Element enthalten müsste.

Falls Knoten Element enthält: Erfolg!

Falls Unterbaum leer, dann Element nicht vorhanden.

ADT Binärer Suchbaum: Aufräumen

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



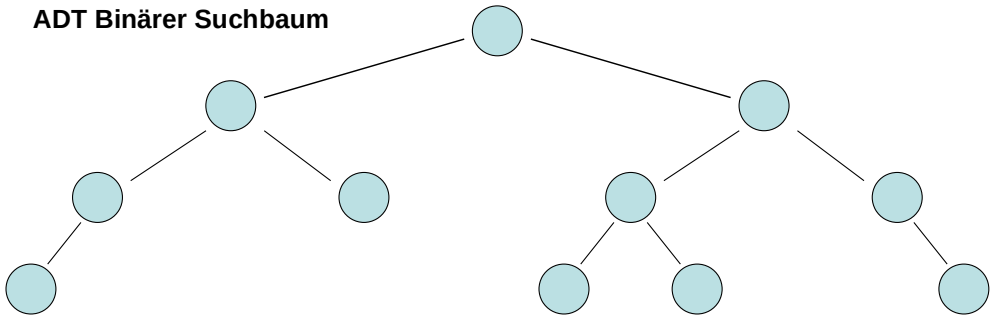
ADT Binärer Suchbaum: Einfügen

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

Nötig, wenn Rückgabewert ein lokaler Typ der Klasse ist. (ISO-Norm)

Rekursives Einfügen

ADT Binärer Suchbaum



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

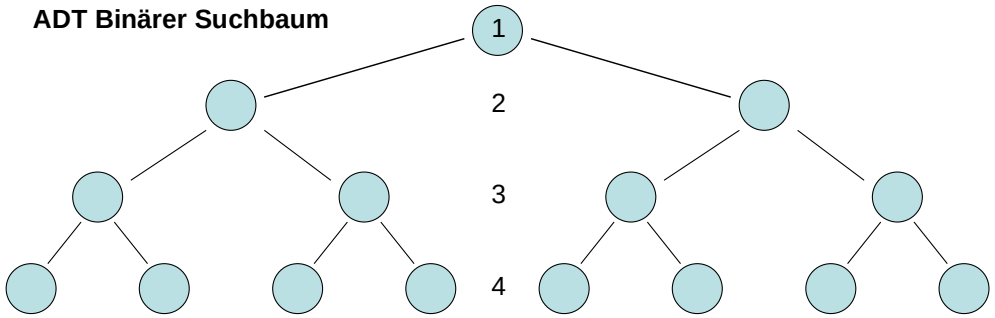
Höhe(leerer Baum) = 0

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

Anmerkung: rekursive Definition!

(U-Baum = Unterbaum)

ADT Binärer Suchbaum

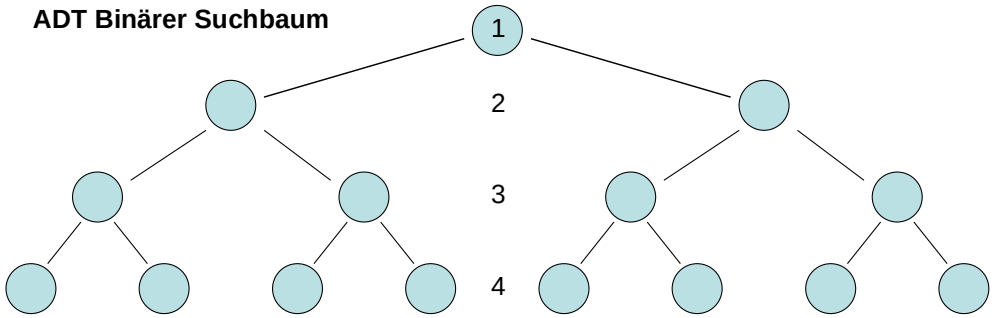


Auf Ebene k können jeweils zwischen 1 und 2^{k-1} Elemente gespeichert werden.

