

# Vorlesung Wirtschaftsinformatik

---

**Prof. Dr. Clemens Espe, MBA**  
**Fakultät für Informatik**  
**Wirtschaftsinformatik**  
**Hochschule Augsburg**



# Inhaltsverzeichnis

---

## 4. Algorithmen

1. Definition und Beschreibung von Algorithmen
2. Komplexität und O-Notation
3. Suchalgorithmen

## 5. Datenstrukturen

1. Elementare Datenstrukturen und Listen
2. Binäre Suchbäume
3. Balancierte Suchbäume
4. Hashtabellen

## 6. Datenkommunikation

1. Kommunikationssysteme
2. Internet und Web 2.0

# Kapitel 5: Datenstrukturen

## 5.1. Elementare Datenstrukturen und Listen

---

# Elementare Datenstrukturen

## Basis Datentypen

---

**char:** Alle darstellbaren Zeichen,

**int:** Alle ganzen darstellbaren Zahlen,

**float:** Alle darstellbaren Gleitkommazahlen mit einfacher Genauigkeit,

**double:** Alle darstellbaren Gleitkommazahlen mit doppelter Genauigkeit,

**array:** Zusammenfassung zusammengehöriger Daten gleichen Typs,

# Basis Datentypen

## Arrays

---

### Eindimensionaler Array:

Zusammenfassung eines Blocks von gleichartigen Daten.

Zu beachten ist, dass Arrays mit den Elementnummern 0 bis n-1 versehen sind!!



**Beispiel:** Array mit 9 string Elementen, string Buchname[9]

Buchname[0]	
Buchname[1]	Informatik
Buchname[2]	
Buchname[3]	
Buchname[4]	Algorithmen
Buchname[5]	
Buchname[6]	
Buchname[7]	Datenstrukturen
Buchname[8]	

Die Elemente eines Arrays liegen immer hintereinander im Speicher des Rechner (was sie von verketteten Listen unterscheidet, deren Speicherplatz dynamisch zugeteilt wird und durch Zeiger vom Programm verwaltet werden muss)

### Mehrdimensionaler Array:

Mehrdimensionale Arrays (in der Mathematik Matrizen genannt) verfügen über x Dimensionen (Vorsicht: Speicherplatzbedarf!!)



# Elementare Datenstrukturen: Verkettete Listen

## Einfach verkettete Listen

---

Nicht in jedem Fall ist bereits zum Zeitpunkt der Implementierung bekannt, aus wie vielen Elementen eine Liste bestehen wird (was zur Definition eines Arrays mit zusammenhängendem Speicherplatz nötig ist).

**Hier bieten sich verkettete Listen als Datenstruktur an:**

Mit jedem Listenelement wird ein Verweis auf den Speicherplatz des folgenden Elements gespeichert -> die Elemente können beliebig im Speicher verteilt sein, der Speicherplatz passt sich also dynamisch der jeweils aktuellen Größe der Liste an.

# Einfach verkettete Listen

---

**Implementierung:** Die lineare Liste wird als Folge von Knoten implementiert, die jeweils ein Listenelement und einen Zeiger auf das Folgeelement enthalten:

Eine Liste  $L = \langle a_1, \dots, a_n \rangle$  kann folgendermaßen dargestellt werden:

# Einfach verkettete Listen

## Implementierung von Listenanfang und -ende

---

Als nächster Schritt muss festgelegt werden, wie der Listenanfang und das Listenende markiert werden sollen.

Dies soll realisiert werden, indem jeweils ein Dummy Element als Zeiger auf den Anfang (head) und das Ende (tail) hinzugefügt wird:

# Einfach verkettete Listen

## Initialisierung der Liste

---

Um die Liste zu initialisieren, werden die Knoten für den Listenanfang und das Listenende eingeführt. Dabei soll der Zeiger Ende-Knoten immer auf das vorherige Element der Liste zeigen:

# Einfach verkettete Listen

## Suchen eines Elementes in der Liste

---

Zum Suchen eines Elementes schreibt man das zu suchende Element vor Beginn der Suche in das Dummy Element (tail) am Listenende.

