Karlsruher Institut für Technologie

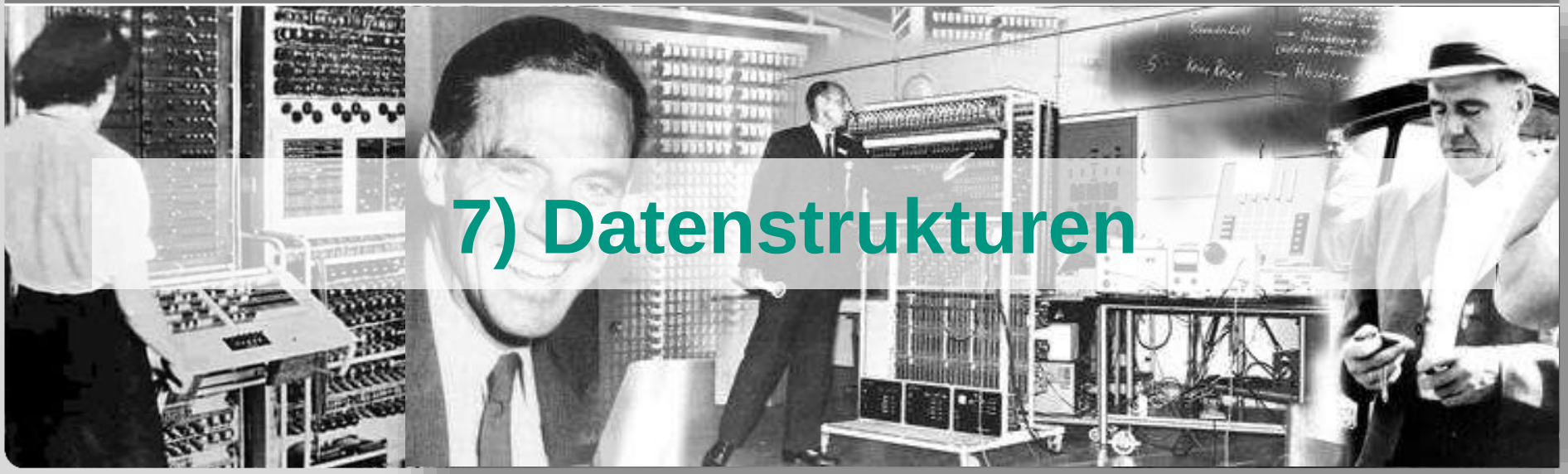
Institut für Technik der Informationsverarbeitung

Institutsleitung
Prof. Dr.-Ing. J. Becker
Prof. Dr.-Ing. E. Sax
Prof. Dr. rer. nat. W. Stork

Vorlesung Informationstechnik (IT)

Sommersemester 2018

Institut für Technik der Informationsverarbeitung (ITIV)



7) Datenstrukturen

IT Inhalt

6. Objektorientierung

- Grundidee und Motivation
- Klassen
- Objekte
- Datenkapselung

7. Datenstrukturen

- Array
- Liste
- Stack
- Queue
- Hash-Tabelle
- Graph
- Baum
- Haufen

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
  - Es handelt sich um eine Struktur, weil die Daten in einer bestimmten Art und Weise angeordnet und verknüpft werden, um den Zugriff auf sie und ihre Verwaltung effizient zu ermöglichen.
- 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.

Struktur → Ordnung

Abstraktion der Datenspeicherung

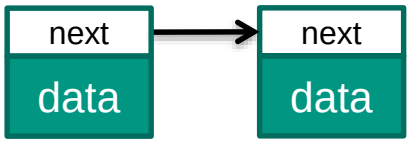
Abstraktion

00100010010
01010001000
10

Daten im Speicher abgelegt als Nullen und Einsen in Speicherzellen

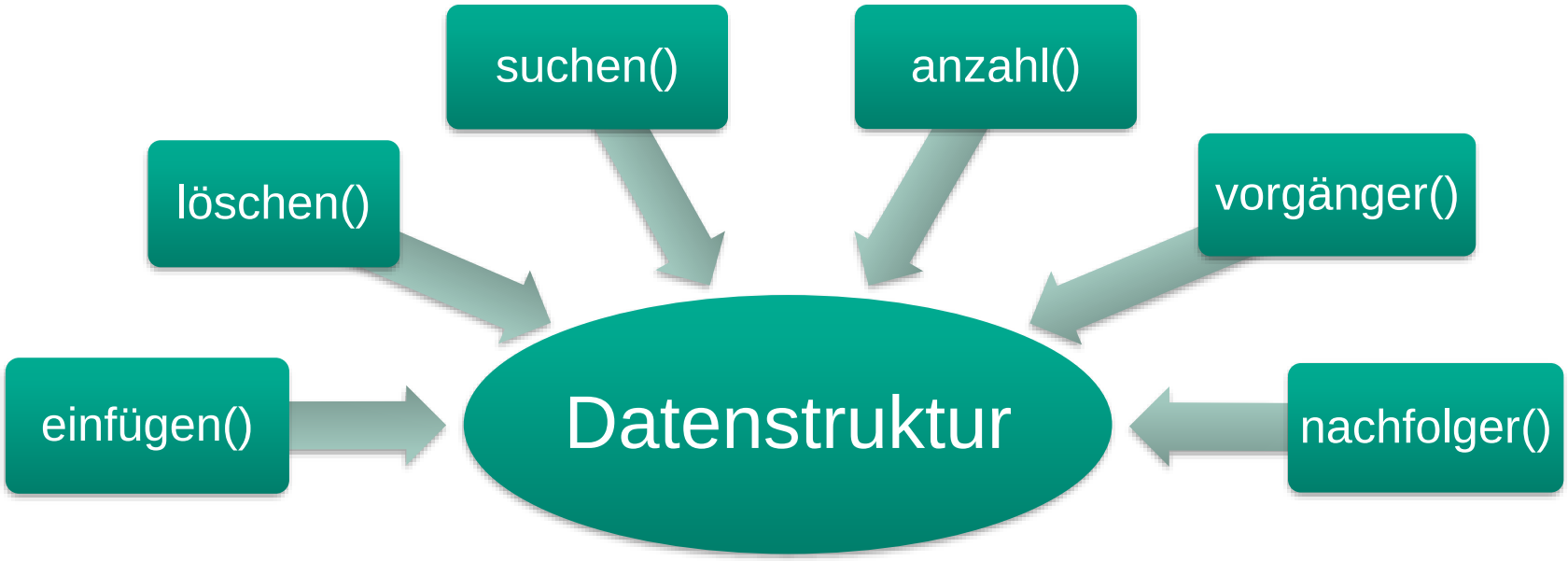
double a;
int b;
float arr[10];

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



Datenstrukturen organisieren Daten unabhängig vom Datentyp in definierter Art und Weise

Operationen auf Datenstrukturen



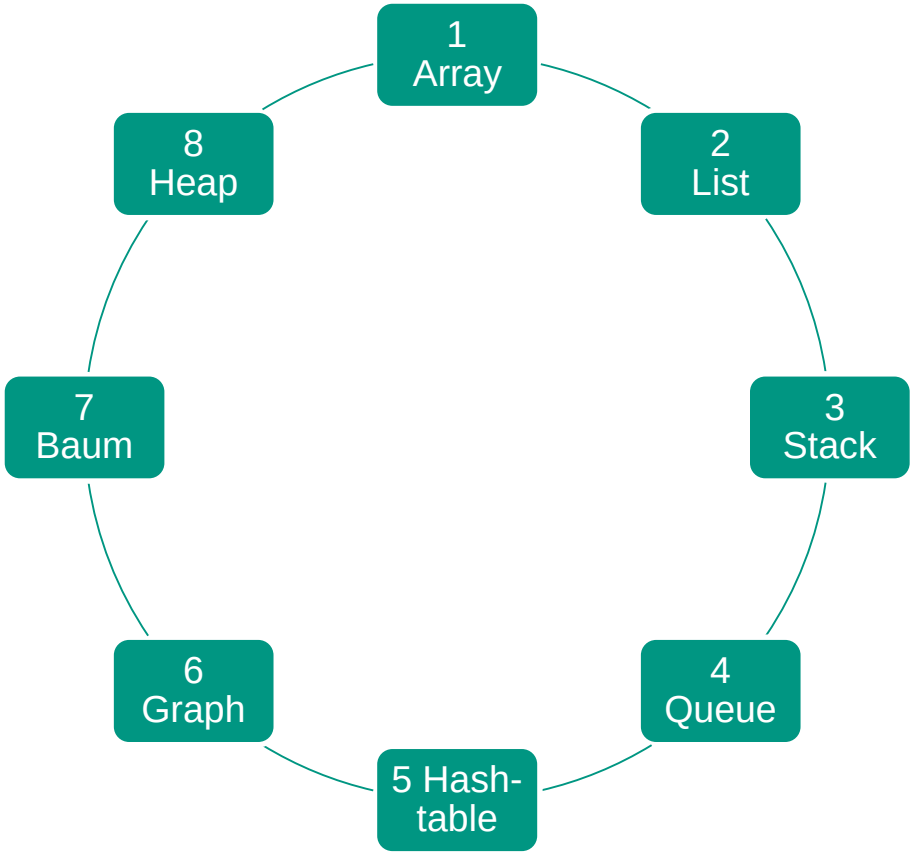
- Standard-Operationen auf Datenstrukturen
- Verschiedene Datenstrukturen sind unterschiedlich effizient für diese Operationen
- Bieten die Möglichkeit Datenstrukturen zu klassifizieren

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
  - Es ist ein Mechanismus, der es erlaubt:
    - an jeder Position auf ein entsprechendes Objekt lesend und / oder schreibend zuzugreifen
    - von der Position eines Objekts zur Position des nächsten Objekts in der Liste zu wechseln.

Bekannte Datenstrukturen

- Übersicht über bekannte Datenstrukturen:
  (für imperative Programmierung)



http://www.cplusplus.com/reference/

C Library

The elements of the C language library are also included as a subset of the C++ Standard library. These cover many aspects, from general utility functions and macros to input/output functions and dynamic memory management functions:

<cassert> (assert.h)	C Diagnostics Library (header)
<cctype> (ctype.h)	Character handling functions (header)
<cerrno> (errno.h)	C Errors (header)
<cfenv> (fenv.h)	Floating-point environment (header)
<cfloat> (float.h)	Characteristics of floating-point types (header)
<cinttypes> (inttypes.h)	C integer types (header)
<ciso646> (iso646.h)	ISO 646 Alternative operator spellings (header)
<climits> (limits.h)	Sizes of integral types (header)
<clocale> (locale.h)	C localization library (header)
<cmath> (math.h)	C numerics library (header)
<csetjmp> (setjmp.h)	Non local jumps (header)
<csignal> (signal.h)	C library to handle signals (header)
<cstdarg> (stdarg.h)	Variable arguments handling (header)
<cstdbool> (stdbool.h)	Boolean type (header)
<cstddef> (stddef.h)	C Standard definitions (header)
<cstdint> (stdint.h)	Integer types (header)
<cstdio> (stdio.h)	C library to perform Input/Output operations (header)
<cstdlib> (stdlib.h)	C Standard General Utilities Library (header)
<cstring> (string.h)	C Strings (header)
<ctgmath> (tgmath.h)	Type-generic math (header)
<ctime> (time.h)	C Time Library (header)
<cuchar> (uchar.h)	Unicode characters (header)
<cwchar> (wchar.h)	Wide characters (header)
<cwctype> (wctype.h)	Wide character type (header)

Containers

<array>	Array header (header)
<bitset>	Bitset header (header)
<deque>	Deque header (header)
<forward_list>	Forward list (header)
<list>	List header (header)
<map>	Map header (header)
<queue>	Queue header (header)
<set>	Set header (header)
<stack>	Stack header (header)
<unordered_map>	Unordered map header (header)
<unordered_set>	Unordered set header (header)
<vector>	Vector header (header)

Elements of the iostream library (char instantitation)

Classes:

ios_base	Base class for streams (class)
ios	Base class for streams (type-dependent components) (class)
istream	Input stream (class)
ostream	Output Stream (class)
iostream	Input/output stream (class)
ifstream	Input file stream class (class)
ofstream	Output file stream (class)
fstream	Input/output file stream class (class)
istringstream	Input string stream (class)
ostringstream	Output string stream (class)
stringstream	Input/output string stream (class)
streambuf	Base buffer class for streams (class)
filebuf	File stream buffer (class)
stringbuf	String stream buffer (class)

Bekannte Datenstrukturen (Übersicht 1)

- **1) Array**
  - Es werden hierbei mehrere Variablen vom selben Basisdatentyp gespeichert.
  - Ein Zugriff auf die einzelnen Elemente wird über einen Index möglich.
    - Technisch gesehen entspricht dieser dem Wert, der zu der Startadresse des Arrays im Speicher addiert wird, um die Adresse des Objektes zu erhalten.
  - Operationen die auf jindiziertes des Element
  - Im eindimensionalen Fall wird das Array häufig als Vektor und im zweidimensionalen Fall als Tabelle oder Matrix bezeichnet.

- Remark: *Arrays sind aber keinesfalls nur auf zwei Dimensionen beschränkt, sondern werden beliebig mehrdimensional verwendet.*

Bekannte Datenstrukturen (Übersicht 2)

- **2) Verkettete Liste**
  - Die verkettete Liste ist eine Datenstruktur zur dynamischen Speicherung von beliebig vielen Objekten.
  - Dabei beinhaltet jedes Listenelement einer verketteten Liste als Besonderheit einen Verweis auf das nächste Element, wodurch die Gesamtheit der Objekte zu einer Verkettung von Objekten wird.
- **3) Stapelspeicher**
  - In einem Stapelspeicher (engl. *stack* oder ‚Kellerspeicher') kann eine beliebige Anzahl von Objekten gespeichert werden, jedoch können die gespeicherten Objekte nur in umgekehrter Reihenfolge wieder gelesen werden.
    - Dies entspricht dem LIFO-Prinzip.
  - Für die Definition und damit die Spezifikation des Stapelspeichers ist es unerheblich, welche Objekte in ihm gespeichert werden.
  - Zu einem Stapelspeicher gehören zumindest die Operationen
    - *push*, um ein Objekt im Stapelspeicher abzulegen
    - *pop*, um das zuletzt gespeicherte Objekt wieder zu lesen und vom Stapel zu entfernen.
  - Ein Stapelspeicher wird gewöhnlich als Liste implementiert, kann aber auch ein Vektor sein.