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

$$\sum_{k=1}^{h} 2^{k-1} = 2^h - 1$$ 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“) z.B. `.txt` oder `.asc` (oder `.cpp`)
  - 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>
```

- Eingabe-Datei = **i**nput **f**ile

```
ifstream Quelldatei;
```

Datentyp Bezeichner

- Ausgabe-Datei = **o**utput **f**ile

```
ofstream Zieldatei;
```

Datentyp Bezeichner

- Öffnen der Datei:

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

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

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

- Öffnen der Datei:

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

ist Kurzform von
Zieldatei.open(dateiName, modus);

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

modus:

`ios::binary`	binäre Datei
`ios::in`	öffnet für Eingabe (implizit bei ifstream)
`ios::out`	öffnet für Ausgabe (implizit bei ofstream)
`ios::app`	hängt Daten am Dateiende an

(weitere: ios::ate, ios::trunc)

Man kann diese Modi mit dem bitweisen Oder-Operator | miteinander kombinieren:

`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 intern Flag gesetzt → mit
  `file.is_open()`prüfen!
- **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** (in ofstream)
  `file.put(c);` schreibt ein Zeichen
  `file << x;` schreibt verschiedene Typen

Merke:

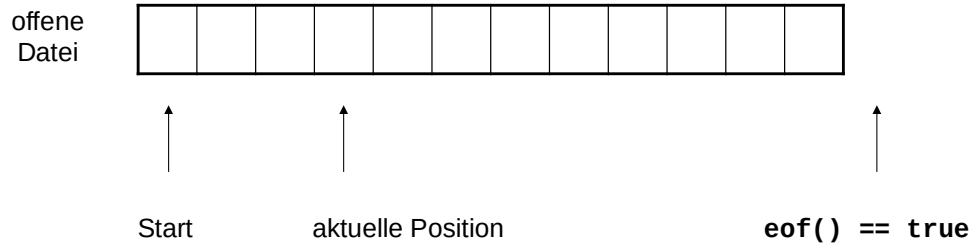
1. Auf eine geöffnete Datei darf **immer nur einer** zugreifen.
2. Eine geöffnete Datei belegt Ressourcen des Betriebssystems.
   ⇒ Deshalb Datei **nicht länger als nötig geöffnet** halten.
3. Eine geöffnete Datei unbekannter Länge kann solange gelesen werden, bis das Ende-Bit (*end of file*, EOF) gesetzt wird.
4. Der Versuch, eine nicht vorhandene Datei zu öffnen (zum Lesen) oder eine schreibgeschützte Datei zu öffnen (zum Schreiben), führt zum Setzen eines **Fehlerzustandes** im `fstream`-Objekt.
   ⇒ 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();

  return 0;
}
```



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

wir 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“ (aneinanderhängen)

```
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 {
  T data;
  unsigned int cnt;
  BinTree *left, *right;
};
```

gelesenes Wort
wie oft gelesen?

zusätzlicher Konstruktor (zum Einlesen der Datei)

```
template<typename T>
BinTree<T>::BinTree(string& filename)
  : BinTree() {
  ifstream source;
  source.open(filename.c_str());
  if (!source.is_open()) exit(1);
  T s;
  while (!source.eof()) {
    source >> s;
    insert(s);
  }
  source.close();    }
```

Delegation an Default-Konstruktor: Wurzel-Knoten initialisieren

Datei öffnen

Worte einzeln auslesen und im Baum einfügen

Datei schließen

Einfügen (Änderungen in rot)

```
template<typename T>
typename BinTree<T>::Node *BinTree<T>::insert(Node *node, T
key) {
  if (node == nullptr) {
    node = new Node;
    node->data = key;
    node->cnt  = 1;
    node->left = node->right = nullptr;
    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)

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

Dies ist die **Inorder**-Ausgabe.

Präorder:

```
cout …;
print(…);
print(…);
```

Postorder:

```
print(…);
print(…);
cout …;
```


Hauptprogramm:

```
#include <string>
#include "BinTree.h"
using namespace std;