Dann wird von Beginn der Liste Schritt für Schritt jedes Listenelement mit dem gesuchten x verglichen. Kommt x in der Liste vor wird ein Zeiger auf das erste vorkommende Element mit Inhalt x geliefert. Kommt x nicht vor, wird ein Zeiger auf das Dummy Element (tail) geliefert:

# Einfach verkettete Listen

## Einfügen eines Elementes an gegebener Position p (1)

Zum Einfügen eines neuen Elementes x an der Position p wird das bestehende Element an der Position p an einen neuen Speicherplatz  $p+1$  geschrieben und das Element p durch x ersetzt. Dadurch wird sicher gestellt, dass die jeweiligen Zeiger (insbesondere Element p-1) weiterhin korrekt sind:

Situation vor dem Einfügen von x:

Situation nach dem Einfügen von x:

# Einfach verkettete Listen

## Löschen eines Elementes an gegebener Position p (1)

---

Es soll das erste Element der Liste nach dem Dummy Element head gelöscht werden, das  $x$  enthält. Um zu verhindern, dass man durch das Löschen des letzten Elements in der Liste keinen Zugriff mehr auf das vorletzte Element hat (das ja jetzt auf das wiederum vorherige Element zeigen müsste), wird mit der Position pos auf das Element gezeigt, dessen next Komponente auf das zu löschende Element zeigt:

Situation vor dem Löschen von  $a_p = x$ :

Situation nach dem Löschen von  $a_p = x$ :

# Einfach verkettete Listen

## Löschen eines Elementes an gegebener Position p (2)

Umsetzung in Pseudo-Code:

```
procedure Entfernen (x:Grundtyp; x:Grundtyp; head; tail: Zeiger);  
var  
    pos: Zeigt ;  
begin  
    pos: Read;  
    tail^.dat = x { x in Dummy Element am Ende kopieren }  
    while pos^.next^.dat ≠ x do pos = pos^.next; { wandern durch Liste bis Element mit  
    Inhalt
```

# Doppelt verkettete Listen

---

Durch die Implementierung von doppelt verketteten Listen kann das Handling der Listen (natürlich auf Kosten des zusätzlichen Speicherbedarfs) vereinfacht werden.

Implementierung:

Die Zeiger in den Elementen der Liste können folgendermaßen dargestellt werden:

# Stack (Stapel)

## LIFO (last-in-first-out-Prinzip)

---

Für viele Anwendungen genügt es, wenn Operationen am Ende der Liste durchgeführt werden können. Der Stack ist ein klassisches Anwendungsbeispiel, bei dem typischerweise folgende Operationen definiert sind:

**push()**: Fügt ein Element am oberen Ende des Stack ab

**pop()**: Entfernt das oberste Element aus dem Stack

**top()**: liefert das oberste Element ohne es zu löschen

Wenn die maximale Größe des Stack im vorhinein bekannt ist, bietet sich eine Realisierung als Array an, ansonsten als verkettete Liste.

# Queue (Warteschlange)

## FIFO (first-in-first-out) Prinzip

---

Die klassischen Operationen einer Warteschlangen:

**put()**: Fügt ein Element am Ende der Queue hinzu

**get()**: Entnimmt ein Element am Anfang der Queue und liefert es

Auch bei der Queue bietet sich eine Array-Realisierung an, wenn die maximale Größe der Queue im Vorhinein bekannt ist. Für dynamische Queue-Größen ist eine verkettete Liste empfehlenswert.

# Kapitel 5: Datenstrukturen

## 5.2. Binäre Suchbäume

---



# Baumstrukturen

## Einige Definitionen (1)

---

Baumstrukturen sind eine der am häufigsten auftretenden Datenstrukturen in der Informatik (z.B. Suchbäume, Entscheidungsbäume, Kodebäume etc.).

Bäume haben im Vergleich zu Listen nicht nur einen Nachfolger, sondern zwei (Binärbaum) bis n. Die Elemente werden *Knoten* genannt und der erste Knoten als *Wurzel* bezeichnet. Die direkten Nachfolgeknoten sind *Söhne* und direkte Vorgängerknoten *Väter*. Jeder Knoten ist mit einem anderen Knoten nur über EINEN *Pfad* verbunden.

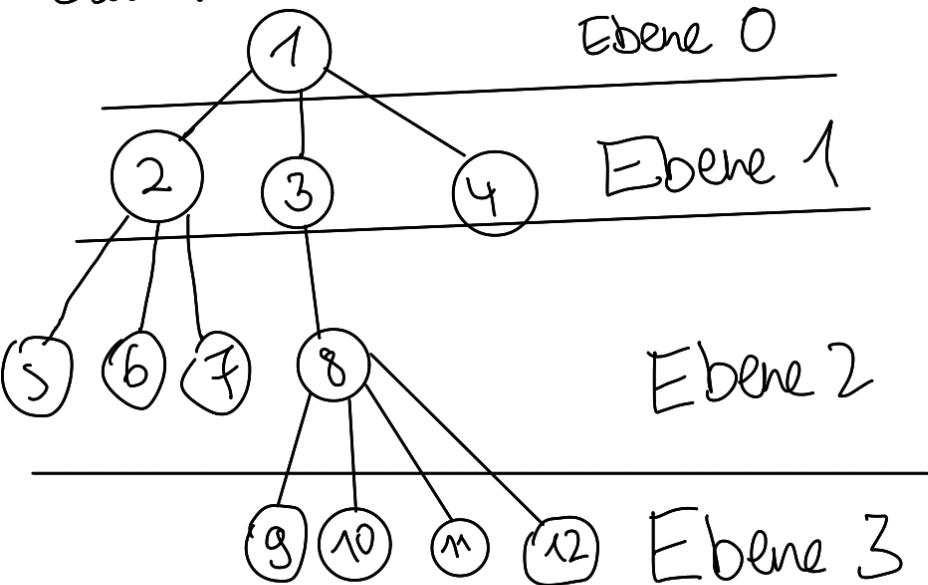
*Geordnete Bäume* zeichnen sich dadurch aus, dass alle Söhne eines Baumes einer gegebenen Struktur (z. B. 1., 2., 3. Sohn) folgen. Knoten ohne Söhne werden als *Blätter* bezeichnet.

