Karlsruher Institut für Technologie

Institut für Technik der Informationsverarbeitung

Institutsleitung
Prof. Dr.-Ing. K. D. Müller-Glaser
Prof. Dr.-Ing. J. Becker
Prof. Dr. rer. nat. W. Stork

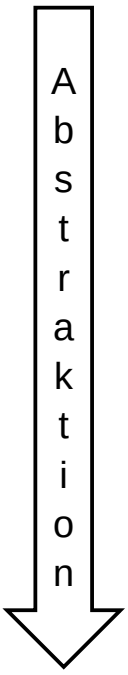
Vorlesung: Informationstechnik (IT)

Prof. Dr.-Ing. K.D. Müller-Glaser

Institut für Technik der Informationsverarbeitung (ITIV)

Datenstrukturen

Abstraktion der Datenspeicherung

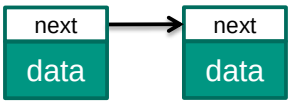


001000100100
101000100010

Daten im Speicher abgelegt als Nullen und Einsen in Speicherzellen

double a;
int b;
float arr[10];

Im Sprachumfang einer Programmiersprache werden Daten als Variablen eines elementaren Datentyps realisiert



Datenstrukturen organisieren Daten unabhängig vom Datentyp in definierter Art und Weise

Die Sicht eines Programmierers auf die im Programm verarbeiteten Daten ist bereits abstrahiert. Eine normale Hochsprache stellt Variablen bereit, welche einen Datentyp besitzen und sich somit Daten in bestimmten Formaten zuweisen lassen. Oftmals ist diese Art der Datenbehandlung nicht sehr effizient, weder bei der Programmierung noch während der Ausführung. Deshalb kann entweder mit selbstdefinierten Datentypen (z.B.: Klassen) oder mit definierten Datenstrukturen das Level der Abstraktion erhöht werden.

Beispiel

- Programmierung einer Datenbank für Personen:

```
...
char[100][20] namen;
int alter[100];
...
```

Mehr Übersichtlichkeit durch Verwendung eines selbst definierten Datentyps

```
...
class Person { ... };
Person personen[100];
...
```

Unbeschränkte Anzahl Personen speicherbar durch Verwendung einer Datenstruktur

```
...
class Person { ... };
list<Person> personen;
...
```


Formale Definition

- Eine Datenstruktur ist eine Methode, um Daten strukturiert abzuspeichern und zu organisieren sowie den Zugriff auf die Daten und die Modifikation der Daten zu erleichtern.

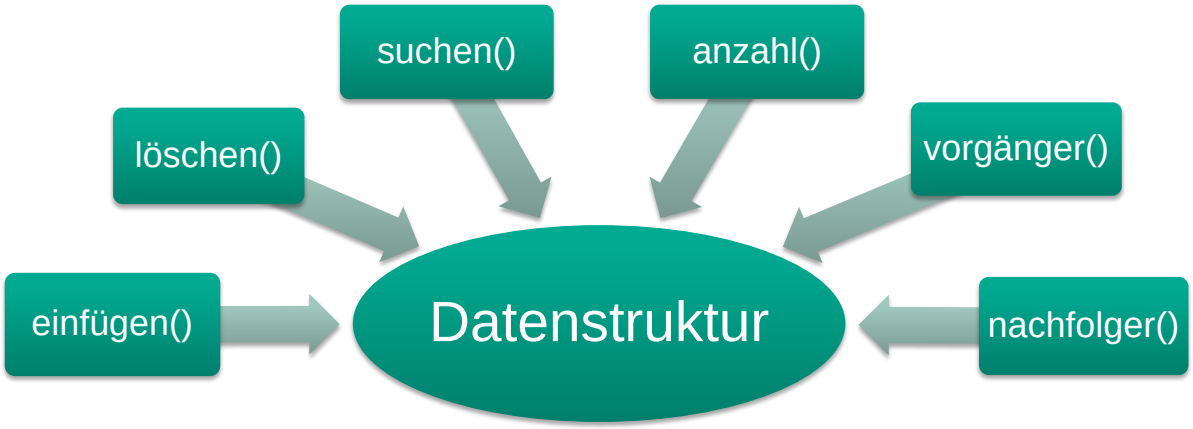
- In der Informatik ist eine Datenstruktur ein mathematisches Objekt zur Speicherung von Daten.

Algorithmen und Datenstrukturen

- Nach moderner Auffassung sind Datenstrukturen ein Bestandteil von Algorithmen.
- Bei vielen Algorithmen hängt der Ressourcenbedarf, also sowohl die benötigte Laufzeit als auch der Speicherplatzbedarf, von der Verwendung geeigneter Datenstrukturen ab.

Für Algorithmen ist entscheidend mit welchen Datenstrukturen diese operieren, da dies die Ausführgeschwindigkeit und Komplexität beeinflusst. Spricht man von einem Algorithmus, ist damit auch eine Datenstruktur gemeint.

Operationen auf Datenstrukturen



- Standard-Operationen auf Datenstrukturen
- Verschiedene Datenstrukturen sind unterschiedlich effizient für diese Operationen
- Bieten die Möglichkeit Datenstrukturen zu klassifizieren

Containerklassen und STL

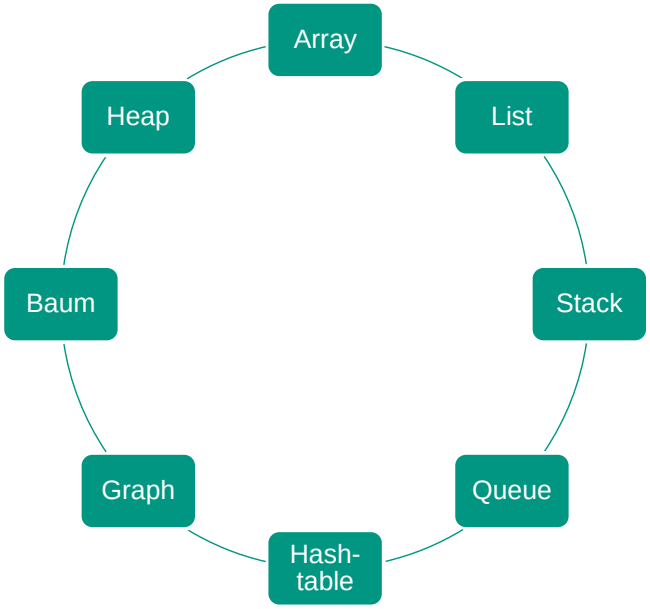
- Realisierung der Datenstrukturen in C++ üblicherweise als Klasse

- Standard Template Library (STL) ist eine gängige Klassenbibliothek
- Die STL umfasst:
  - Implementierungen gängiger Datenstrukturen
  - Klassen zur Ein- und Ausgabe, zum Beispiel für Dateien
  - Gängige Algorithmen zum Sortieren oder ähnlichem

- STL bietet datentyp-unabhängige „Container“ dank Templates
- Zur Arbeit mit Datentypen bietet die STL sogenannte Iteratoren
- Iteratoren sind Klassen, die wie ein erweiterter Zeiger arbeiten

Bekannte Datenstrukturen

- Übersicht über bekannte Datenstrukturen:
  (für imperative Programmierung)



Bekannte elementare Datenstrukturen

- In höheren Programmiersprachen
  C, C++ und andere

- short, int, long
- float, double, long double
- bool
- char
- string (nur für C++)

- Erweiterte Datenstruktur
  - Array
  - Struct
  - Class
  - …

Struct Gatter

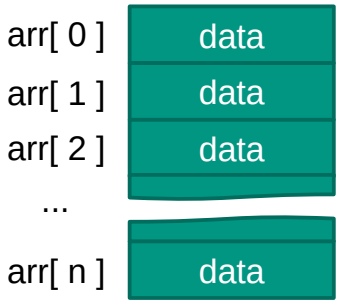
```
struct gatter {
  int index;
  string typ;
  float grundlaufzeit;
};

gatter g001;
g001.index = 0;
g001.grundlaufzeit = 70.0;
```

Struct Gatter

```
INDEX: 0
Typ: and2
grundlaufzeit: 70.0
lastfaktor: 1700
lastkapazitaet: 3
isFlipflop: false
setuptime:
holdtime:
lastkapazitaet_clock:
laufzeit:
```

Array (Feld)

- einfachste verwendete Datenstruktur
- Speichert mehrere Variablen des selben Datentyps
- Schneller Zugriff auf Elemente über Index
- Nachteil: keine dynamische Größenanpassung
- Operationen:
  - Indiziertes Lesen
  - Indiziertes Speichern



Array (Feld):

Das Array ist die einfachste verwendete Datenstruktur. Es werden hierbei mehrere Variablen vom selben Basisdatentyp gespeichert. Ein Zugriff auf die einzelnen Elemente wird über einen Index möglich. Die einzigen notwendigen Operationen sind das indizierte Speichern und das indizierte Lesen, die auf jedes Element des Arrays direkt zugreifen können (eindimensional Vektor, zweidimensional Tabelle oder Matrix, n-dimensional).