Bekannte Datenstrukturen (Übersicht 3)

- **4) Warteschlange**
  - In einer Warteschlange (engl. *queue*) kann eine beliebige Anzahl von Objekten gespeichert werden.
  - Die gespeicherten Objekte werden in der gleichen Reihenfolge wieder gelesen werden, wie sie gespeichert wurden. → Dies entspricht dem FIFO-Prinzip.
  - Für die Definition und damit die Spezifikation der Queue ist es unerheblich, welche Objekte in ihm gespeichert werden.
  - Zu einer Queue gehören zumindest die Operationen
    - *enqueue*, um ein Objekt in der Warteschlange zu speichern und
    - *dequeue*, um das zuerst gespeicherte Objekt wieder zu lesen und aus der Warteschlange zu entfernen.
  - Eine Warteschlange wird gewöhnlich als verkettete Liste implementiert, kann intern aber auch ein Vektor sein; in diesem Fall ist die Anzahl der Elemente begrenzt.
- **5) Hashtabelle**
  - Die Hashtabelle bzw. Streuwerttabelle ist eine spezielle Index-Struktur, bei der die Speicherposition direkt berechnet werden kann.
  - Hashtabellen stehen dabei in Konkurrenz zu Baumstrukturen, die im Gegensatz zu Hashtabellen alle Indexwerte in einer Ordnung wiedergeben können, aber einen größeren Verwaltungsaufwand benötigen, um den Index bereitzustellen.
  - Beim Einsatz einer Hashtabelle zur Suche in Datenmengen spricht man vom *Hashverfahren*.
- **6) Graph**
  - Ein Graph ermöglicht es als Datenstruktur die Unidirektionalität der Verknüpfung zu überwinden.
  - Die Operationen sind auch hier das *Einfügen*, *Löschen* und *Finden* eines Objekts.
  - Die bekannteste Repräsentation von Graphen im Computer sind die Adjazenzmatrix und die Inzidenzmatrix.

Bekannte Datenstrukturen (Übersicht 4)

- **7) Baum**
  - Bäume sind spezielle Formen von Graphen in der Graphentheorie.
    - Dabei können ausgehend von der Wurzel mehrere gleichartige Objekte miteinander verkettet werden, so dass die lineare Struktur der Liste aufgebrochen wird und eine Verzweigung stattfindet.
  - Da Bäume zu den meist verwendeten Datenstrukturen in der Informatik gehören, gibt es viele Spezialisierungen.
    - So beträgt bei Binärbäumen die Anzahl der Kinder höchstens 2
  - Bei geordneten Bäumen, insbesondere Suchbäumen, sind die Elemente in der Baumstruktur geordnet abgelegt, sodass man schnell Elemente im Baum finden kann.
  - Bäume sind in ihrem Aufbau zwar mehrdimensional jedoch in der Verkettung der Objekte unidirektional.
    - Die Verkettung der gespeicherten Objekte beginnt bei der Wurzel des Baums und von dort in Richtung der Knoten des Baums.
- **8) Heap**
  - Der Heap (auch Halde oder 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*.


WESTFALEN
STADION

IT Inhalt

6. Objektorientierung

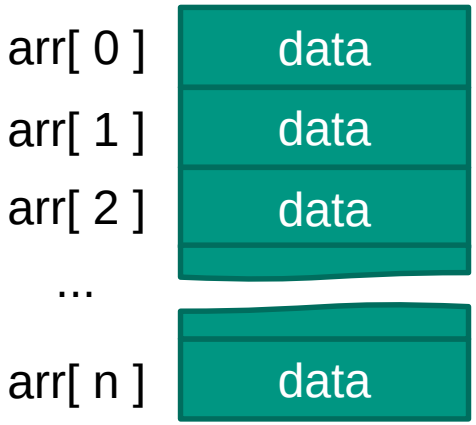
- Grundidee und Motivation
- Klassen
- Objekte
- Datenkapselung

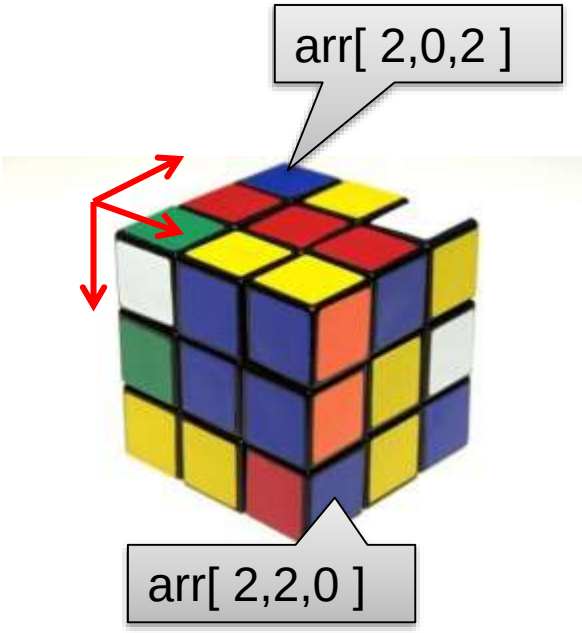
7. Datenstrukturen

- Array
- Liste
- Stack
- Queue
- Hash-Tabelle
- Graph
- Baum
- Haufen

1) Array („Feld“)

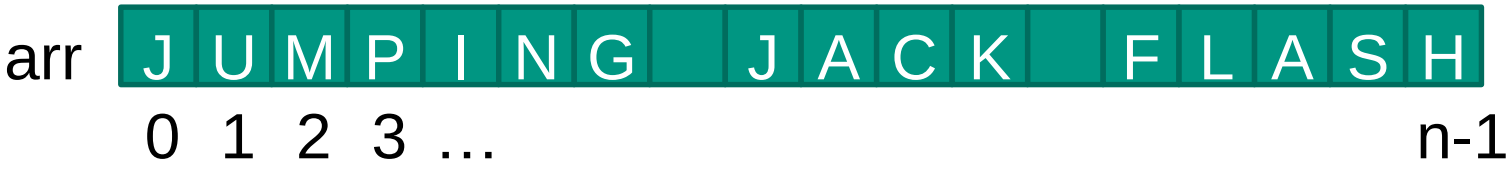
- Einfachste verwendete Datenstruktur
- Speichert mehrere Variablen des selben Datentyps
  - eindimensional → Vektor
  - zweidimensional → Tabelle oder Matrix
  - n-dimensional
- Schneller Zugriff auf Elemente über Index
- Nachteil: keine dynamische Größenanpassung
- Operationen:
  - Indiziertes Lesen
  - Indiziertes Speichern





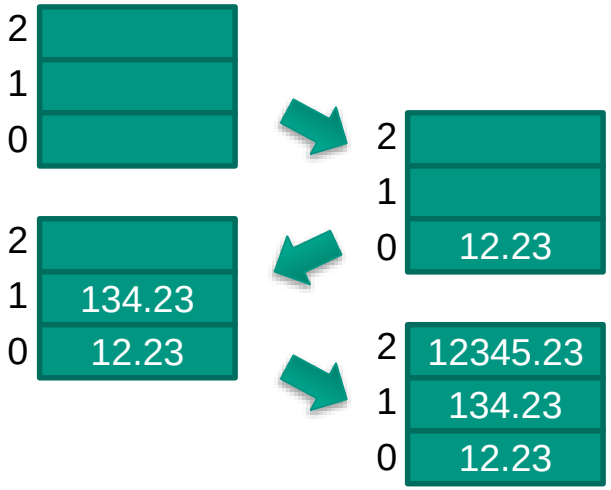
Array in C++ Code

- 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
```

- Array

IT Inhalt

6. Objektorientierung

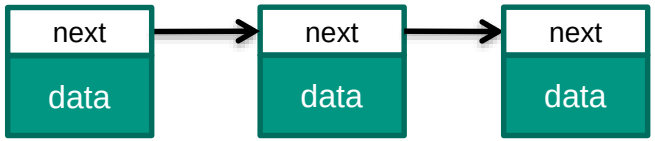
- Grundidee und Motivation
- Klassen
- Objekte
- Datenkapselung

7. Datenstrukturen

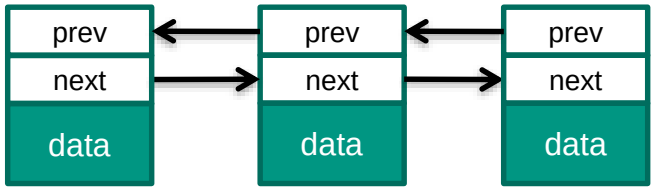
- Array
- Liste
- Stack
- Queue
- Hash-Tabelle
- Graph
- Baum
- Haufen

2) List (Liste)

- Dynamische Speicherung beliebig vieler Elemente
- 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

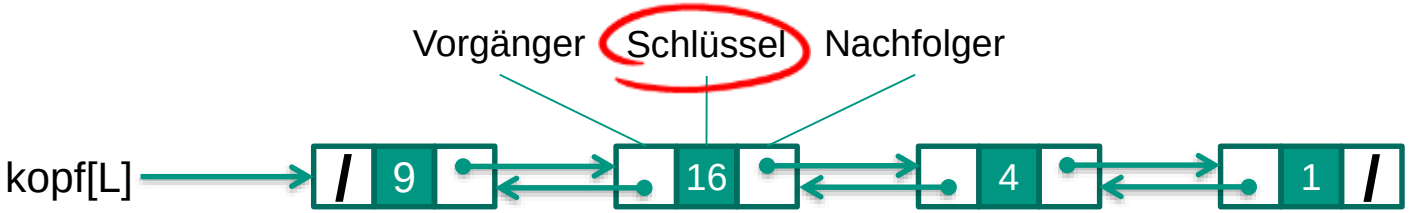


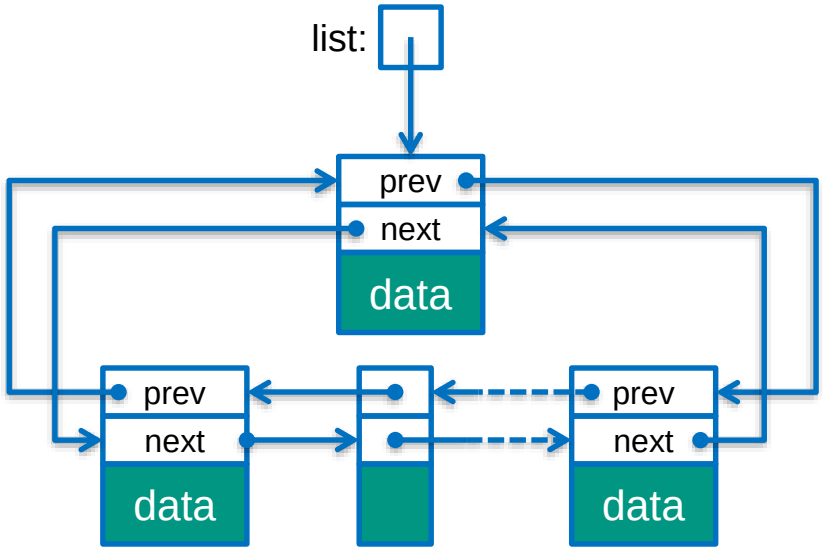
doppelt verkettete Liste

- Unterschiede:
  - Speicherbedarf: 1 oder 2 zusätzliche Informationen pro Listen-Element
  - Suchperformanz: von einer bzw. beiden Seiten
    - Nachteil: Langsamerer Zugriff als bei Array, kein indizierter Zugriff

List: Listenkopf, NIL

- „NIL“ (not in list) oder „/“
  → Nullwert, zeigt das Fehlen eines gültigen Wertes und hier das Listenende an.





- Wesentliche semantische Ergänzung.
- Zeigt (Anfang und) Ende einer (doppelt verketteten) Liste an

- Zyklische Liste
  - Ohne Null-Zeiger

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);

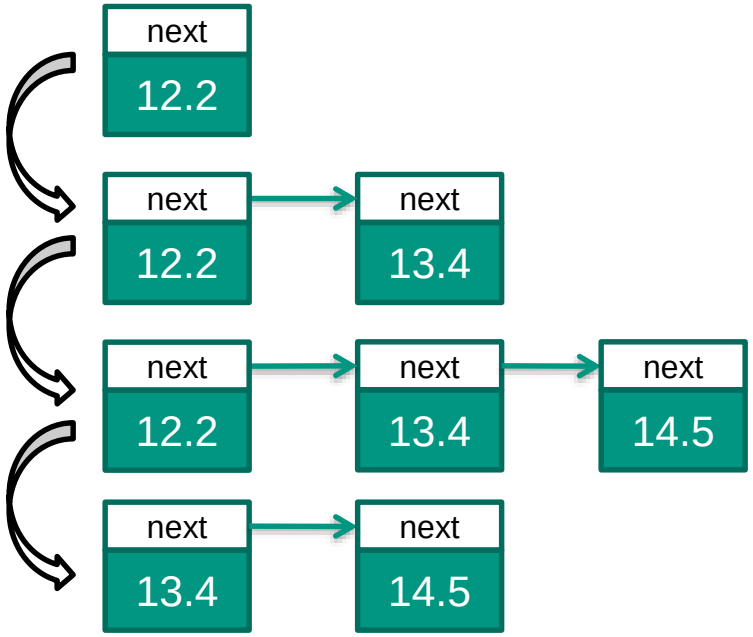
Wert löschen:	myList.erase(iterator);
Wert lesen:	variable = myList.back(); variable = myList.front();

Am Anfang hinzufügen

Am Ende hinzufügen

Beispiel: List in C++

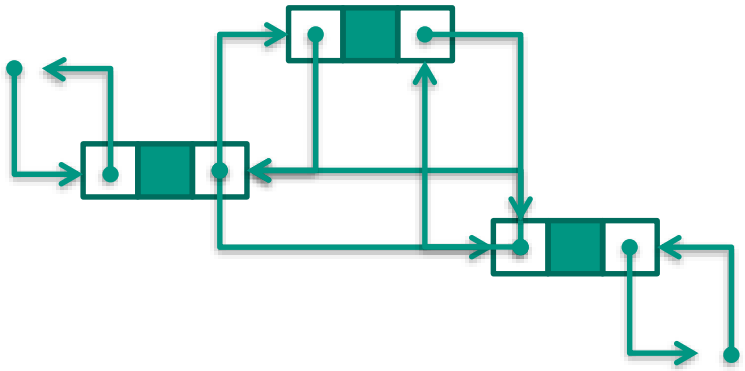
```
#include <iostream>
#include <list>