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


Durchlaufstrategien:

z.B. Ausdruck des Knotenwertes

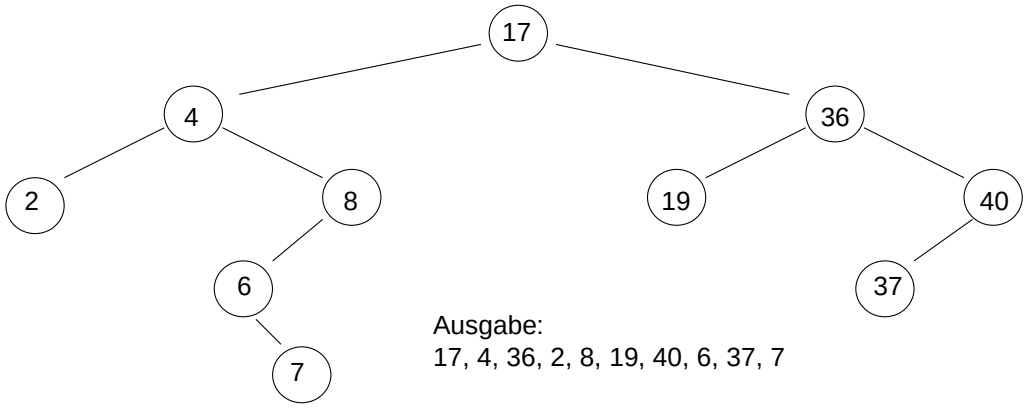
- **Tiefensuche** („depth-first search“, DFS)
  - Präorder
    Vor (*prä*) Abstieg in Unterbäume die „Knotenbehandlung“ durchführen
  - Postorder
    Nach (*post*) Abstieg in bzw. Rückkehr aus Unterbäumen die „Knotenbehandlung“ durchführen
  - Inorder
    Zwischen zwei Abstiegen „Knotenbehandlung“ durchführen

- **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, 40, 19, 6, 7, 37



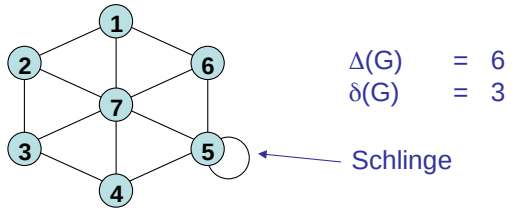
Ausgabe:
17, 4, 36, 2, 8, 19, 40, 6, 37, 7

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 $E \subseteq V \times V$.



$\Delta(G) = 6$
$\delta(G) = 3$

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: $deg(v) = | \{ (a, b) \in E : a = v \text{ oder } b = v \} |$. Schlingen zählen doppelt.

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

Die **Länge** eines Pfades ist die Anzahl seiner Kanten.

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

Distanz dist(u, v) von zwei Knoten ist die Länge des kürzesten Pfades von u nach v.

Durchmesser diam(G) eines Graphes G ist das Maximum über alle Distanzen:

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

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:

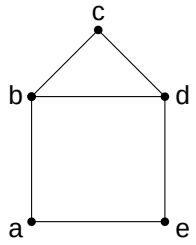
Da $| E | \leq | V |^2 = n^2$ ist Datenstruktur ineffizient (viele Nullen) wenn | E | verschwindend klein.

- **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

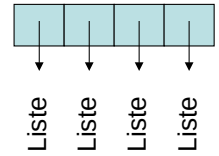
Array[][]

Mögliche Funktionalität

```
typedef unsigned int uint;
```
⟶ **typedef** Datentyp TypName;

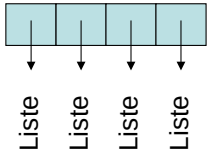
```
class Graph {
public:
  Graph(uint NoOfNodes);
  void addEdge(uint Node1, uint Node2);
  bool hasEdge(uint Node1, uint Node2);
  uint noOfEdges();
  uint noOfNodes();
  void printGraph();
  ~Graph();
private:
  uint mNoOfNodes;
  Liste<uint> *mAdjList;
};
```

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




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

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




```
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!

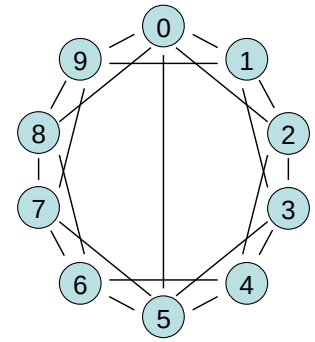
Speicherung redundanter Information!

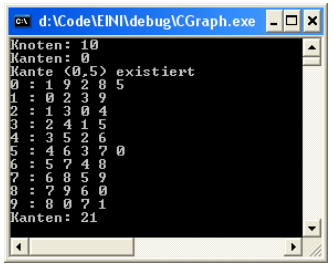
Ineffizient!

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;
}
```





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.