Array in C++ Code

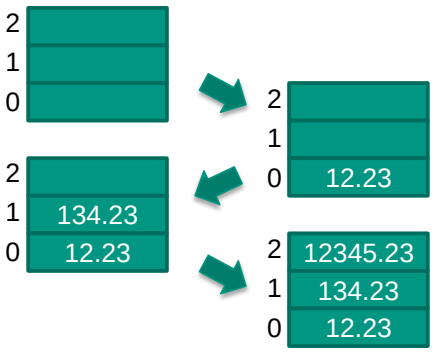
Array erzeugen: Datentyp arr[Anzahl];

Wert speichern: arr[index] = Wert;

Wert lesen: variable = arr[index];

Beispiel: Array

```
#include <iostream>

int main() {
    double meinArray[3];

    meinArray[0] = 12.23;

    meinArray[1] = 134.23;

    meinArray[2] = 12345.23;


    cout << meinArray[0] << endl;
    cout << meinArray[1] + meinArray[2] << endl;

    return 0;
}
```



Ausgabe:

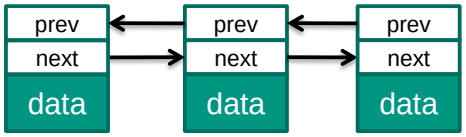
```
12.23
12479.46
```

List (Liste)

- Dynamische Speicherung beliebig vieler Elemente
- Nachteil: Langsamerer Zugriff als bei Array, kein indizierter Zugriff
- Operationen:
  - suchen() und sortieren()
  - einfügen() und löschen()
  - vorgänger() und nachfolger()

- Verkettung der Elemente untereinander
- Elemente enthalten den Verweis auf Nachfolger (und Vorgänger)

einfach verkettete Liste



doppelt verkettete Liste

NIL oder **/** : not in list, Nullwert, nicht vorhanden. Zeigt das Fehlen eines gültigen Wertes an. Wesentliche semantische Ergänzung.

Liste:

Die Liste ist eine Datenstruktur zur dynamischen Speicherung von beliebig vielen Objekten. Dabei beinhaltet jedes Listenelement als Besonderheit einen Verweis auf das nächste Element, wodurch die Gesamtheit der Objekte zu einer Verkettung von Objekten wird. Die zu einer Liste gehörenden Operationen sind typischerweise Element suchen, einfügen, löschen, Vorgänger/Nachfolger. Listen sind stets linear.

Elemente können sehr schnell am Anfang der Liste eingefügt werden. Im Gegensatz zu einem Array ist das Einfügen problemlos möglich, ohne dass der komplette Datensatz bei jeder Vergrößerung umkopiert werden muss.

Listen Typen:

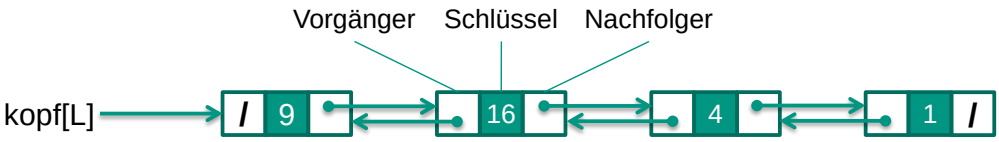
Man unterscheidet grundsätzlich zwischen **einfach** und **doppelt verketteten Listen**.

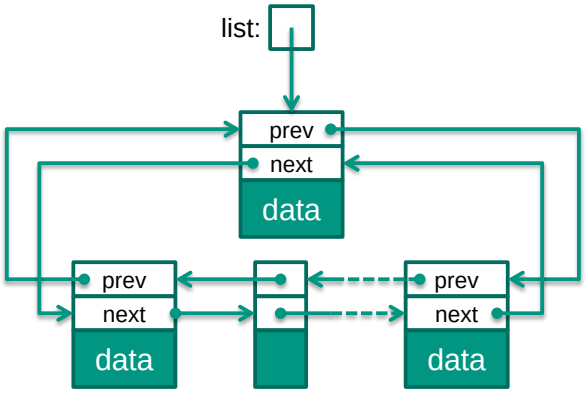
Einfach verkettete Liste :

Sie ist die einfachste Form der verketteten Liste, da sie neben den einzelnen Knoten nur einen "Start"-Zeiger besitzt. Dieser zeigt auf den ersten Knoten in der Liste (roter Pfeil). Der letzte Knoten in der Liste zeigt auf den Nullwert (hier *NIL* für Not In List). Als Nullwert (kurz NULL oder NIL) bezeichnet man in der Informatik einen Wert, der das Fehlen eines gültigen Wertes anzeigen soll. Er steht für „nicht vorhanden“ und darf nicht mit dem Zahlenwert 0 verwechselt werden. Die Vereinbarung eines undefinierten Werts ist eine wesentliche semantische bzw. pragmatische Ergänzung der Informatik (bzw. Angewandten Mathematik) gegenüber der theoretischen Mathematik.

List Listenkopf, NIL

NIL oder **/** : not in list, Nullwert, nicht vorhanden.
Zeigt das Fehlen eines gültigen Wertes an.
Wesentliche semantische Ergänzung.





Zyklische Liste

Doppelt (mehrfach) verkettete Liste:

Im Gegensatz zur einfach-verketteten Liste hat jedes Element sowohl einen Zeiger auf das nachfolgende als auch auf das vorhergehende Element. Der *Vorgänger-Zeiger* des ersten Elementes zeigt auf ein Dummy-Element **Vorteile**: Die Elemente in der zweiten Hälfte einer sortierten Liste sind schneller aufzufinden. Fehlerhafte Verweise können erkannt werden und ein schnelles Löschen und Einfügen von Elementen ist möglich. Das macht zum Beispiel das effiziente Sortieren der Liste möglich. Über die Liste kann von hinten nach vorne iteriert werden.

Nachteile: Höherer Speicherbedarf für die zusätzlichen Zeiger, komplexere Handhabung.

(NULL), wie auch der *Nachfolger-Zeiger* des letzten Elementes. Dieses besondere Element dient zum Auffinden des Anfangs und des Endes einer doppelt verketteten Liste.

Ein Sonderfall stellt die zyklische Liste (Ringliste) dar, die ohne Null-Zeiger auskommt.

Vergleich mit anderen Datenstrukturen

Im Gegensatz zu Arrays müssen die einzelnen Speicherzellen nicht nacheinander im Speicher abgelegt sein, es kann also nicht mit einfacher Adress-Arithmetik gearbeitet werden, sondern die Speicherorte müssen absolut referenziert werden. Im Gegensatz zu Bäumen sind Listen linear, d.h. ein Element hat genau einen Nachfolger und einen Vorgänger.

List in C++ Code

STL-Header:	#include <list>
Liste erzeugen:	list<Datentyp> myList;

Wert speichern:	myList.push_front(Wert);
	myList.push_back(Wert);
	myList.insert(iterator, Wert);

Wert löschen:	myList.erase(iterator);
Wert lesen:	variable = myList.back(); variable = myList.front();

Iterator: ermöglicht das Positionieren und Durchlaufen von Listen (allgemeiner von Containern).

Jedes Objekt in einer Liste besitzt eine bestimmte Position, in der es abgelegt ist.

Um mit den Objekten einer Liste arbeiten zu können, müssen die Positionen der Objekte in der Liste erreichbar sein.

Es braucht also einen Mechanismus, der es erlaubt:

- an jeder Position auf das entsprechende Objekt lesend und / oder schreibend zuzugreifen
- von der Position eines Objekts zur Position des nächsten Objekts in der Liste zu wechseln.