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:

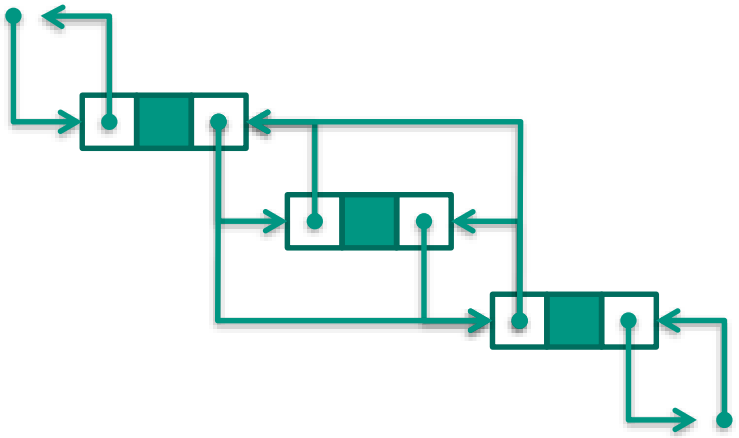
```
12.2
27.9
```

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

- List

IT Inhalt

6. Objektorientierung

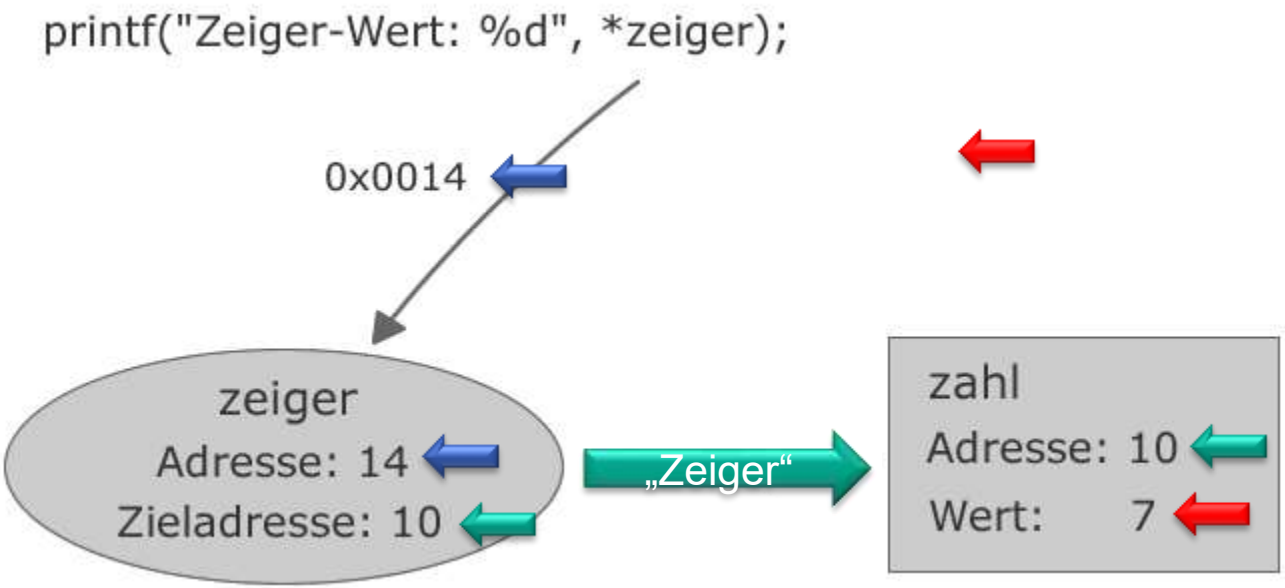
- Grundidee und Motivation
- Klassen
- Objekte
- Datenkapselung

7. Datenstrukturen

- Array
- Liste
- Stack
- Queue
- Hash-Tabelle
- Graph
- Baum
- Haufen

Zsf. Zeiger (*Pointer*)

- Ein Zeiger repräsentiert eine Adresse → eine Variable einen Wert.
- Will man auf den Wert in der Adresse zugreifen, auf die ein Zeiger zeigt, muss der Stern * vor den Namen gesetzt werden.
- Will man die Adresse einer Variablen ermitteln, muss das Kaufmanns-Und & vor den Namen gesetzt werden.



Zsf. Zeiger

- Zu dem vorangegangenem Beispiel das konkrete Bild im Speicher:

1) `int zahl = 7;`

Adresse	Wert
0x0010	0x00000007

2) `int *zeiger;`

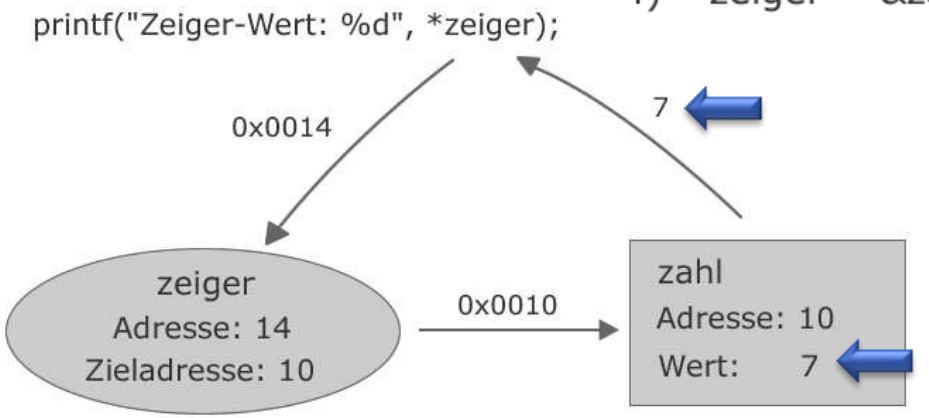
Adresse	Wert
0x0010	0x00000007
0x0014	0xFF1CA68D

3) `zeiger = NULL;`

Adresse	Wert
0x0010	0x00000007
0x0014	0x00000000

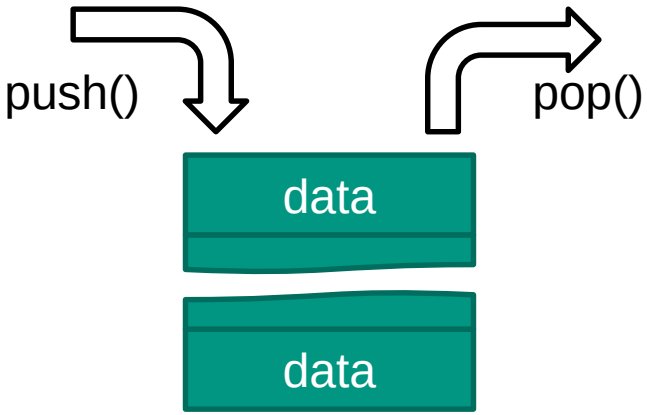
4) `zeiger = &zahl;`

Adresse	Wert
0x0010	0x00000007
0x0014	0x00000010



Stack (Stapelspeicher)

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

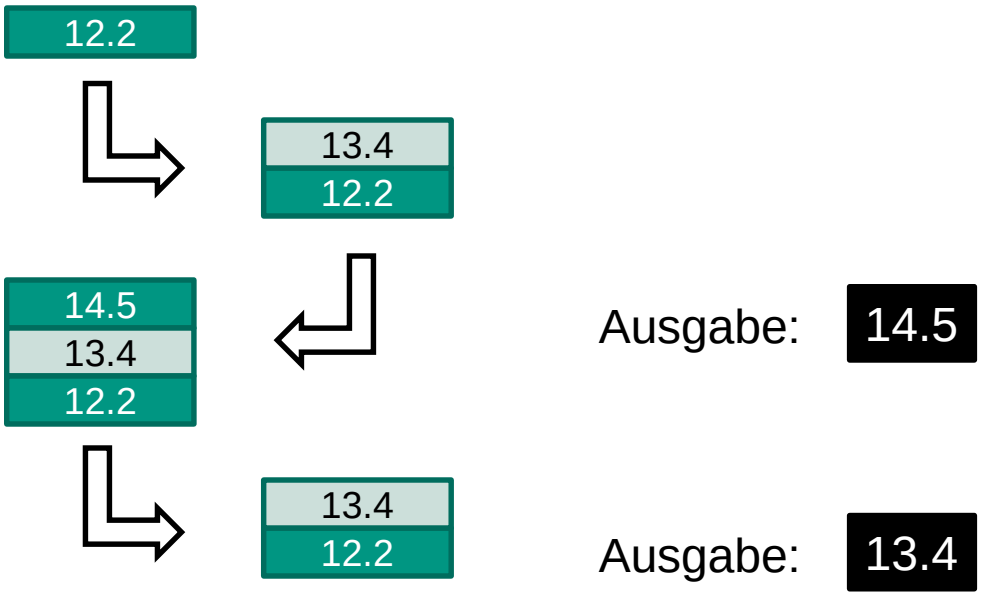


Stack in C++ Code

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

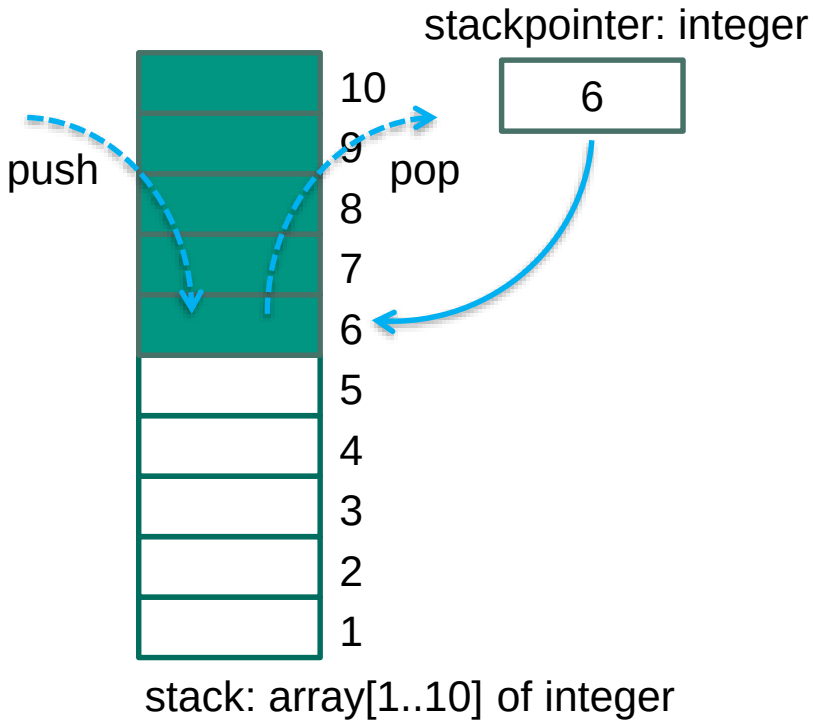
Übung: Stack

```
#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;
}
```



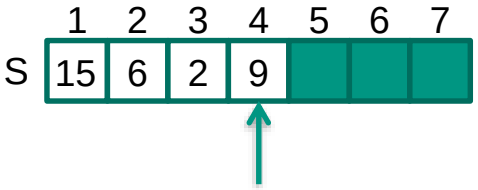
Stack (Stapelspeicher als Array)

- Stackpointer (Stapelzeiger) ist ein Zeiger zur Stapelverwaltung:
  - Der Stackpointer ist ein Register und zeigt auf die nächste freie Zelle.
  - Remark: *je nach Architektur auch auf das letzte Element*

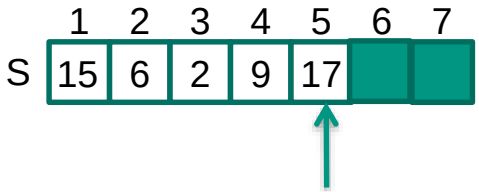


Stack

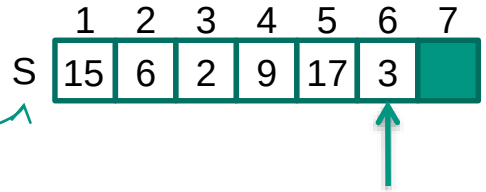
- top[s]:
  - Stackpointer
    (*hier auf die letzte besetzte Stelle*)



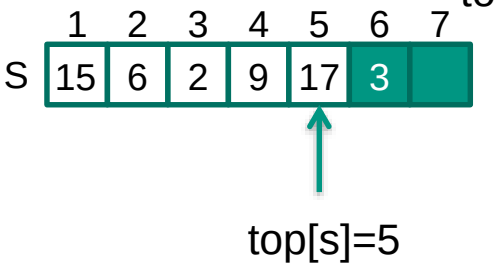
- Aufrufe:
  - push(s,17)