Die *Ebene* eines Knotens ergibt sich aus der Anzahl der Knoten auf dem Pfad zur Wurzel (ohne dem genannten Knoten). Die *Höhe* des Baums entspricht seiner höchsten Ebenen Nummer und die *Pfadlänge* eines Baumes ist die Summe der Ebenen aller Knoten.

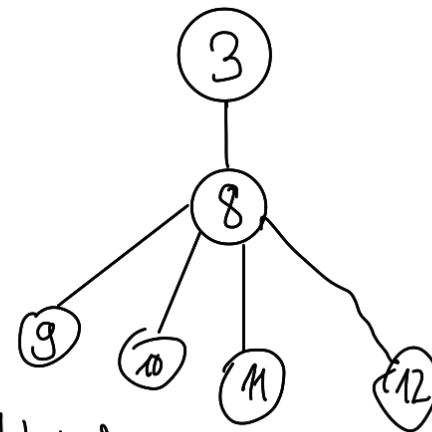
# Baumstrukturen

## Einige Definitionen (2)

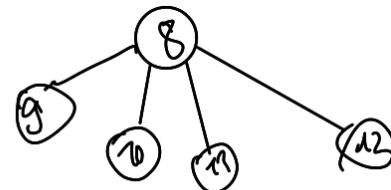
Baum :



Unterbau mit 6 Knoten



Unterbau mit 5 Knoten



2012

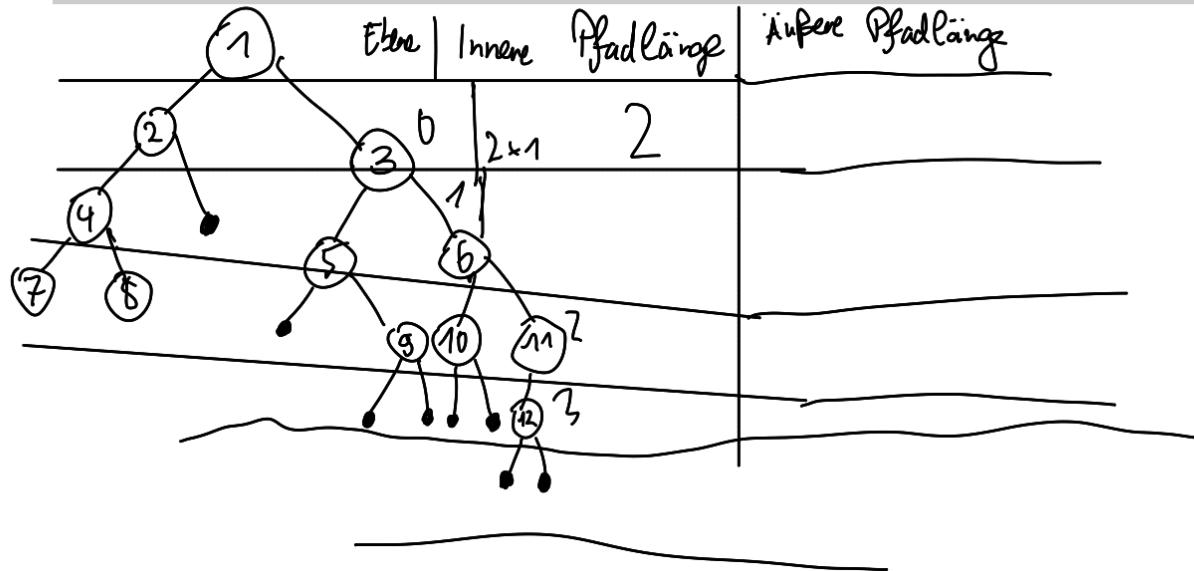
8 Unterbäume mit einem Knoten (Blätter)