Ein solcher Mechanismus ist bereits von den Zeigern her bekannt (Operatoren: * lesender Elementzugriff, ++ Iterator weitersetzen, == und != zwei Iteratoren vergleichen)

Es sind 5 Iterator-Kategorien definiert:

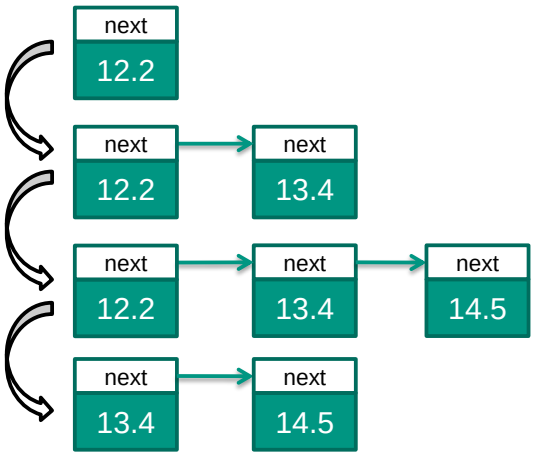
Input-Iterator	nur lesender Elementzugriff, nur vorwärts
Output-Iterator	nur schreibender Elementzugriff, nur vorwärts
Forward-Iterator	lesender und schreibender Elementzugriff, nur vorwärts
Bidirektionaler Iterator	lesender und schreibender Elementzugriff, vorwärts und rückwärts
Random-Access-Iterator	lesender und schreibender Elementzugriff, beliebig posititonierbar

Container speichern Objekte gleichen Typs und bieten Operatoren zur Verwaltung dieser Objekte

die Standardbibliothek in C++ definiert drei „sequentielle container“: Container-Klassen vector, list (double linked) und deque (double ended Queue)

Beispiel: List

```
#include <iostream>
#include <list>

int main() {
    list<double> myList;

    myList.push_back( 12.2 );
    myList.push_back( 13.4 );
    myList.push_back( 14.5 );

    cout << myList.front() << endl;

    myList.erase( myList.begin() );
    cout << myList.front() + myList.back() << endl;

    return 0;
}
```



Ausgabe:

```
12.2
27.9
```

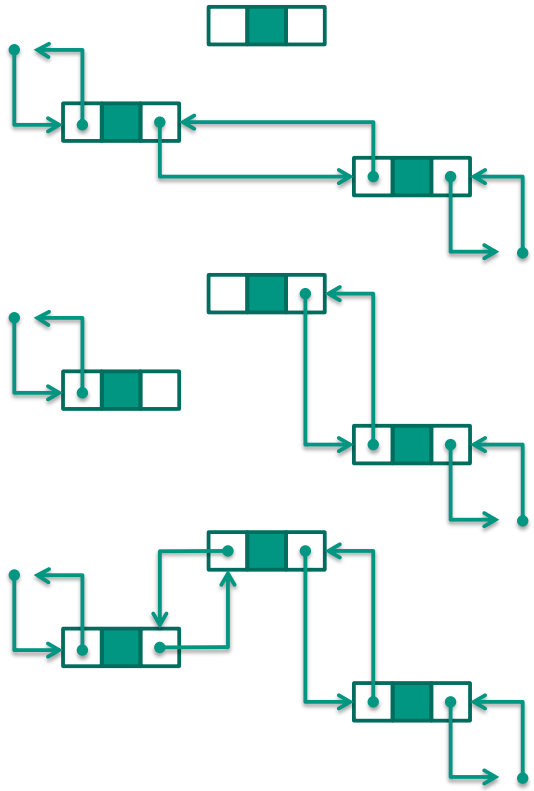

Vorteile:

Elemente können sehr schnell am Anfang der Liste eingefügt werden und sehr geringer Speicherbedarf. Im Gegensatz zu einem Array ist das Einfügen problemlos möglich, ohne dass der komplette Datensatz bei jeder Vergrößerung umkopiert werden muss.

Nachteile:

Es ist aufwändig, nach Daten zu suchen und die Liste zu sortieren, da über jedes einzelne Element gegangen (*iteriert*) werden und das Einfügen an der ersten und der letzten Stelle gesondert behandelt werden muss.

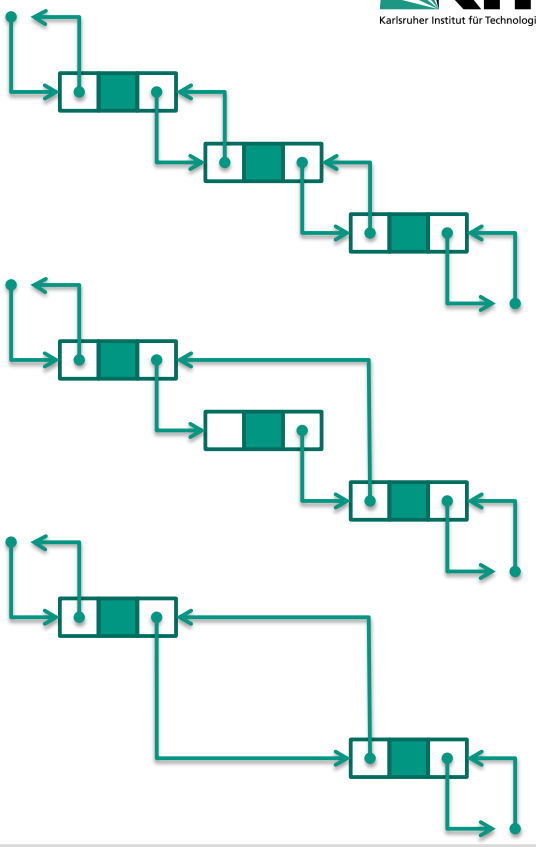
List: einfügen



- Neues Element im Speicher erzeugen
- Folgezeiger des Vorgängerelements auf neues Element zeigen lassen
- Folgezeiger des neuen Elements auf Nachfolgerelement setzen
- In die andere Richtung analog verfahren

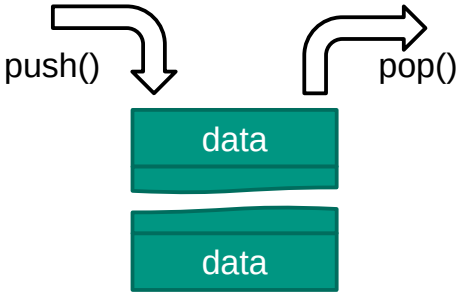
List: löschen

- Zeiger ändern, sodass zu löschendes Element „übersprungen“ wird
- Evtl. Speicher des gelöschten Elements freigeben



Stack (Stapelspeicher)

- LIFO-Prinzip (Last In, First Out): Lesen nur in umgekehrter Reihenfolge wie beim Schreiben
- Speichert beliebig viele Elemente, da intern oft als Liste realisiert
- Operationen:
  - push() – Daten auf den Stapel legen
  - pop() – die zu oberst liegenden Daten holen



Stack (Stapelspeicher):

LIFO-Prinzip (Last-In First-Out). In einem Stack kann eine beliebige Anzahl von Objekten gespeichert werden, jedoch können die gespeicherten Objekte nur in umgekehrter Reihenfolge wieder gelesen werden. Ein Stapelspeicher wird gewöhnlich als Liste implementiert.

Stack in C++ Code

STL-Header:	#include <stack>
Stack erzeugen:	stack<Datentyp> myStack;
Wert speichern:	myStack.push(Wert);
Wert löschen:	myStack.pop();
Wert lesen:	variable = myStack.top();

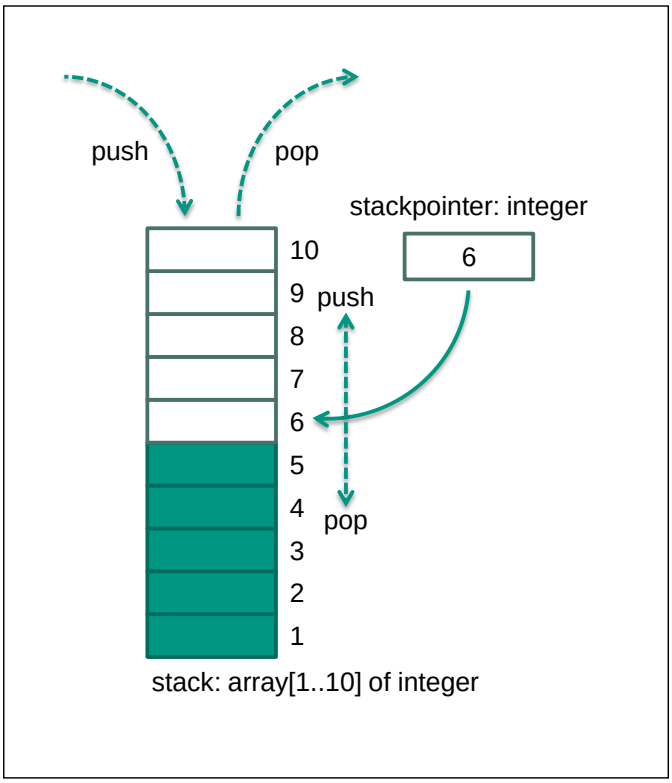
Zwischenübung 01: Stack

```
#include <iostream>
#include <stack>

int main() {
    stack<double> myStack;

    myStack.push( 12.2 );
    myStack.push( 13.4 );
    myStack.push( 14.5 );

    cout << myStack.top() << endl;

    myStack.pop();

    cout << myStack.top() << endl;

    return 0;
}
```

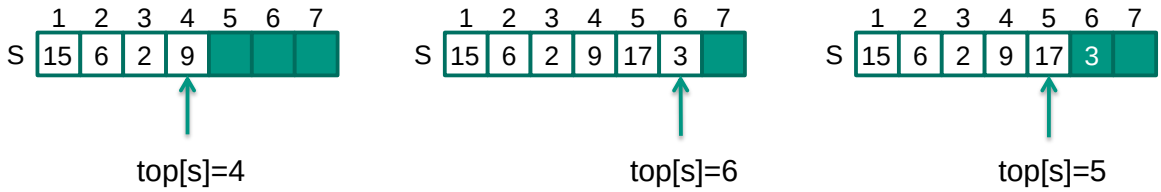
- Überlegen Sie, wie myStack nach jeder Anweisung aussieht.
- Was gibt das Programm aus?