  - push(s,3)



  - pop(s)



- Remark:
  - *nur Stackpointer wird versetzt, altes Element „3“ wird nicht explizit gelöscht*

- Stack

IT Inhalt

6. Objektorientierung

- Grundidee und Motivation
- Klassen
- Objekte
- Datenkapselung

7. Datenstrukturen

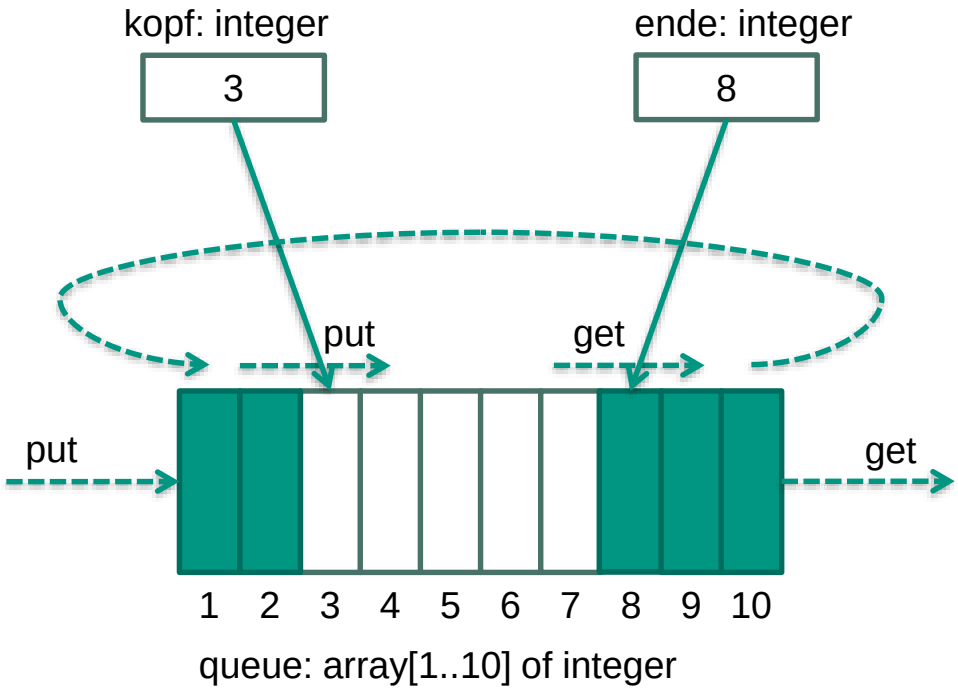
- Array
- Liste
- Stack
- Queue
- Hash-Tabelle
- Graph
- Baum
- Haufen

Queue (Warteschlange)

- FIFO-Prinzip (First In, First Out):
  - Lesen nur in Reihenfolge wie beim Schreiben
- Operationen:
  - `enqueue()` – hinten in die Schlange einreihen
  - `dequeue()` – vorderstes Element der Schlange holen
- Eine Warteschlange wird gewöhnlich als Liste implementiert.
  - Speichert beliebig viele Elemente, wenn intern als Liste realisiert

Queue

- Zwei Zeiger, Zeigerverwaltung



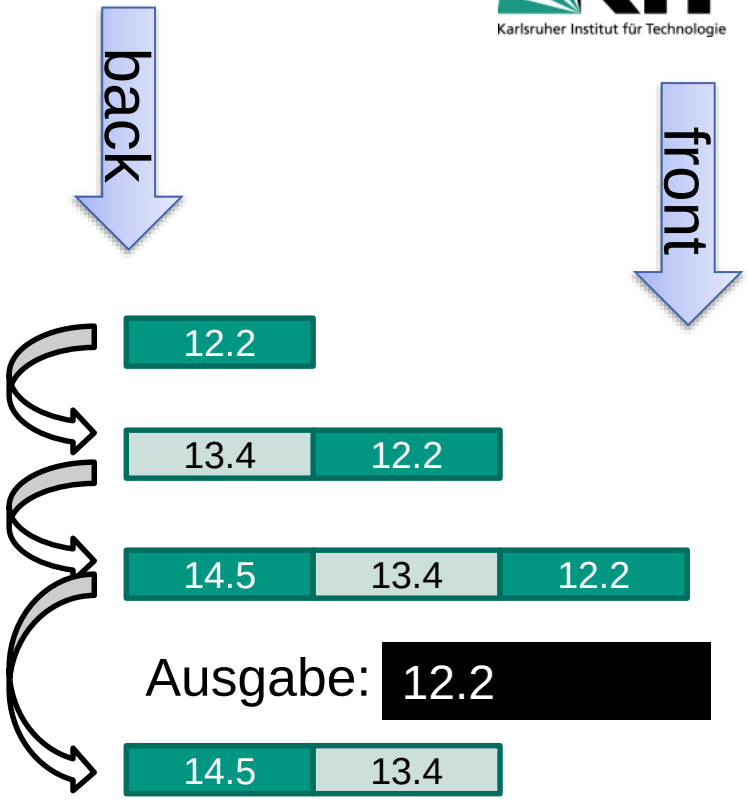
- Remark: *Hier als Vektor realisiert!*

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: 27.9

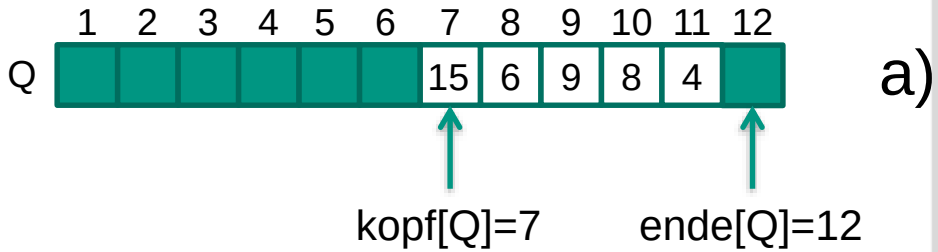
Queue implementiert mit Feld

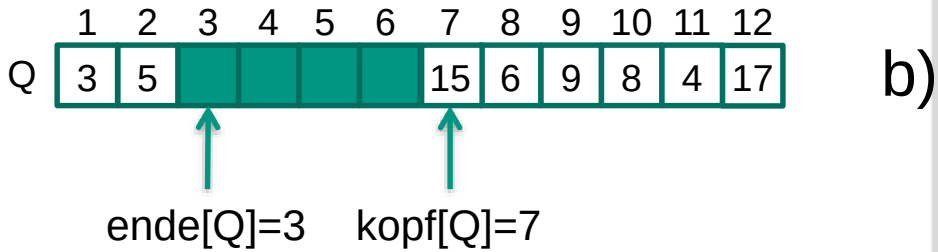
- Die Implementierung einer Warteschlange mithilfe eines Feldes Q[1..12].
- Elemente der Warteschlange befinden sich nur an den weißen Positionen.

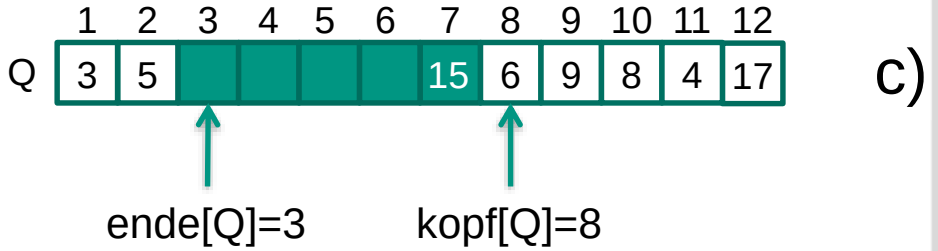
a) Die Warteschlange hat 5 Elemente an den Stellen Q[7..11].

b) Die Konfiguration der Warteschlange nach den Aufrufen
   a) enqueue(Q, 17)
   b) enqueue(Q, 3)
   c) enqueue (Q, 5)

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







- Queue

Queuing

IT Inhalt

6. Objektorientierung

- Grundidee und Motivation
- Klassen
- Objekte
- Datenkapselung

7. Datenstrukturen

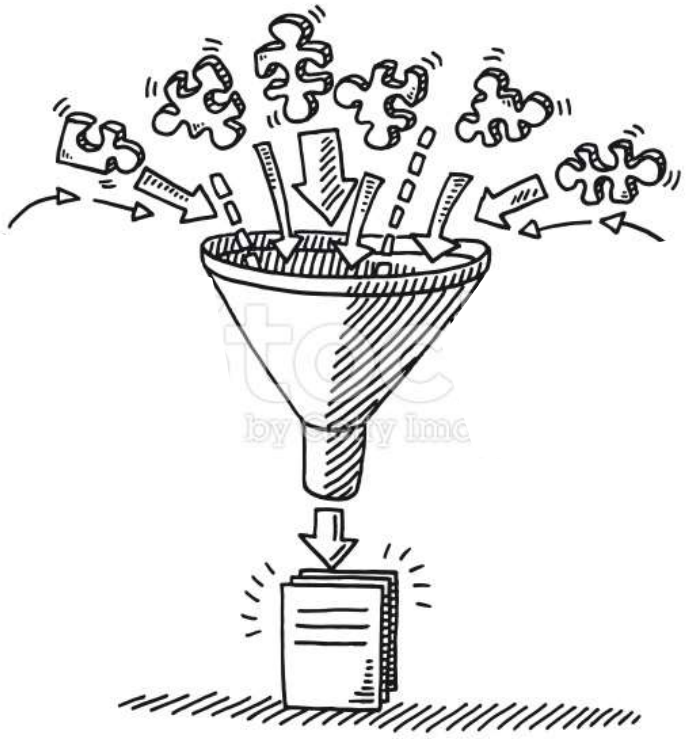
- Array
- Liste
- Stack
- Queue
- Hash-Tabelle
- Graph
- Baum
- Haufen

5) Hash-Tabellen

- Probleme bei Implementierung durch schon bekannte Datenstrukturen
  - Listen: Suche ist aufwändig und vor jedem Löschen und Einfügen (an sich effizient) ist eine Suche notwendig
  - Arrays: Suche schnell aber dynamisches Wachstum teuer, viel Speicher notwendig bei dünn besetzten Arrays
- Anwendung von Hashing:
  - Verfahren für dynamisch veränderliche Menge von Objekten mit effizienten Grundoperationen (z.B. Wörterbuchoperationen)
    - Suchen
    - Einfügen
    - Löschen
- Die mittlere Zeit zum Auffinden eines Elements in einer Hash-Tabelle ist unter realistischen Annahmen konstant.
  - Bei Array oder Liste linear abhängig von Anzahl der Elemente

Idee der Hashtabellen

- Eine mathematische Funktion berechnet die Position eines Objektes in einer Tabelle.
  - Dadurch erübrigt sich das Durchsuchen vieler Datenobjekte, bis das Zielobjekt gefunden wurde → mehr oder weniger konstante Zugriffszeit.
  - Man schließt anhand des Elementes selbst auf die Position, an der das Element gespeichert werden soll.
  - Bei gegebenem Schlüssel k wird nicht einfach k als Feldindex verwendet → der Index ergibt sich stattdessen aus einer Funktion angewandt auf k, der Hash-Funktion!
- Effizient ist dies umzusetzen, wenn sich jedes Objekt eindeutig einem Index in einem Array zuordnen lassen würde.



Hash-Tabellen

- Hash-Tabelle ist eine Verallgemeinerung eines gewöhnlichen Feldes
  - Jedes Element wird mit einem Schlüssel versehen
  - Element mit Schlüssel k wird an Position h(k) gespeichert
  - Um ein Element mit Schlüssel k wieder aufzufinden, muss lediglich an Position h(k) des Feldes nachgeschaut werden
- Einsatz einer Hash-Tabelle dann, wenn Anzahl der tatsächlich auftretenden Schlüssel klein ist im Vergleich zur Anzahl möglicher Schlüssel
  - Z.B. Namen bis zu 10 Buchstaben:
    - mehr als 141 Milliarden mögliche Namen (26^{10})
  - Z.B. KIT Studierende ca. 24 Tausend
    - Matrikelnummer 7stellig (1 Mio)
- Hash-Tabelle ist ein Feld, dessen Speicherbedarf proportional zur Anzahl tatsächlich gespeicherter Elemente ist
  - anstelle der Anzahl möglicher Elemente

Hash-Funktion

- Eine Hash-Funktion (auch Streuwertfunktion) ist eine Abbildung, die eine große Eingabemenge (die Schlüssel) auf eine kleinere Zielmenge (die Hashwerte) abbildet.
  - Sie ist daher nicht injektiv, da verschiedene Elemente der Definitionsmenge auf dasselbe Element der Zielmenge abgebildet werden können.
  - *Remark: Der Name „Hashfunktion“ stammt vom englischen Verb to hash, das sich als „zerhacken“ übersetzen lässt.*
- Divisionsmethode:
  - Jedes Element wird mit einem Schlüssel versehen
  - 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
  - *Remark: Eine gute Wahl für m ist eine Primzahl, die nicht zu nahe bei einer Potenz von 2 liegt.*

Zwischenübung: 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

Zwischenübung: Hashtabelle - Lsg.