Stack (Stapelspeicher als Array)

- Ein Zeiger zur Zeigerverwaltung



Stack



top[s]: Stackpointer

Aufrufe: push(s,17) und push(s,3)
pop(s)

Achtung: nur Stackpointer wird versetzt,
alte Elemente („3“) werden nicht explizit gelöscht

Die Implementierung eines Stapels mithilfe eines Feldes. Stapelelemente befinden sich nur in den schwach schattierten Positionen.

a) Der Stapel hat 4 Elemente, das oberste Element ist 9.

b) Der Stapel S nach den Aufrufen push(s, 17) und push(s, 3).

c) Der Stapel nachdem der Aufruf pop(s) das Element 3 zurückgegeben hat.

Dies ist zugleich das letzte abgelegte Element. Obwohl Element 3 noch im Feld vorkommt, ist es nicht mehr im Stapel enthalten. Das oberste Element ist der Wert 17.

Queue (Warteschlange)

- FIFO-Prinzip (First In, First Out): Lesen nur in Reihenfolge wie beim Schreiben
- Speichert beliebig viele Elemente, da intern oft als Liste realisiert
- Operationen:
  - enqueue() – hinten in die Schlange einreihe
  - dequeue() – vorderstes Element der Schlange holen

Queue (Warteschlange):

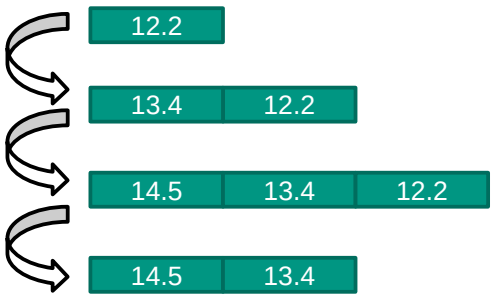
FIFO Prinzip (First-In First-Out): in einer Queue kann eine beliebige Anzahl von Objekten gespeichert werden. Die gespeicherten Objekte können nur in der gleichen Reihenfolge wieder gelesen werden, wie sie gespeichert wurden. Eine Warteschlange wird gewöhnlich als verkettete Liste implementiert, kann intern aber auch ein Array verwenden, in diesem Fall ist die Anzahl der Elemente begrenzt.

Queue in C++

STL-Header:	#include <queue>
Queue erzeugen:	queue<Datentyp> myQueue;
Wert speichern:	myQueue.push(Wert);
Wert löschen:	myQueue.pop();
Wert lesen:	variable = myQueue.front(); variable = myQueue.back();

Beispiel: Queue

```
#include <iostream>
#include <queue>

int main() {
    queue<double> myQueue;

    myQueue.push( 12.2 );
    myQueue.push( 13.4 );
    myQueue.push( 14.5 );

    cout << myQueue.front() << endl;

    myQueue.pop();

    double a = myQueue.front();

    cout << a + myQueue.back() << endl;

    return 0;
}
```

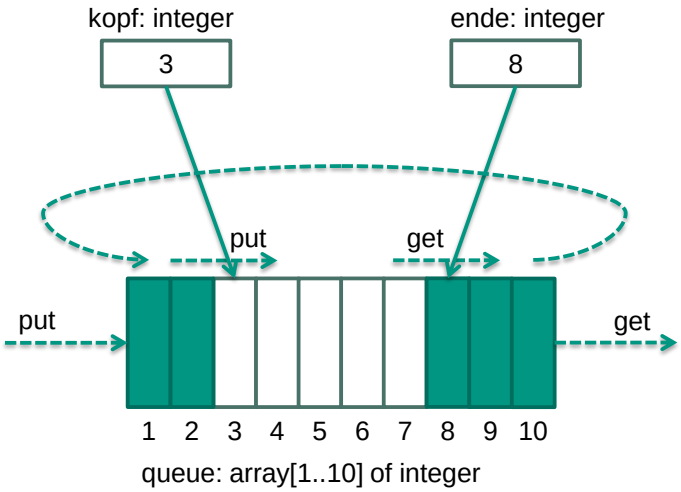


Ausgabe:

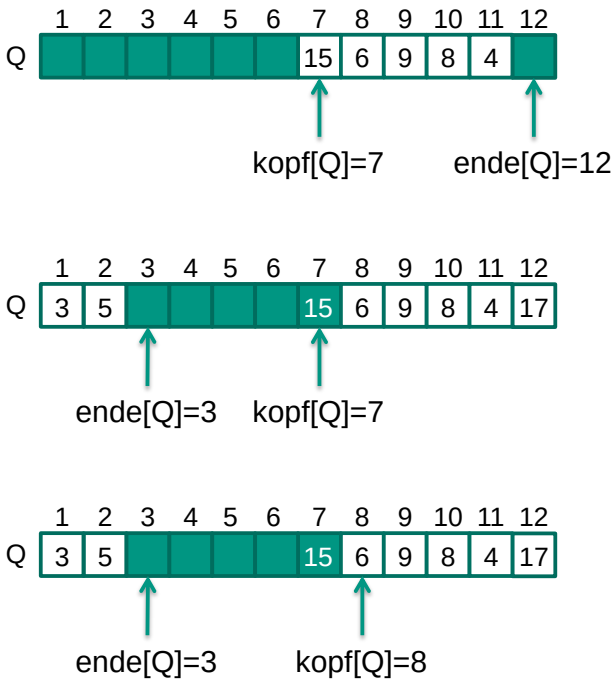
```
12.2
27.9
```

Queue

- Zwei Zeiger, Zeigerverwaltung



Queue implementiert mit Feld



Die Implementierung einer Warteschlange mithilfe eines Feldes Q[1..12].

Elemente der Warteschlange befinden sich nur an den schwach schattierten Positionen.

a) Die Warteschlange hat 5 Elemente an den Stellen Q[7..11].

b) Die Konfiguration der Warteschlange nach den Aufrufen enqueue(Q, 17), enqueue(Q, 3), enqueue (Q, 15).

c) Die Konfiguration der Warteschlange: nachdem der Aufruf dequeue(Q) den Schlüsselwert 15 zurückgegeben hat, der sich ehemals am Kopf der Warteschlange befand. Der neue Kopf hat den Schlüssel 6.

Sonderfall: Vorrangwarteschlange

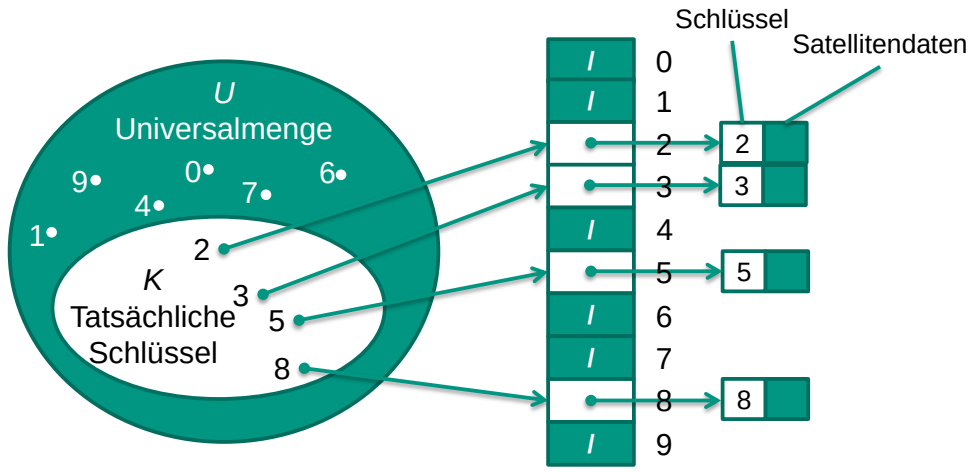
- Auch Prioritätswarteschlange (Priority Queue)
- Abweichung vom FIFO-Prinzip

- Operationen:
  - enqueue(priority) – insert gemäß Priorität in die Schlange einsortieren
  - dequeue() – sucht Objekt mit der höchsten Priorität

Vorrangwarteschlange:

Eine Spezialisierung der Warteschlange ist die Vorrangwarteschlange, welche auch Prioritätswarteschlange bzw. engl. Priority Queue genannt wird. Dabei wird vom FIFO-Prinzip abgewichen. Die Durchführung der enqueue Operation, die in diesem Fall auch insert Operation genannt wird, sortiert das Objekt gemäß einer gegebenen Priorität, die jedes Objekt mit sich führt, in die Vorrangwarteschlange ein. Die dequeue Operation liefert immer das Objekt mit der höchsten Priorität. Vorrangwarteschlangen werden meist mit Heaps implementiert.

Hashtabelle (Hashtable)



- Implementierung einer dynamischen Menge durch eine Adresstabelle mit direktem Zugriff T
- Hash: mathematische Funktion, die die Position des Datenelements in der Menge berechnet z.B. h(k) = k modulo m

Implementierung einer dynamischen Menge durch eine Adresstabelle mit direktem Zugriff T.

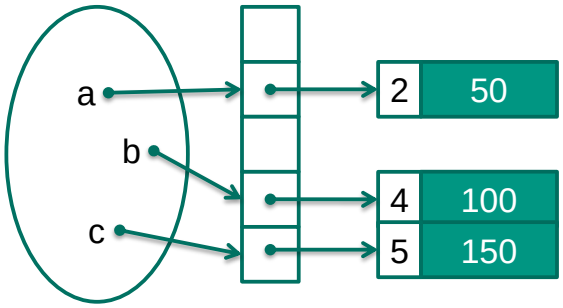
Jeder Schlüssel der Universalmenge U = {0, 1, …, 9) entspricht einem Index der Tabelle. Die Menge K = {2, 3, 5, 8} der tatsächlichen Schlüssel bestimmt die Plätze der Tabelle, die Zeiger auf Elemente enthalten. Die anderen Plätze (stark schattiert) enthalten NIL.

Hashtabelle in C++

STL-Header:	#include <unordered_map>
Hashtabelle erzeugen:	unordered_map<Schlüsseltyp, Datentyp> myMap;
Wert speichern:	myMap.insert(pair (Key, Wert));
Wert löschen:	myMap.erase(Iterator);
Wert lesen:	variable = myMap.find(Key);

Beispiel: Hashtabelle

```
#include <iostream>
#include <unordered_map>
using namespace std;

int main () {
    unordered_map<char, int> myMap;

    myMap[ 'a' ] = 50;
    myMap[ 'b' ] = 100;
    myMap[ 'c' ] = 150;

    myMap.erase( myMap.find( 'b' ) );

    cout << myMap.find( 'a' )->second << endl;
    cout << myMap.find( 'c' )->second << endl;

    return 0;
}
```



Ausgabe:

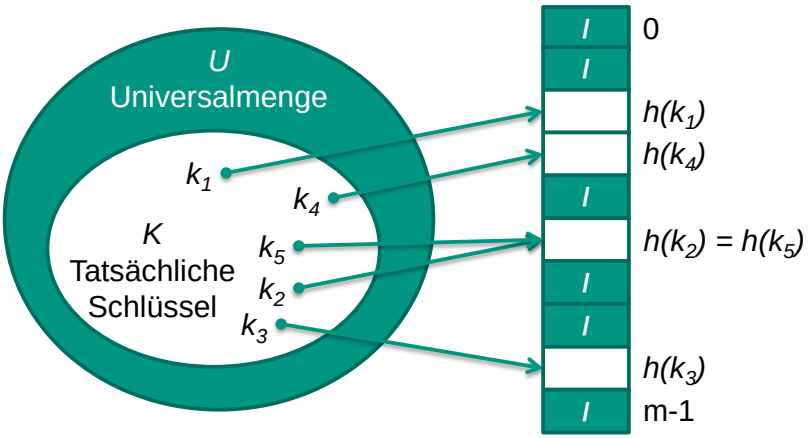
```
50
150
```

Hashfunktion

- **Divisionsmethode:**
  - Schlüssel *k* wird auf einen von *m* Schlüsseln abgebildet
  - Hashwert ergibt sich aus Rest bei Teilen von *k* durch *m*
  - ***=> h(k) = k mod m***

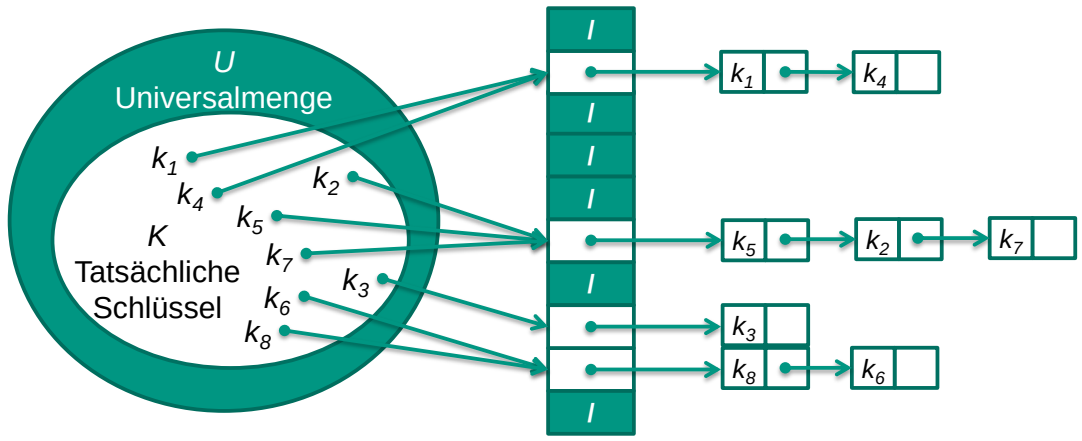
- Eine gute Wahl für *m* ist eine Primzahl, die nicht zu nahe bei einer Potenz von 2 liegt.

Hashfunktion - Kollision



- k_2 und k_5 werden auf den selben Platz abgebildet
- k_2 und k_5 kollidieren

Hashfunktion Kollisionsauflösung durch Verkettung



- Kollisionsauflösung durch Verkettung
- Elemente der Hashtabelle sind verkettete Listen

Jeder Platz T[j] der Hashtabelle enthält eine verkettete Liste der in der Menge K enthaltenen Schlüssel, deren Hashwert j ist. Beispielsweise gilt:

$h(k_1) = h(k_4)$ und $h(k_5) = h(k_2) = h(k_7)$.

Zwischenübung 02: Hashtabelle

- Tabelle soll $n = 2000$ Zeichenketten enthalten
- Kollisionen werden durch Verkettung gelöst
- Wir akzeptieren, dass im Mittel bei einer erfolglosen Suche drei Elemente überprüft werden müssen

- Wie groß muss m sein?
- Geben Sie die Hashfunktion an

Graphen und Bäume

Wiederholung

Vorlesung Digitaltechnik

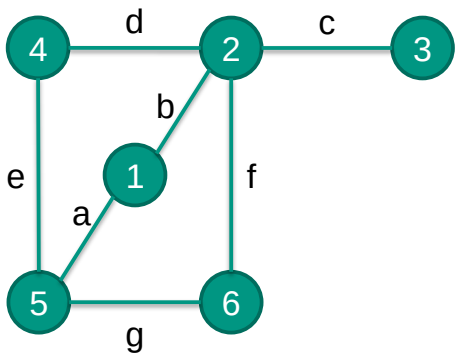
Graphen und Bäume

Graphentheorie: Abstrakter Graph

- Formale mathematische Beschreibung:
  - **Abstraktion** von der Bedeutung der Darstellungselemente:
    → Verknüpfung mit der **Begriffswelt** der **Mengen** und **Relationen**
  - Graphen können (unabhängig von der Darstellung) durch **zwei Mengen** und **eine Abbildung** beschrieben werden:
    - V = Menge der Knoten
    - E = Menge der Kanten
    - Φ (e) ordnet jeder Kante e $\in$ E zwei Knoten aus V zu
      -> diejenigen, die durch die Kante e verbunden sind
  - G (V, E, Φ) wird abstrakter Graph genannt

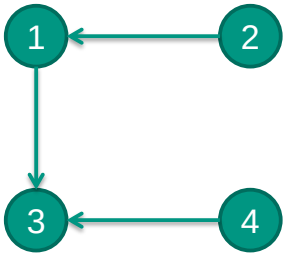
Graphentheorie: Abstrakter Graph

- **Beispiel:**
  - V = { 1, 2, ..., 6 }
  - E = { a, b, ..., g }
  - Φ: E $\rightarrow$ { v, w } v, w $\in$ V
    - Φ (a) = { 1, 5 }
    - Φ (b) = { 1, 2 }
    - Φ (c) = { 2, 3 }
    - Φ (d) = { 2, 4 }
    - Φ (e) = { 4, 5 }
    - Φ (f) = { 2, 6 }
    - Φ (g) = { 5, 6 }