- 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

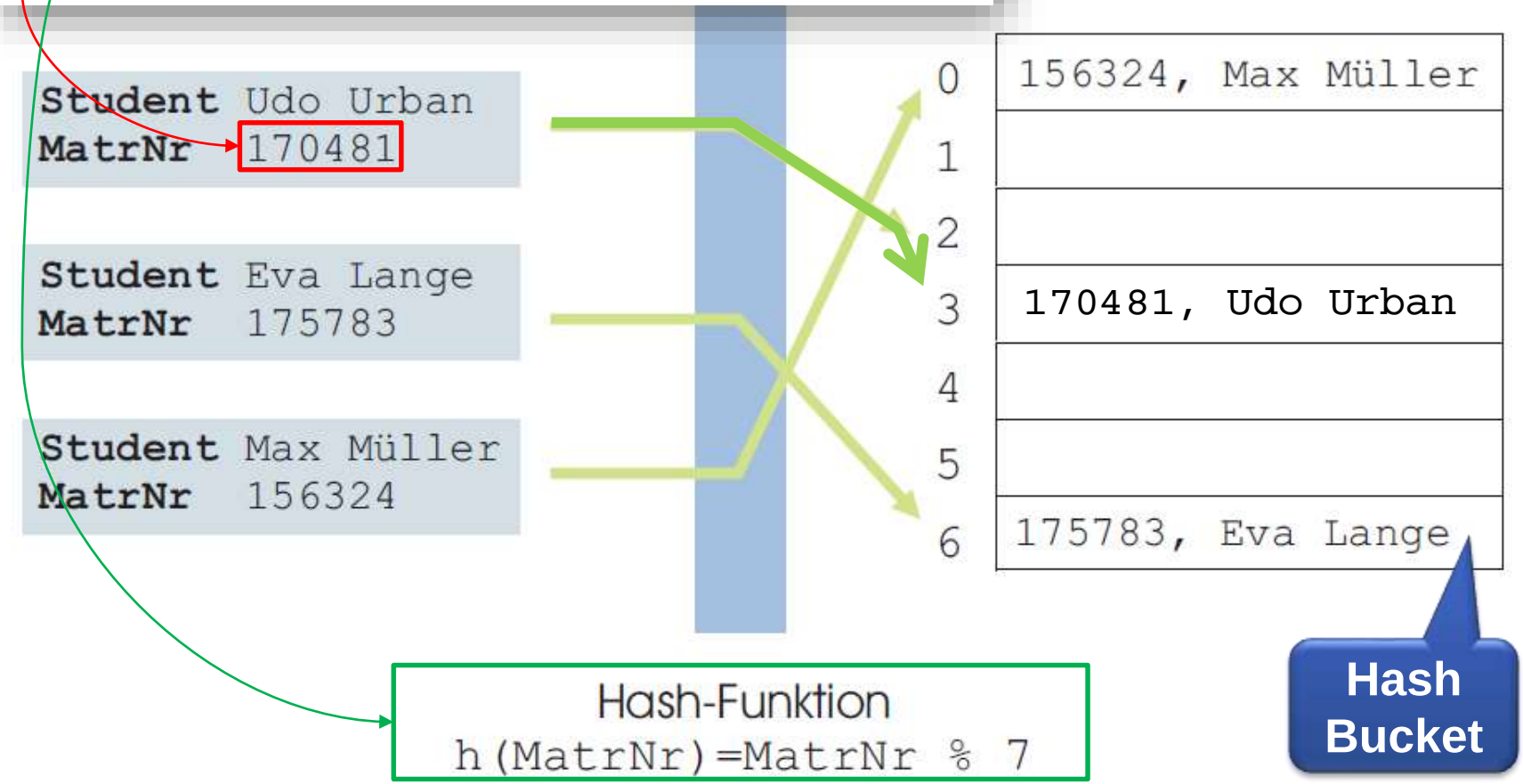
- Wie groß muss *m* sein?
- Geben Sie die Hashfunktion an

Lösung:
m soll eine Primzahl in der Nähe von 2000/3 sein
m soll nicht zu nahe bei einer 2er-Potenz liegen
m = 701 erfüllt beide Bedingungen

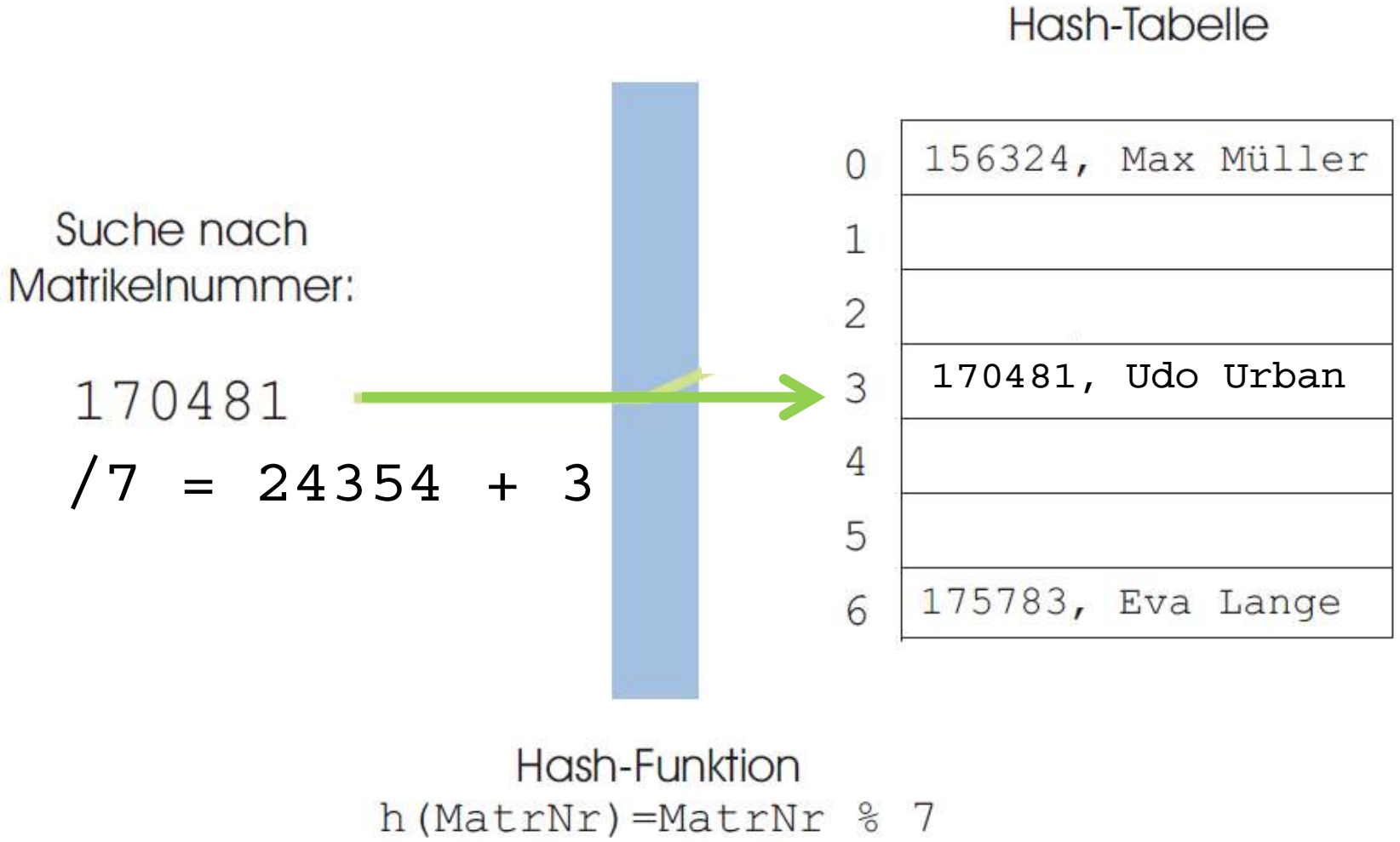
Es folgt:
h(k) = k mod 701 *für k = 0..1999*

Beispiel: Einfügen in einer Hash-Tabelle
(wenig Plätze → 7)

- Divisionsmethode:
  - Jedes Element wird mit einem Schlüssel versehen
  - 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



Beispiel: Suchen in einer Hash-Tabelle



Hash-Tabellen: Kollisionen

- Wertebereich ist durch Speichergröße M (“7 Studenten”) bestimmt
- Problem (s. zuvor):
  - Bei 21.000 Studenten → 3.000 pro Platz …
  - Hash-Funktion ist nicht injektiv, d.h. verschiedene Schlüssel können auf eine Adresse abgebildet werden → Kollisionen!
- Gute Hash-Funktionen (z.B. Primzahlen als Parameter) erzeugen möglichst zufällig gestreute Speicherzuordnung und machen dadurch Kollisionen unwahrscheinlich
- Kollisionen lassen sich aber meist nicht völlig vermeiden → erfordern Kollisionsbehandlung

Hash-Tabellen: Kollisionsbehandlung

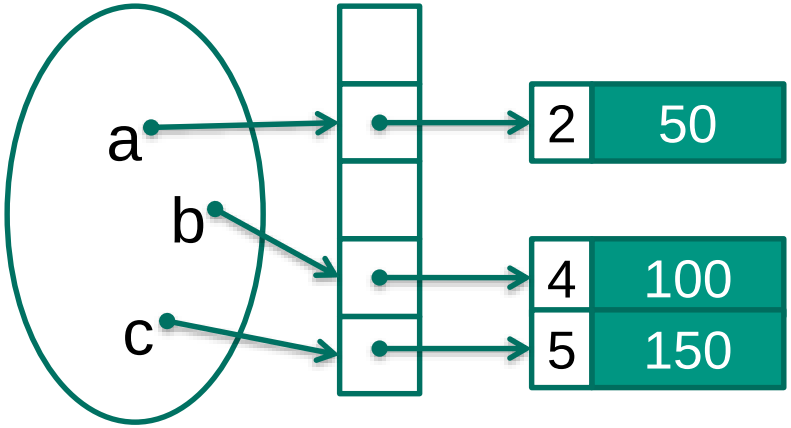
- Verkettete Liste:
  - der Eintrag in einer Hash-Tabelle verweist nicht auf ein einzelnes Element sondern auf eine Liste der dorthin ge-hashten Daten
  - Kann bei schlechter Hash-Funktion mit vielen Kollisionen zu Entartung führen (doch wieder “Liste” → Mehraufwand für Speicherung)
- Sondieren: (engl. Probing)
  - ist der Hash Bucket belegt, wird nach einem einfachen Muster ein anderer Platz gesucht, z.B. lineares Sondieren: Prüfen ob folgender Hash Bucket frei ist, erster freier wird genutzt
- „Doppeltes Hashen“
  - ist der Hash Bucket belegt, wird (ggf. wiederholt) ein weiterer Hash-Wert berechnet und diese Position getestet (s. Beispiel)