Graphentheorie: Gerichteter Graph

- **Gerichteter Graph:**
  - Für manche Probleme benutzt man auch **gerichtete Graphen**
    → **Kanten** haben eine **festgelegte Richtung**
  - Bei gerichteten Graphen gilt:
    Φ bildet **Kanten** auf geordnete **Knoten-Tupel** aus **V x V** ab
  - Ein gerichteter Graph muss mindestens eine Kante zwischen zwei Knoten (g,h) besitzen, so dass keine Kante in umgekehrter Richtung (h,g) existiert
  - Gerichtete Graphen werden auch als **Digraphen** bezeichnet



Graphentheorie: Begriffe

- **Sprechweisen:**
  - Verbindet die Kante e die Knoten g und h, so sagt man:
    „e ist inzident zu g bzw. zu h“ und schreibt:
    - Φ (e) = (g, h) für gerichtete Graphen
    - Φ (e) = { g, h } = { h, g } für ungerichtete Graphen
  - Φ wird daher Inzidenzabbildung genannt
    → Graph lässt sich formal durch eine Inzidenzmatrix beschreiben
  - die Knoten g und h heißen adjazent zur Kante e
  - Adjazenzmatrix
    → weitere Möglichkeit zur formalen Beschreibung eines Graphen

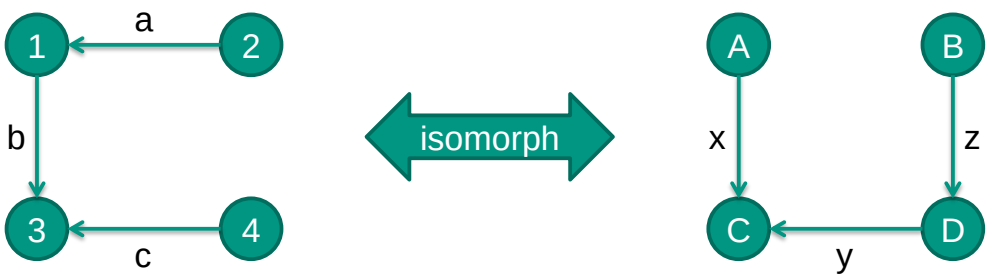
Graphentheorie: Begriffe

- **Globale Charakterisierung von Graphen:**
  - Der Graphentheorie fehlt es selbstverständlich nicht an Begriffen, um die unerschöpfliche Vielfalt der Graphen zu kategorisieren
  - Im folgenden wollen wir uns nur mit Graphen beschäftigen, deren Mengen V und E endlich sind:
    → endliche Graphen (insbesondere für technische Anwendungen)
  - Ist die Menge der Kanten E leer
    → so handelt es sich um einen entarteten Graphen
    → dieser besteht nur aus isolierten Knoten
  - Wenn zu je zwei verschiedenen Knoten höchstens eine Kante existiert
    → so handelt es sich um einen einfachen Graphen

Graphentheorie: Isomorphie

- Vergleichbarkeit zweier Graphen:
  → es werden Abbildungen zwischen Knoten und Kanten gesucht, so dass die Inzidenzbeziehungen erhalten bleiben
- Sind diese Zuordnungen bijektiv (eineindeutig)
  → so wird der Graph isomorph genannt
- Isomorphie von Graphen:
  → Strukturen isomorpher Graphen sind gleich
  → wichtige Eigenschaft in der Vergleichbarkeit und formalen Verifikation digitaltechnischer Schaltungen und Systeme

Isomorphie: Beispiel



- **Zuordnung der Knoten φ:** (1,2,3,4) ⇒ (D,B,C,A)
- **Zuordnung der Kanten ψ:** (a,b,c) ⇒ (z,y,x)
- **Inzidenzbeziehungen:**

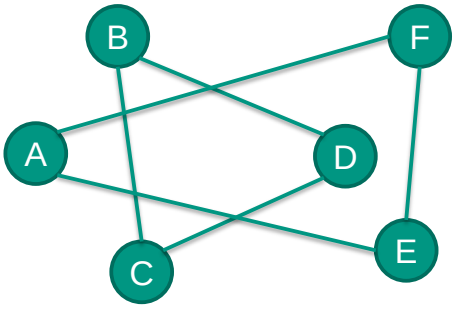
$\Phi(a) = (2,1) \quad \Leftrightarrow \quad \Phi(\psi(a)) = \Phi(z) = (B,D) = (\varphi(2), \varphi(1))$
$\Phi(b) = (1,3) \quad \Leftrightarrow \quad \Phi(\psi(b)) = \Phi(y) = (D,C) = (\varphi(1), \varphi(3))$
$\Phi(c) = (4,3) \quad \Leftrightarrow \quad \Phi(\psi(c)) = \Phi(x) = (A,C) = (\varphi(4), \varphi(3))$

Zusammenhängende Graphen

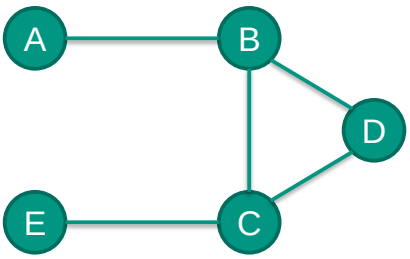
durch Folgen von Kanten und Knoten kann man von jedem beliebigen Knoten des Graphen zu jedem anderen Knoten des Graphen gelangen
→ der **Graph** ist **zusammenhängend**

Ist der Graph **nicht zusammenhängend**
→ so besteht er aus **mindestens zwei Teilgraphen**

Beispiel:



nicht zusammenhängend

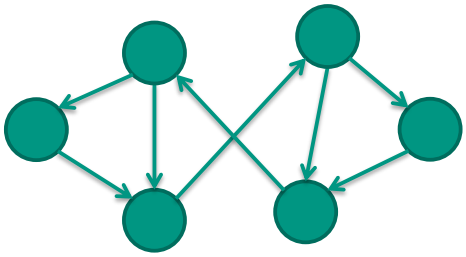


zusammenhängend

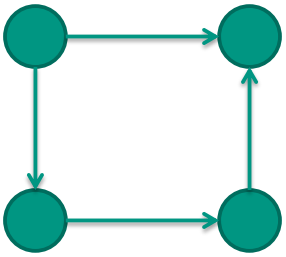
Streng zusammenhängende Graphen

Gerichtete Graphen bezeichnet man als **zusammenhängend**, wenn der zugehörige **ungerichtete Graph zusammenhängend** ist

Findet man zusätzlich von jedem beliebigen Knoten des Graphen zu jedem anderen Knoten eine **gerichtete Folge von Kanten** (Weg unter Einbezug der Richtungen!) → so ist der **Graph streng zusammenhängend**



streng zusammenhängend



nicht streng zusammenhängend

Lokale Eigenschaften von Graphen:

Schlinge oder Schleife

Verbindet eine Kante einen Knoten mit sich selbst, so wird diese als **Schleife** oder **Schlinge** bezeichnet

$\Phi(e) = (g,g)$

Mehrfachkanten

Existieren **mehrere Kanten** zwischen zwei Knoten g und h, so heißen diese **parallel** bzw. **Mehrfachkanten**

$\Phi(e) = \Phi(f) = \{g,h\}$ bzw. (g,h) $e \neq f$

Bei **gerichteten Graphen** nennt man Kanten **antiparallel**, falls sie zwei Knoten in entgegengesetzter Richtung verbinden

$\Phi(e) = (g,h)$, $\Phi(f) = (h,g)$

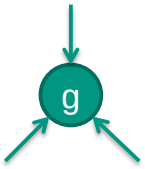
Grad eines Knotens

Betrachtet man einen **Knoten**, so wird die **Anzahl** der damit **inzidenten Kanten** als **Grad d(g)** des Knotens bezeichnet

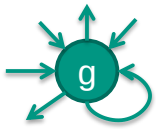
Bei **gerichteten Graphen** unterscheidet man zusätzlich
abgehende Kanten ⇒ **Ausgangsgrad $d^+(g)$**
von ankommenden Kanten ⇒ **Eingangsgrad $d^-(g)$**

Beispiele:

d(g) = 6



$d^-(g)= 3$



$d^+(g) = 3$
$d^-(g) = 4$

Nachbarschaft von Knoten

- **Unmittelbare Nachbarschaft** von Knoten
  - Sind **zwei Knoten** durch **eine Kante** verbunden
    → so sind die Knoten **unmittelbar benachbart**
  - **Menge** der **unmittelbar benachbarten Knoten**
    → wird mit **V'(g)** bezeichnet
  - Bei **gerichteten Graphen** gilt:
    → die Menge der benachbarten Knoten wird **entsprechend** der Richtungen der Kanten eingeteilt in:
      → **unmittelbare Vorgänger** $V'_1(g)$ und
      → **unmittelbare Nachfolger** $V'_2(g)$

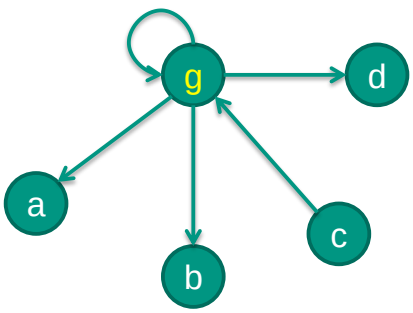
Nachbarschaft von Knoten

- **Mittelbare Nachbarschaft** von Knoten
  - Schwächt man die Forderung so ab, dass nur eine **Folge** von **Kanten zwischen zwei Knoten** existieren muss
    → so sind die Knoten **mittelbar benachbart**
  - Nützlich ist die mittelbare Nachbarschaft besonders bei gerichteten Graphen bzgl. der Unterscheidung in
    → **mittelbare Vorgänger** und
    → **mittelbare Nachfolger**

Zwischenübung 03: Nachbarschaft

Bestimmen Sie für den gegebenen Graphen

- Vorgänger: V‘1(g)
- V‘(g)
- Nachfolger: V‘2(g)
- V‘(g)
- d-(g)
- d+(g)
- d(g)
- |V‘(g)|



Kantenfolgen in Graphen

Bereits bei der mittelbaren Nachbarschaft haben wir uns dafür interessiert, ob Knoten über **Kantenfolgen** verbunden sind

Definition: **Kantenprogression** der **Länge n**

→ **endliche Folge** von **n nicht** notwendigerweise **verschiedenen Kanten (e^i),** die n + 1 nicht notwendigerweise verschiedene Knoten (g^i) verbinden

$$\Phi(e^i) = (g^i, g^{i+1}) \quad \text{für } i = 1, 2, ..., n$$

Sind in einer **Kantenprogression alle Knoten (g^i)** voneinander **verschieden** und damit auch alle Kanten, so heißt sie **einfach**

Je nachdem, ob der Anfangsknoten gleich dem Endknoten ist ($g^1 = g^{n+1}$), oder ob es sich um einen gerichteten Graphen handelt, verwendet man unterschiedliche Bezeichnungen

Spezielle Kantenfolgen in Graphen

Ungerichteter Graph		**Gerichteter Graph**	
$g^1 \neq g^{n+1}$	$g^1 = g^{n+1}$	$g^1 \neq g^{n+1}$	$g^1 = g^{n+1}$
offene Kanten-progression	geschlossene Kanten-progression	offene Kanten-progression	geschlossene Kanten-progression
Sind alle Kanten einer Progression voneinander verschieden			
Ketten-progression	geschlossene Kantenzug-progression	Weg-progression	Zyklus-progression
Berücksichtigt man die Kanten einer Progression ohne Ordnung			
Kette	geschlossener Kantenzug	Weg	Zyklus

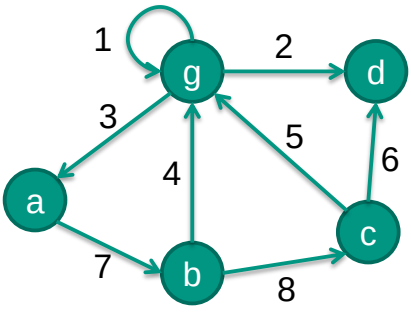
Zyklus

- Begriff des Zyklus/Zyklen in Graphen:
  - Für viele Algorithmen, die auf Graphen arbeiten ist es wichtig zu wissen, ob es möglich ist, mit unterschiedlichen Kanten „im Kreis zu laufen“
  - Man bezeichnet einen kompletten **Graphen** als **zyklisch**, wenn
    → wenigstens eine **geschlossene Kantenzugprogression**
    (= **Zyklus**) existiert in diesem Graphen
    → eine **Schleife** ist ebenfalls ein **Zyklus**
  - Wenn ein **Graph nicht zyklisch** ist
    → nennt man ihn **zyklenfrei** oder **azyklisch**

Zwischenübung 04: Spezielle Kantenfolgen

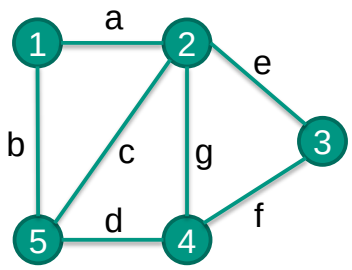
- **Beispiele am gerichteten Graphen:**
  Finden Sie für den gegebenen Graphen folgende Kantenfolgen:

Merkmal
Offene Kantenprogression
Geschl. Kantenprogression
Wegprogresssion
Zyklusprogression
Weg
Zyklus



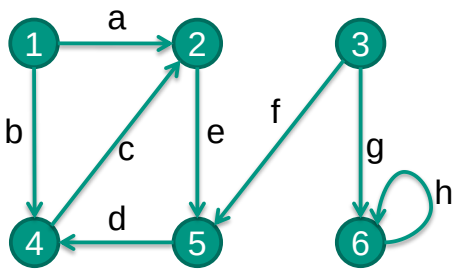
Graphen: Darstellung Inzidenzmatrix

Inzidenzdarstellung für einen ungerichteten Graphen:



	a	b	c	d	e	f	g
1	1	1					
2	1		1		1		1
3					1	1	
4				1		1	1
5		1	1	1			

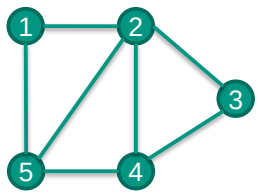
Inzidenzdarstellung für einen gerichteten Graphen:

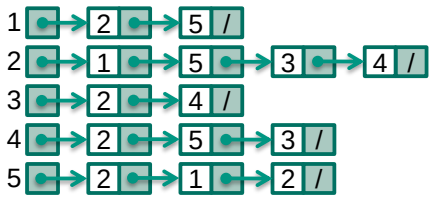


	a	b	c	d	e	f	g	h
1	-1	-1						
2	1		1		-1			
3						-1	-1	
4		1	-1	1				
5				-1	1	1		
6							1	1

Graphen: Darstellung Adjazenzmatrix oder Liste

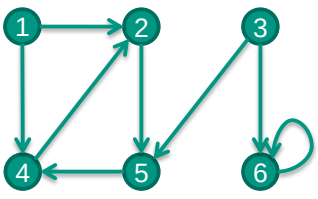
Adjazenzdarstellung für einen ungerichteten Graphen:

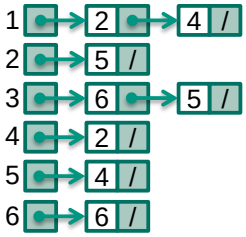




	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Adjazenzdarstellung für einen gerichteten Graphen:





	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Spezieller Graph: Baum

- **Definition:** **Baum**
  → ein **zusammenhängender**, **zyklenfreier** Graph

- **Also:** bei einem **ungerichteten Baum**
  → es darf **kein geschlossener Kantenzug** existieren
- bei einem **gerichteten Baum** gilt:
  → es darf **kein Zyklus** existieren
  → der zugehörige ungerichtete Graph muss **zusammenhängend** sein

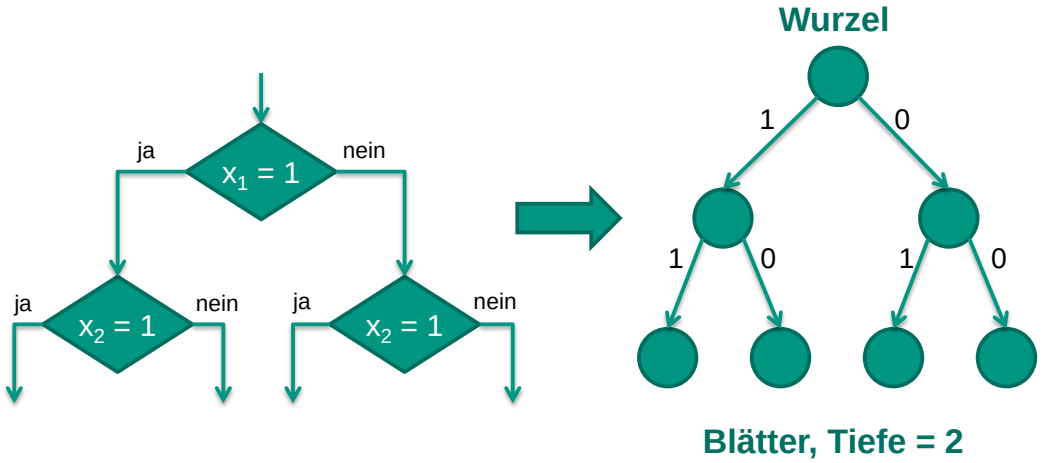
Bäume: Wurzel, Blätter, Tiefe

- **Definition Wurzel, Blätter:** In einem **gerichteten Baum**
  → gibt es genau einen **Knoten**, der **keine Vorgänger** hat
  $d^- = \emptyset$
  → dieser Knoten wird **Wurzel** genannt
  Die Knoten ohne Nachfolger ($d^+ = \emptyset$) heißen **Blätter**
- Bei einem **ungerichteten Baum**
  → kann die **Wurzel** frei gewählt werden
  → daraus ergeben sich die **Blätter** als übrige **Knoten** mit **Knotengrad eins**