Hash-Tabelle 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: Hash-Tabelle

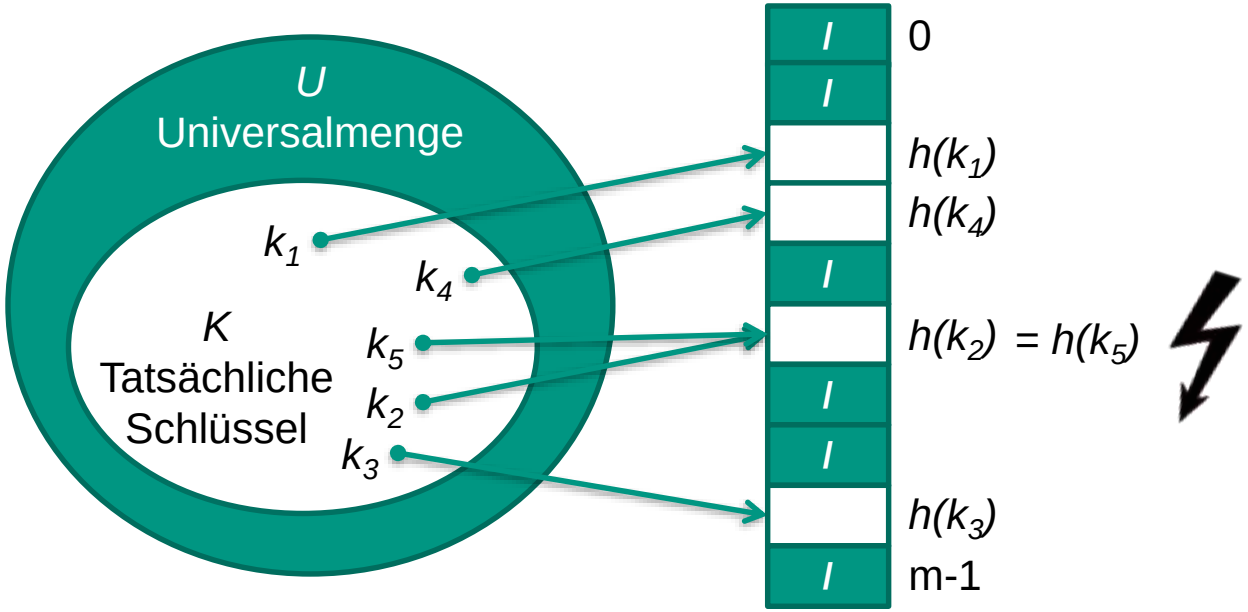
```
#include <iostream>
#include <unordered_map>
using namespace std;

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:

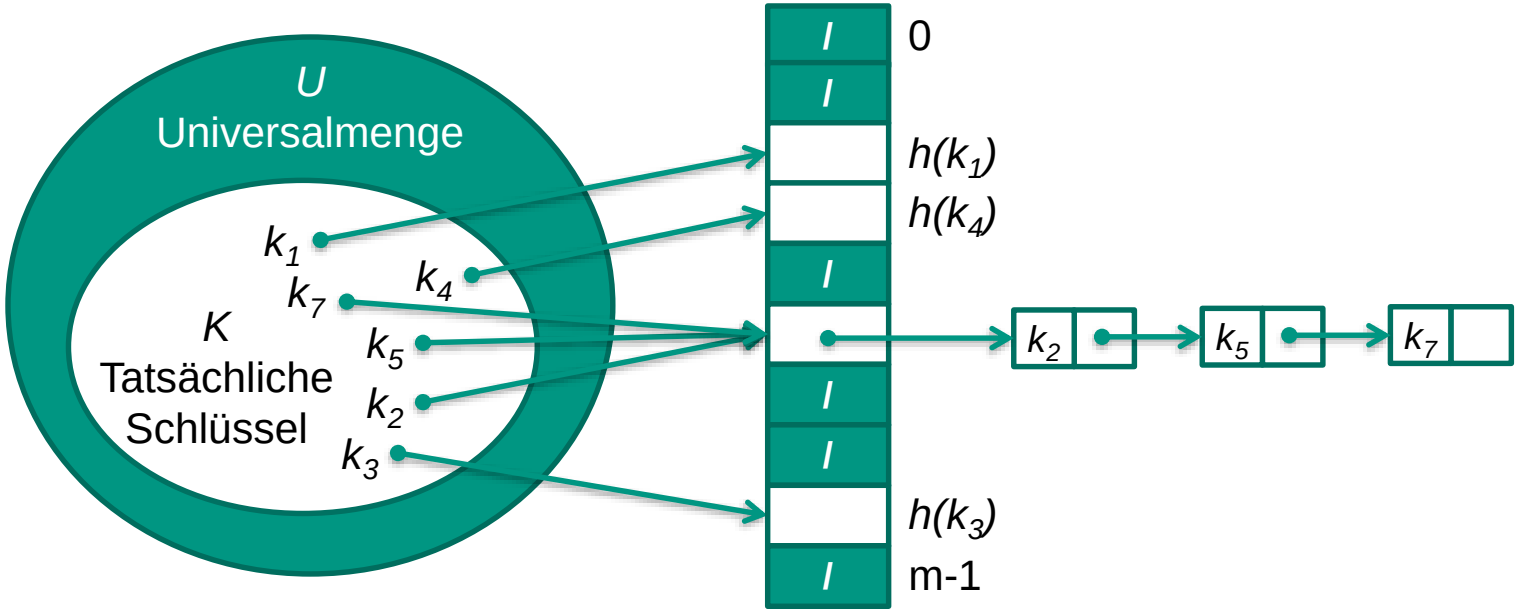
```
50
150
```

Kollisionsbehandlung mit verketteter Liste



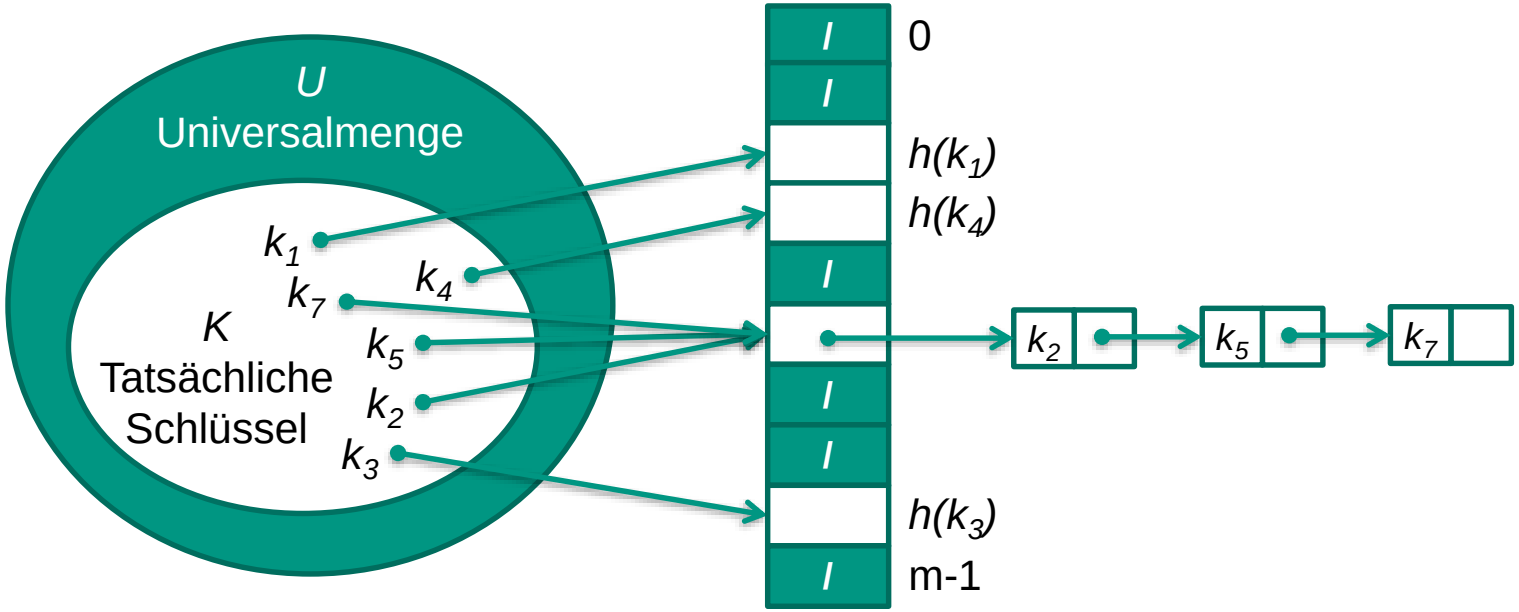
- Jeder Schlüssel der Universalmenge U = {0, 1, …, m) entspricht einem Index der Tabelle.
- Die Menge K = {1, 2, 3, 4, 5} der tatsächlichen Schlüssel bestimmt die Plätze der Tabelle, die Zeiger auf Elemente enthalten.
- Die anderen Plätze (stark schattiert) enthalten NIL.
- k_2 und k_5 werden auf den selben Platz abgebildet
- k_2 und k_5 kollidieren

Kollisionsbehandlung mit verketteter Liste



- Jeder Schlüssel der Universalmenge U = {0, 1, …, m) entspricht einem Index der Tabelle.
- Die Menge K = {1, 2, 3, 4, 5} der tatsächlichen Schlüssel bestimmt die Plätze der Tabelle, die Zeiger auf Elemente enthalten.
- Die anderen Plätze (stark schattiert) enthalten NIL.
- k_2 und k_5 werden auf den selben Platz abgebildet
- k_2 und k_5 kollidieren

Kollisionsbehandlung mit verketteter Liste



- Kollisionsauflösung durch Verkettung
  - Elemente der Hashtabelle sind verkettete Listen
- Um dies zu umgehen, verwendet man beim Doppel-Hashing eine Sondierungsfunktion, die eine sekundäre Hash-Funktion beinhaltet, und die angewendet wird, falls der durch die primäre Hash-Funktion berechnete Index bereits besetzt ist: $h_j(k) = (h(k) + h'(k) \cdot j) \mod m$

Kollisionsauflösung durch doppeltes Hashing (1)

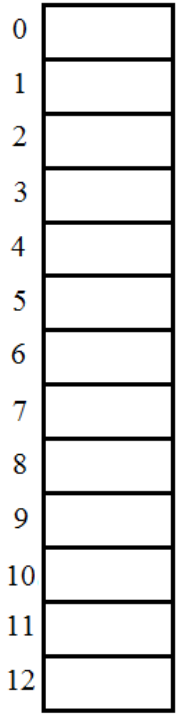
- Hashtabelle

- Hashtabelle mit Länge m = 13
- Hashfunktion für Doppeltes Hashing:

$$h(k,i) = (h_1(k) + ih_2(k)) \bmod m$$

$$h_1(k) = k \bmod m$$
$$h_2(k) = 1 + (k \bmod m')$$

h_1 und h_2 sind die dabei die zwei Hilfshashfunktionen unserer Hashfunktion h(k,i).



Quelle: Uni Jena

Kollisionsauflösung durch doppeltes Hashing (1)

Hashtabelle

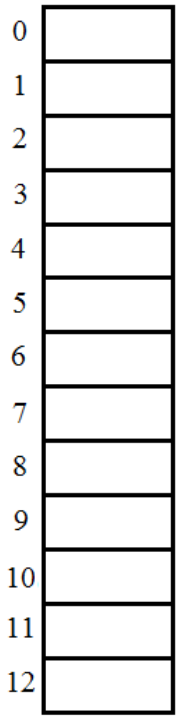
- Hashtabelle mit Länge m = 13
- Hashfunktion für Doppeltes Hashing:

$h(k,i) = (h_1(k) + ih_2(k)) \bmod m$

$h_1(k) = k \bmod m$

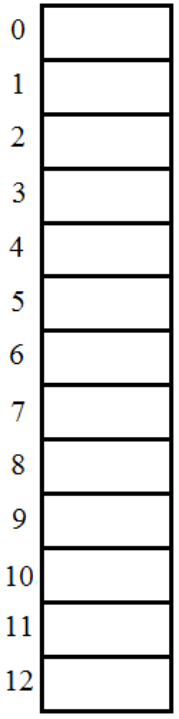
$h_2(k) = 1 + (k \bmod m')$

h_1 und h_2 sind die dabei die zwei Hilfshashfunktionen unserer Hashfunktion h(k,i).



Vorgehensweise

Mit Hilfe von h_1 versucht man eine erste Sondierung.
Falls die Position schon besetzt ist geht man schrittweise um $h_2(k) \bmod m$ weiter und schaut ob ein Tabellenplatz frei ist.
i ist dabei unser Zähler, der Startwert ist 0 (die erste Sondierung erfolgt nur nach h_1) und wird dann jeweils um 1 erhöht bis der Schlüssel eingefügt werden kann.
$h_2(k)$ muss dabei offensichtlich teilerfremd zu m sein, um die komplette Hashtabelle durchsuchen zu können.
Man wählt m' dann so, dass h_2 immer eine pos. ganze Zahl kleiner m ergibt.
In unserem Beispiel wäre m' = 12 eine solche Zahl, da $h_2 = 1 + (k \bmod m')$ immer kleiner als m ist.



Quelle: Uni Jena

Kollisionsauflösung durch doppeltes Hashing (2)

- Erste Sondierung

Nun wollen wir ein paar Schlüssel der Reihe nach in unsere noch leere Hashtabelle einfügen.

Wir starten mit dem Schlüssel k = 17

Man nehme die Hashfunktion h(k,i) mit den Werten k = 17 und i = 0 (Startwert).

$h(17,0) = (h_1(17) + 0h_2(17)) \bmod 13$

Die erste Sondierung erfolgt nur nach der Hilfshashfunktion h_1.

$h_1(17) = 17 \bmod 13$

$h_1(17) = 4$

$h(17,0) = (h_1(17) + 0h_2(17)) \bmod 13$

$h(17,0) = 4$

Die Hashfunktion hat für den Schlüssel 17 den Tabellenplatz 4 errechnet. Da der Tabellenplatz noch frei ist fügen wir den Schlüssel dort ein und sind fertig.

0	
1	
2	
3	
4	17
5	
6	
7	
8	
9	
10	
11	
12	

Quelle: Uni Jena

Kollisionsauflösung durch doppeltes Hashing (2)

Erste Sondierung

Nun wollen wir ein paar Schlüssel der Reihe nach in unsere noch leere Hashtabelle einfügen.

Wir starten mit dem Schlüssel k = 17

Man nehme die Hashfunktion h(k,i) mit den Werten k = 17 und i = 0 (Startwert).

$h(17,0) = (h_1(17) + 0h_2(17)) \bmod 13$

Die erste Sondierung erfolgt nur nach der Hilfshashfunktion h_1.

$h_1(17) = 17 \bmod 13$

$h_1(17) = 4$

$h(17,0) = (h_1(17) + 0h_2(17)) \bmod 13$

$h(17,0) = 4$

Die Hashfunktion hat für den Schlüssel 17 den Tabellenplatz 4 errechnet. Da der Tabellenplatz noch frei ist fügen wir den Schlüssel dort ein und sind fertig.

0	
1	
2	
3	
4	17
5	
6	
7	
8	
9	
10	
11	
12	

Befüllung

Jetzt wollen wir noch ein paar weitere Schlüssel in unsere Hashtabelle einfügen:

23, 1, 8, 15 und 28

Zum Einfügen des Schlüssels k = 23 nehmen wir uns wieder die Hashfunktion mit dem Startwert i = 0 und schauen nach, ob der errechnete Wert ein freier Tabellenplatz ist.

$h(23,0) = (h_1(23) + 0h_2(23)) \bmod 13$

Die erste Sondierung erfolgt nur nach h_1.

$h_1(23) = 23 \bmod 13 = 10$

$h(23,0) = h_1(23) = 10$

Tabellenplatz 10 ist noch frei, also fügen wir den Schlüssel k = 23 an der Stelle ein.

0	
1	
2	
3	
4	17
5	
6	
7	
8	
9	
10	23
11	
12	

Quelle: Uni Jena

Kollisionsauflösung durch doppeltes Hashing (3)

- Weitere Befüllung

Das Einfügen der Schlüssel k = 1, k = 8 und k = 15 folgt nun analog zu den vorherigen Einfügeoperationen:

$h(1,0) = (h_1(1) + 0h_2(1)) \bmod 13$

$h_1(1) = 1 \bmod 13$

$h(1,0) = h_1(1) = 1$

Tabellenplatz 1 ist noch frei, wir können also den Schlüssel k = 1 dort einfügen.

$h(8,0) = (h_1(8) + 0h_2(8)) \bmod 13$

$h_1(8) = 8 \bmod 13$

$h(8,0) = h_1(8) = 8$

$h(15,0) = (h_1(15) + 0h_2(15)) \bmod 13$

$h_1(15) = 15 \bmod 13$

$h(1,0) = h_1(15) = 2$

Die Plätze für die Schlüssel k = 8 und k = 15 sind auch noch frei, können also problemlos eingefügt werden.

0	
1	1
2	15
3	
4	17
5	
6	
7	
8	8
9	
10	23
11	
12	

Quelle: Uni Jena

Kollisionsauflösung durch doppeltes Hashing (3)

- Weitere Befüllung

Das Einfügen der Schlüssel k = 1, k = 8 und k = 15 folgt nun analog zu den vorherigen Einfügeoperationen:

$h(1,0) = (h_1(1) + 0h_2(1)) \bmod 13$

$h_1(1) = 1 \bmod 13$

$h(1,0) = h_1(1) = 1$

Tabellenplatz 1 ist noch frei, wir können also den Schlüssel k = 1 dort einfügen.

$h(8,0) = (h_1(8) + 0h_2(8)) \bmod 13$

$h_1(8) = 8 \bmod 13$

$h(8,0) = h_1(8) = 8$

$h(15,0) = (h_1(15) + 0h_2(15)) \bmod 13$

$h_1(15) = 15 \bmod 13$

$h(1,0) = h_1(15) = 2$

Die Plätze für die Schlüssel k = 8 und k = 15 sind auch noch frei, können also problemlos eingefügt werden.

0	
1	1
2	15
3	
4	17
5	
6	
7	
8	8
9	
10	23
11	
12	

- Kollision erkannt

Einfügen des Schlüssels k = 28

Wir gehen wieder analog vor und überprüfen zuerst, ob der Tabellenplatz für i = 0 noch frei ist.

$h(28,0) = (h_1(28) + 0h_2(28)) \bmod 13$

$h_1(28) = 28 \bmod 13$

$h(28,0) = h_1(28) = 2$

Der Tabellenplatz 2 ist leider schon von Schlüssel k = 15 belegt, wir müssen also für den Schlüssel k = 28 einen anderen freien Tabellenplatz finden. Daher erhöhen wir jetzt unseren Zähler i um 1, damit die Hilfshashfunktion h_2 jetzt auch ihre

Berücksichtigung bei der Findung eines freien Tabellenplatzes hat.

0	
1	1
2	15
3	
4	17
5	
6	
7	
8	8
9	
10	23
11	
12	

Quelle: Uni Jena

Kollisionsauflösung durch doppeltes Hashing (4)

- Kollision behoben

Wir berechnen einen neuen Tabellenplatz jetzt mit den folgenden Werten:

$k = 28$ und $i = 1$

Die Hashfunktion hat dann folgende Gestalt:

$h(28,1) = (h_1(28) + 1h_2(28)) \bmod 13$

$h_1(28) = 28 \bmod 13$

$h_1(28) = 2$

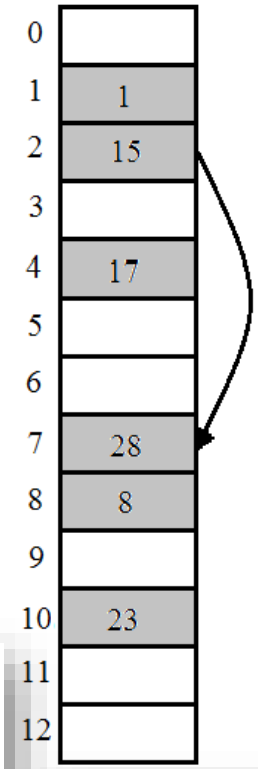
$h_2(28) = 1 + (28 \bmod 12)$

$h_2(28) = 5$

$h(28,1) = (h_1(28) + 1h_2(28)) \bmod 13$

$h(28,1) = 7$

Tabellenplatz 7 ist noch frei, wir können also den Schlüssel k = 28 dort einfügen.



$h(k,i) = (h_1(k) + ih_2(k)) \bmod m$

$h_1(k) = k \bmod m$

$h_2(k) = 1 + (k \bmod m')$

h_1 und h_2 sind die dabei die zwei Hilfshashfunktionen unserer Hashfunktion h(k,i).

Man wählt m' dann so, dass h_2 immer eine pos. ganze Zahl kleiner m ergibt.
In unserem Beispiel wäre m' = 12 eine solche Zahl, da $h_2 = 1 + (k \bmod m')$ immer kleiner als m ist.

Quelle: Uni Jena

Kollisionsauflösung durch doppeltes Hashing (4)

- Kollision behoben

Wir berechnen einen neuen Tabellenplatz jetzt mit den folgenden Werten:

$k = 28$ und $i = 1$

Die Hashfunktion hat dann folgende Gestalt:

$h(28,1) = (h_1(28) + 1h_2(28)) \bmod 13$

$h_1(28) = 28 \bmod 13$

$h_1(28) = 2$

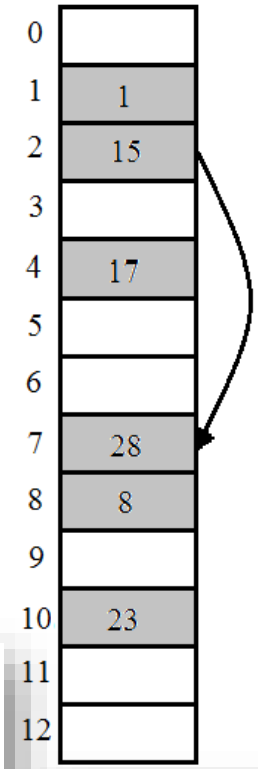
$h_2(28) = 1 + (28 \bmod 12)$

$h_2(28) = 5$

$h(28,1) = (h_1(28) + 1h_2(28)) \bmod 13$

$h(28,1) = 7$

Tabellenplatz 7 ist noch frei, wir können also den Schlüssel k = 28 dort einfügen.



$h(k,i) = (h_1(k) + ih_2(k)) \bmod m$

$h_1(k) = k \bmod m$

$h_2(k) = 1 + (k \bmod m')$

h_1 und h_2 sind die dabei die zwei Hilfshashfunktionen unserer Hashfunktion h(k,i).

Man wählt m' dann so, dass h_2 immer eine pos. ganze Zahl kleiner m ergibt.
In unserem Beispiel wäre m' = 12 eine solche Zahl, da $h_2 = 1 + (k \bmod m')$ immer kleiner als m ist.

- Weiteres Befüllen

Unsere Hastabelle ist noch nicht voll, wir fügen also weitere Schlüssel ein:

44, 26, 38

$h(44,0) = (h_1(44) + 0h_2(44)) \bmod 13$

$h_1(44) = 44 \bmod 13$

$h(44,0) = h_1(44) = 5$

$h(26,0) = (h_1(26) + 0h_2(26)) \bmod 13$

$h_1(26) = 26 \bmod 13$

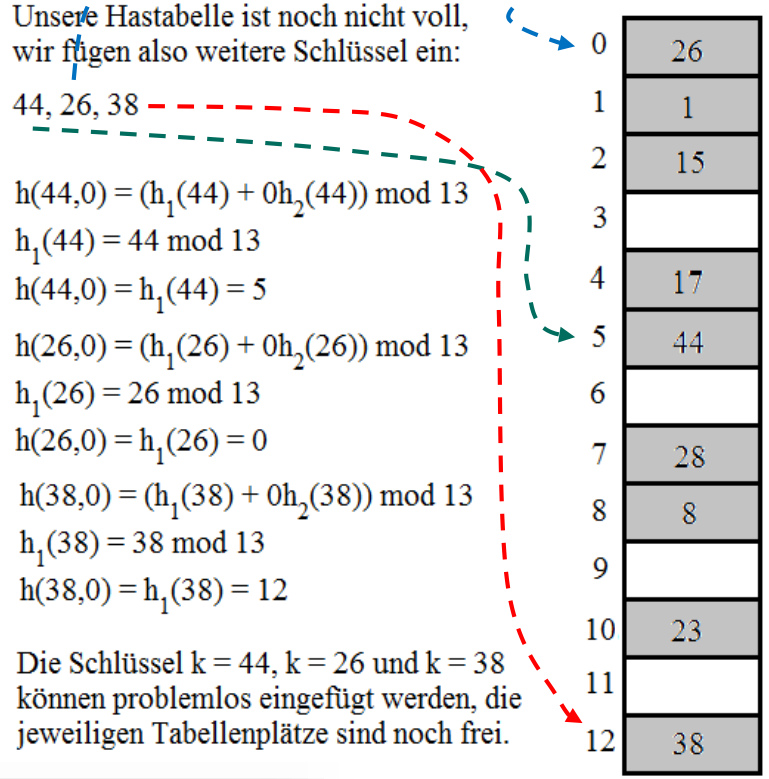
$h(26,0) = h_1(26) = 0$

$h(38,0) = (h_1(38) + 0h_2(38)) \bmod 13$

$h_1(38) = 38 \bmod 13$

$h(38,0) = h_1(38) = 12$

Die Schlüssel k = 44, k = 26 und k = 38 können problemlos eingefügt werden, die jeweiligen Tabellenplätze sind noch frei.



Quelle: Uni Jena

Kollisionsauflösung durch doppeltes Hashing (5)

Wir haben noch 4 freie Plätze in unserer Hashtabelle, also fügen wir noch 4 weitere Schlüssel ein:

86, 0, 32, 75

$h(86,0) = (h_1(86) + 0h_2(86)) \bmod 13$

$h_1(86) = 86 \bmod 13$

$h(86,0) = h_1(86) = 8$

Tabellenplatz 8 ist schon von Schlüssel 8 belegt, also erhöhen wir wieder i um 1.

$h(86,1) = (h_1(86) + 1h_2(86)) \bmod 13$

$h_1(86) = 86 \bmod 13 = 8$

$h_2(86) = 1 + (86 \bmod 12) = 3$

$h(86,1) = (8 + 3) \bmod 13$

$h(86,1) = 11$

Wir fügen Schlüssel k = 86 beim freien Tabellenplatz 11 ein.

0	26
1	1
2	15
3	
4	17
5	44
6	
7	28
8	8
9	
10	23
11	86
12	38

$h_1(k) = k \bmod m$

$h_2(k) = 1 + (k \bmod m')$

h_1 und h_2 sind die dabei die zwei Hilfshashfunktionen unserer Hashfunktion h(k,i).

Man wählt m' dann so, dass h_2 immer eine pos. ganze Zahl kleiner m ergibt.
In unserem Beispiel wäre m' = 12 eine solche Zahl, da $h_2 = 1 + (k \bmod m')$ immer kleiner als m ist.

Quelle: Uni Jena

Kollisionsauflösung durch doppeltes Hashing (5)

Wir haben noch 4 freie Plätze in unserer Hashtabelle, also fügen wir noch 4 weitere Schlüssel ein:

86, 0, 32, 75

$h(86,0) = (h_1(86) + 0h_2(86)) \bmod 13$

$h_1(86) = 86 \bmod 13$

$h(86,0) = h_1(86) = 8$

Tabellenplatz 8 ist schon von Schlüssel 8 belegt, also erhöhen wir wieder i um 1.

$h(86,1) = (h_1(86) + 1h_2(86)) \bmod 13$

$h_1(86) = 86 \bmod 13 = 8$

$h_2(86) = 1 + (86 \bmod 12) = 3$

$h(86,1) = (8 + 3) \bmod 13$

$h(86,1) = 11$

Wir fügen Schlüssel k = 86 beim freien Tabellenplatz 11 ein.

0	26
1	1
2	15
3	
4	17
5	44
6	
7	28
8	8
9	
10	23
11	86
12	38

Jetzt den Schlüssel k = 0

$h(0,0) = (h_1(0) + 0h_2(0)) \bmod 13$

$h_1(0) = 0 \bmod 13$

$h(0,0) = h_1(0) = 0$

Tabellenplatz belegt, weiter mit i = 1

$h(0,1) = (h_1(0) + 1h_2(0)) \bmod 13$

$h_1(0) = 0 \bmod 13 = 0$

$h_2(0) = 1 + (0 \bmod 12) = 1$

$h(0,1) = (0 + 1) \bmod 13 = 1$

Der Tabellenplatz ist auch schon belegt, also wieder i um eins erhöhen:

$h(0,2) = (h_1(0) + 2h_2(0)) \bmod 13$

h_1 und h_2 ändern sich dabei nicht, man muss die Hilfshashfunktionen also nicht jedes mal neu berechnen sondern kann die konstanten Werte direkt in die Hashfunktion einsetzen.

0	26
1	1
2	15
3	
4	17
5	44
6	
7	28
8	8
9	
10	23
11	86
12	38

$h_1(k) = k \bmod m$

$h_2(k) = 1 + (k \bmod m')$

h_1 und h_2 sind die dabei die zwei Hilfshashfunktionen unserer Hashfunktion h(k,i).

Man wählt m' dann so, dass h_2 immer eine pos. ganze Zahl kleiner m ergibt.
In unserem Beispiel wäre m' = 12 eine solche Zahl, da $h_2 = 1 + (k \bmod m')$ immer kleiner als m ist.