- **Definition Tiefe:** Der **Abstand** von der **Wurzel** zu den **Blättern**
  → wird als **Tiefe** des **Baumes** bezeichnet
  - Ist der Abstand von der **Wurzel** zu den **Blättern** für alle Blätter **identisch**
    → so handelt es sich um einen **symmetrischen** Binärbaum
  - Unterscheiden sich die Abstände um maximal eins
    → so nennt man den **Baum ausgeglichen**

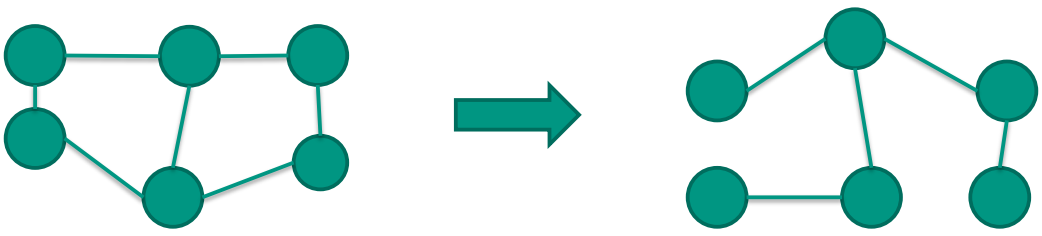
Binärbaum

- **Definition:** **Binärbaum**
  - Hat jeder Knoten eines Baumes, außer den Blättern, **genau zwei** Nachfolger: **$d^+(g) = 2$**, so heißt ein solcher Baum **Binärbaum**



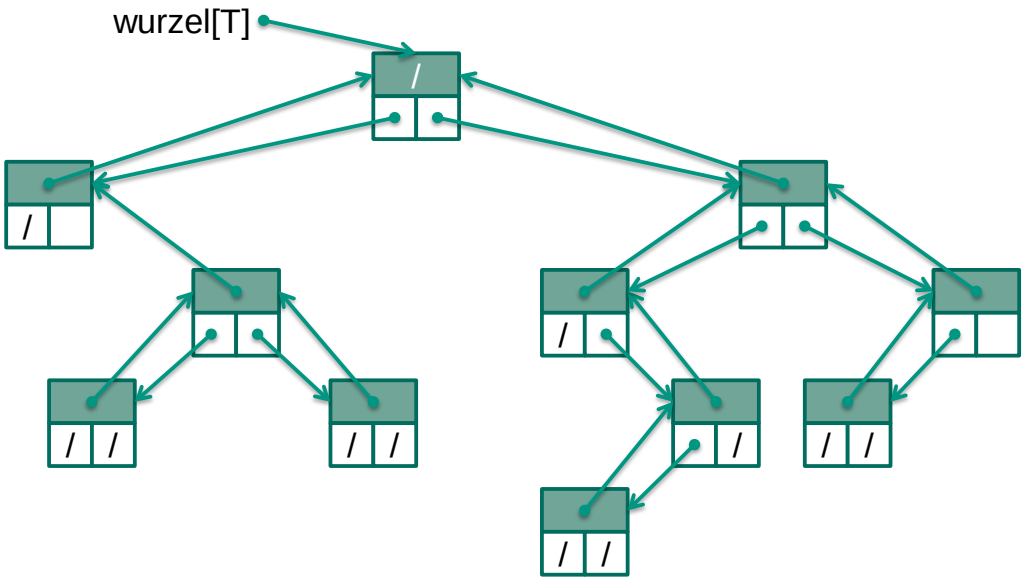
Aufspannender Baum

- **Definition:** **Aufspannender Baum**
  - **Es gilt:** bei jedem **zusammenhängenden Graphen** kann man **alle Zyklen** durch **gezieltes Entfernen** von **Kanten** auflösen (→ *Kruskal Algorithmus*: Minimal Aufspannender Baum, MAB) **ohne** dass der Graph in **zwei Teilgraphen zerfällt**
  - Auf diese Weise erhält man zu jedem beliebigen Graphen einen zugehörigen **aufspannenden Baum**



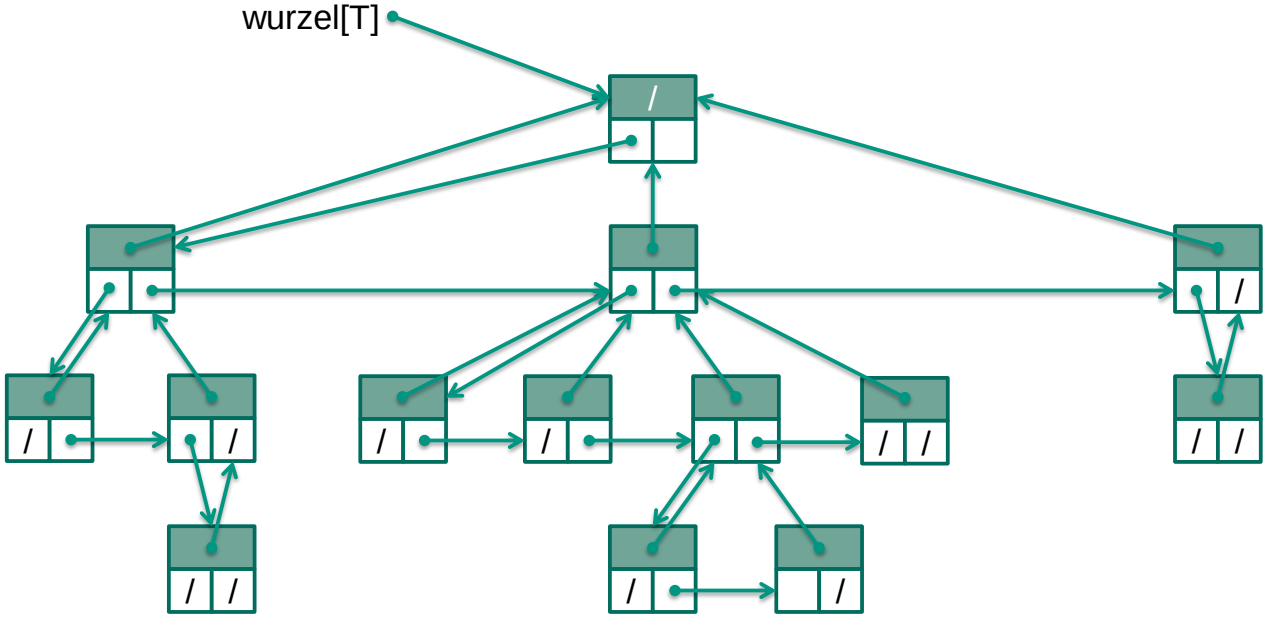
Darstellung eines Binärbaumes über eine Liste

- Dabei besteht jeder Knoten aus dem Wert und drei Zeigern: Vorgänger, linker Nachfolger, rechter Nachfolger



Darstellung eines allgemeinen Baumes über eine Liste

- Nun besteht jeder Knoten aus dem Wert und drei anderen Zeigern: Vorgänger, linker Nachfolger, rechter „Bruder“



Heap

Der Heap (Halde, Haufen) vereint die Datenstruktur eines Baums mit den Operationen einer Vorrangwarteschlange. Häufig hat der Heap neben den minimal nötigen Operationen wie insert, remove und extractMin auch noch weitere Operationen wie merge oder changeKey. Je nach Reihenfolge der Priorität in der Vorrangwarteschlange wird ein Min-Heap oder ein Max-Heap verwendet. Heaps werden meistens über Bäume aufgebaut.

Eigenschaft Max-Heap:
Für jeden Knoten (außer Wurzel) gilt
Der Wert eines Knotens i ist
kleiner, höchstens gleich dem Wert
des Vaterknotens V(i)