Quelle: Uni Jena

Kollisionsauflösung durch doppeltes Hashing (6)

$h(0,2) = (h_1(0) + 2h_2(0)) \bmod 13$

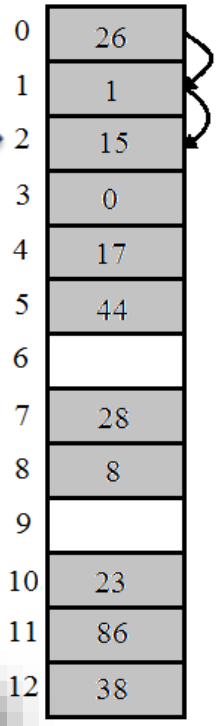
$h(0,2) = (0 + 2) \bmod 13 = 2$

Auch Tabellenplatz 2 ist schon belegt, also erneut den Zähler i um eins erhöhen!

$h(0,3) = (h_1(0) + 3h_2(0)) \bmod 13$

$h(0,3) = (0 + 3) \bmod 13 = 3$

Tabellenplatz 3 ist noch frei, wir tragen also dort unseren Schlüssel k = 0 ein.



$h(k,i) = (h_1(k) + ih_2(k)) \bmod m$

$h_1(k) = k \bmod m$

$h_2(k) = 1 + (k \bmod m')$

h_1 und h_2 sind die dabei die zwei Hilfshashfunktionen unserer Hashfunktion h(k,i).

Man wählt m' dann so, dass h_2 immer eine pos. ganze Zahl kleiner m ergibt.
In unserem Beispiel wäre m' = 12 eine solche Zahl, da $h_2 = 1 + (k \bmod m')$ immer kleiner als m ist.

Quelle: Uni Jena

Kollisionsauflösung durch doppeltes Hashing (6)

$h(0,2) = (h_1(0) + 2h_2(0)) \bmod 13$

$h(0,2) = (0 + 2) \bmod 13 = 2$

Auch Tabellenplatz 2 ist schon belegt, also erneut den Zähler i um eins erhöhen!

$h(0,3) = (h_1(0) + 3h_2(0)) \bmod 13$

$h(0,3) = (0 + 3) \bmod 13 = 3$

Tabellenplatz 3 ist noch frei, wir tragen also dort unseren Schlüssel k = 0 ein.

0	26
1	1
2	15
3	0
4	17
5	44
6	
7	28
8	8
9	
10	23
11	86
12	38

Der Schlüssel k = 32 ist wieder sehr einfach einzufügen:

$h(32,0) = (h_1(32) + 0h_2(32)) \bmod 13$

$h_1(32) = 32 \bmod 13$

$h(32,0) = h_1(32) = 6$

Der Tabellenplatz 6 ist noch frei und wir fügen den Schlüssel k = 32 an dieser Stelle ein.

0	26
1	1
2	15
3	0
4	17
5	44
6	32
7	28
8	8
9	
10	23
11	86
12	38

$h(k,i) = (h_1(k) + ih_2(k)) \bmod m$

$h_1(k) = k \bmod m$

$h_2(k) = 1 + (k \bmod m')$

h_1 und h_2 sind die dabei die zwei Hilfshashfunktionen unserer Hashfunktion h(k,i).

Man wählt m' dann so, dass h_2 immer eine pos. ganze Zahl kleiner m ergibt.
In unserem Beispiel wäre m' = 12 eine solche Zahl, da $h_2 = 1 + (k \bmod m')$ immer kleiner als m ist.

Quelle: Uni Jena

Kollisionsauflösung durch doppeltes Hashing (7)

Jetzt noch den Schlüssel k = 75 einfügen:

$h_1(75) = 75 \bmod 13 = 10$

$h_2(75) = 1 + (75 \bmod 12) = 4$

$h(75,0) = (h_1(75) + 0h_2(75)) \bmod 13$

$h(75,0) = (10 + 0) \bmod 13 = 10$

$h(75,1) = (h_1(75) + 1h_2(75)) \bmod 13$

$h(75,1) = (10 + 4)$ mode $13 = 1$

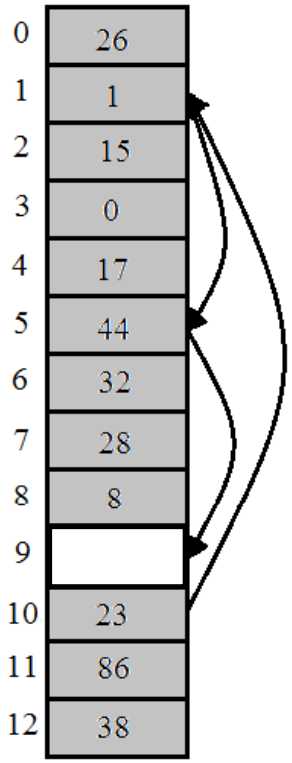
$h(75,2) = (h_1(75) + 2h_2(75)) \bmod 13$

$h(75,2) = (10 + 8)$ mode $13 = 5$

$h(75,3) = (h_1(75) + 3h_2(75)) \bmod 13$

$h(75,3) = (10 + 12)$ mode $13 = 9$

Tabellenplatz 10, 1 und 5 sind jeweils schon besetzt, aber bei Tabellenplatz 9 können wir den Schlüssel k = 75 in die Hashtabelle einfügen.



Quellen:
- Prof. Dr. Thomas H. Cormen: Introductions to Algorithms
- Prof. Dr. Joachim Giesen (Uni Jena): Algorithmen und Datenstrukturen (Vorlesungsmitschriften)

Quelle: Uni Jena

Kollisionsauflösung durch doppeltes Hashing (7)

Jetzt noch den Schlüssel k = 75 einfügen:

$h_1(75) = 75 \bmod 13 = 10$

$h_2(75) = 1 + (75 \bmod 12) = 4$

$h(75,0) = (h_1(75) + 0h_2(75)) \bmod 13$

$h(75,0) = (10 + 0) \bmod 13 = 10$

$h(75,1) = (h_1(75) + 1h_2(75)) \bmod 13$

$h(75,1) = (10 + 4)$ mode $13 = 1$

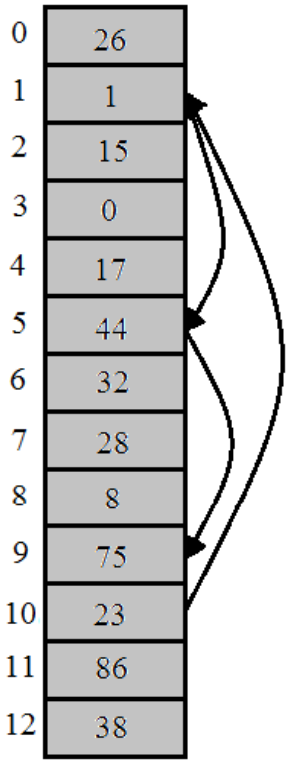
$h(75,2) = (h_1(75) + 2h_2(75)) \bmod 13$

$h(75,2) = (10 + 8)$ mode $13 = 5$

$h(75,3) = (h_1(75) + 3h_2(75)) \bmod 13$

$h(75,3) = (10 + 12)$ mode $13 = 9$

Tabellenplatz 10, 1 und 5 sind jeweils schon besetzt, aber bei Tabellenplatz 9 können wir den Schlüssel k = 75 in die Hashtabelle einfügen.



Die fertige Hashtabelle sieht also so aus. Weitere Schlüssel können nicht hinzugefügt werden, da alle Speicherplätze der Hashtabelle schon belegt sind.

Das Löschen von Schlüsselwerten aus der Hashtabelle gestaltet sich problematisch, da man darauf achten muss ob andere Schlüsselwerte dann noch erreichbar sind (Stichwort: Kollision beim Einfügen)

0	26
1	1
2	15
3	0
4	17
5	44
6	32
7	28
8	8
9	75
10	23
11	86
12	38

Quellen:
- Prof. Dr. Thomas H. Cormen: Introductions to Algorithms
- Prof. Dr. Joachim Giesen (Uni Jena): Algorithmen und Datenstrukturen (Vorlesungsmitschriften)

Quelle: Uni Jena

Kollisionsauflösung durch doppeltes Hashing (7)

Jetzt noch den Schlüssel k = 75 einfügen:

$h_1(75) = 75 \bmod 13 = 10$

$h_2(75) = 1 + (75 \bmod 12) = 4$

$h(75,0) = (h_1(75) + 0h_2(75)) \bmod 13$

$h(75,0) = (10 + 0) \bmod 13 = 10$

$h(75,1) = (h_1(75) + 1h_2(75)) \bmod 13$

$h(75,1) = (10 + 4)$ mode $13 = 1$

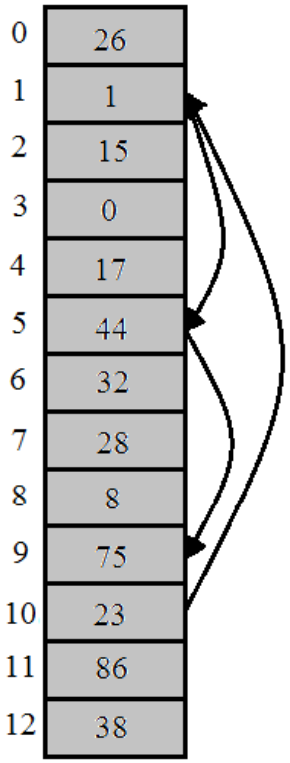
$h(75,2) = (h_1(75) + 2h_2(75)) \bmod 13$

$h(75,2) = (10 + 8)$ mode $13 = 5$

$h(75,3) = (h_1(75) + 3h_2(75)) \bmod 13$

$h(75,3) = (10 + 12)$ mode $13 = 9$

Tabellenplatz 10, 1 und 5 sind jeweils schon besetzt, aber bei Tabellenplatz 9 können wir den Schlüssel k = 75 in die Hashtabelle einfügen.



Die fertige Hashtabelle sieht also so aus. Weitere Schlüssel können nicht hinzugefügt werden, da alle Speicherplätze der Hashtabelle schon belegt sind.

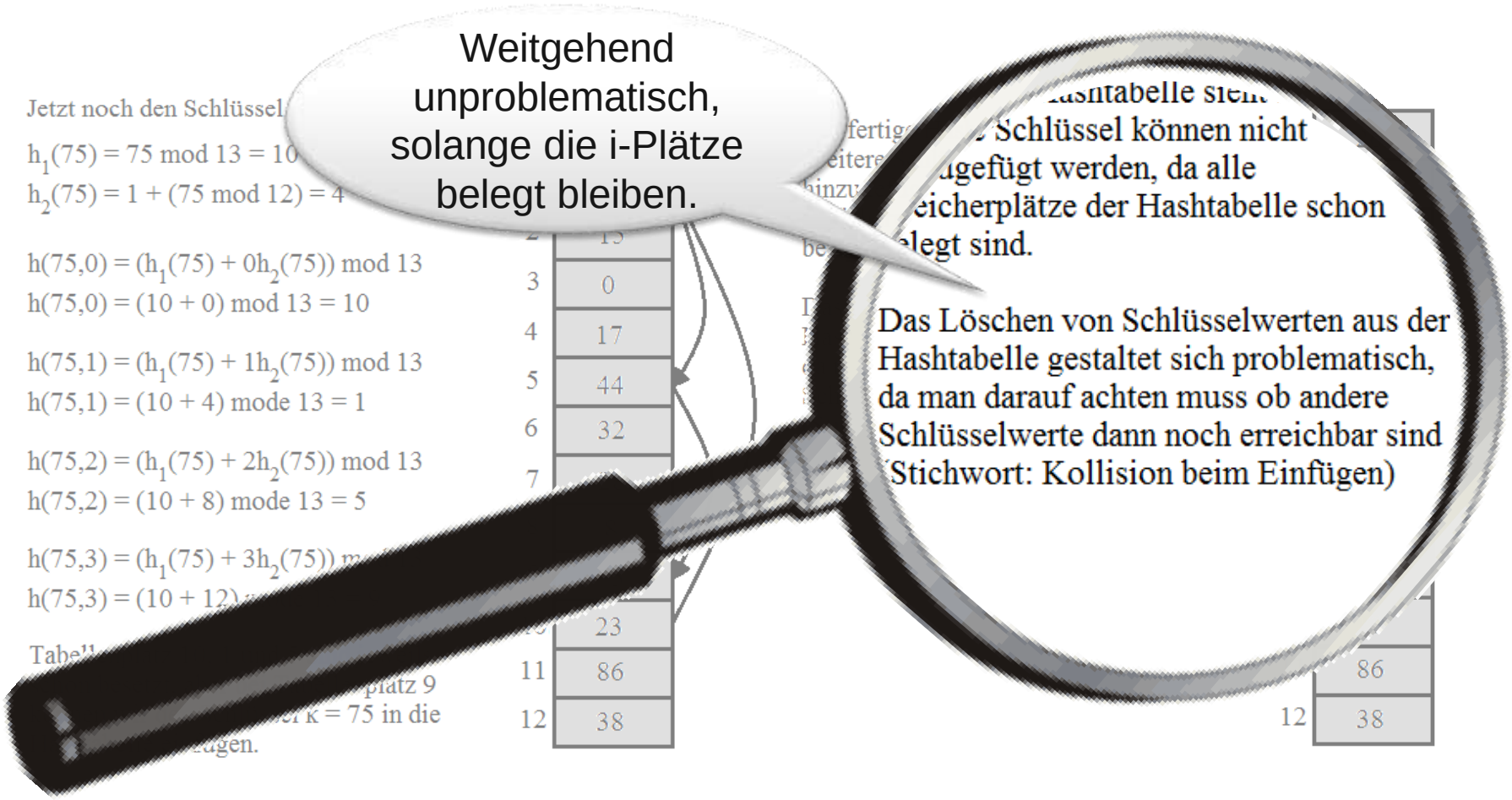
Das Löschen von Schlüsselwerten aus der Hashtabelle gestaltet sich problematisch, da man darauf achten muss ob andere Schlüsselwerte dann noch erreichbar sind (Stichwort: Kollision beim Einfügen)

0	26
1	1
2	15
3	0
4	17
5	44
6	32
7	28
8	8
9	75
10	23
11	86
12	38

Quellen:
- Prof. Dr. Thomas H. Cormen: Introductions to Algorithms
- Prof. Dr. Joachim Giesen (Uni Jena): Algorithmen und Datenstrukturen (Vorlesungsmitschriften)

Quelle: Uni Jena

Löschen von Schlüssewerten



Quellen:
- Prof. Dr. Thomas H. Cormen: Introductions to Algorithms
- Prof. Dr. Joachim Giesen (Uni Jena): Algorithmen und Datenstrukturen (Vorlesungsmitschriften)

Kriterien für eine gute Hash-Funktion

- Geringe Kollisionswahrscheinlichkeit
- Gleichverteilung der Hashwerte.
- Effizienz: schnell berechenbar, geringer Speicherverbrauch, die
- Eingabedaten nur einmal lesen.
- Jeder Ergebniswert soll möglich sein (Surjektivität).
- Hashwert viel kleiner als Eingabedaten (Kompression)
- Zusammenfassung Maxime
  - zerhacken, zerkleinern → verdichten

- Hashtabelle