- ⑤
- ⑥
- ⑦
- ⑨
- ⑩
- ⑪
- ⑫
- ⑭

Ein Wald mit 3 Bäumen

# Baumstrukturen

## Einige Definitionen (3)



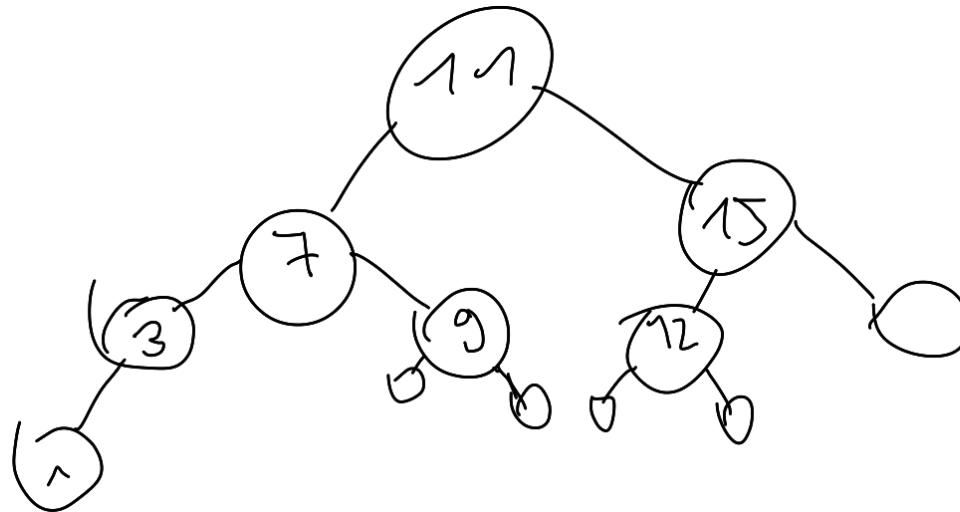
# Baumstrukturen

## Zeiger-Realisierung eines Suchbaumes

Bei Suchbäumen gilt für jeden Knoten p:

Sämtliche Schlüssel im linken Teilbaum von p sind kleiner als dessen Schlüssel, während sämtliche Schlüssel im rechten Teilbaum von p größer als dessen Schlüssel sind.

Beispiel:



# Baumstrukturen

## Zeiger-Realisierung eines Suchbaumes (1)

Grundsätzlich können Bäume programmtechnisch sowohl als Array als auch als Zeiger realisiert werden. Im folgenden soll die Realisierung durch Zeiger näher beleuchtet werden:

Bei der Umsetzung enthält jedes Element des Baumes den Schlüssel und jeweils einen Zeiger zum linken und einen Zeiger zum rechten Sohn. Damit kann jedes Element des Baumes folgendermaßen dargestellt werden:

Implementierung: *typ*

knotenzeiger =  $\uparrow$  knoten

knoten = record

leftson, rightson; knotenzeiger

key : Integer  
End

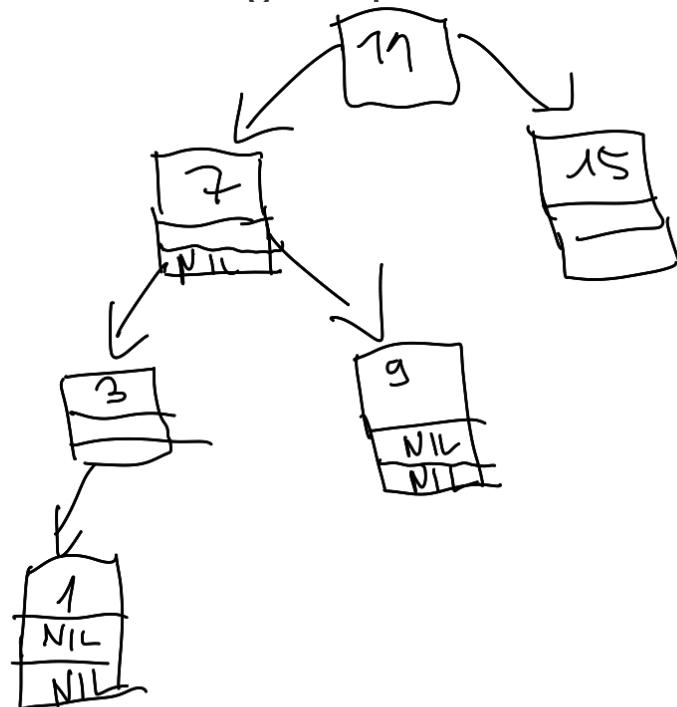


# Baumstrukturen

## Zeiger-Realisierung eines Suchbaumes (2)

Mit der gegebenen Definition ist ein Baum gegeben mit dem Zeiger auf die Wurzel:  
**var root: Knotenzeiger**

Die Blätter des Baumes beinhalten beim Suchbaum keine Schlüssel und können deshalb als nil-Zeiger repräsentiert werden:



# Baumstrukturen

## Implementierung zum Aufbau des Baumes

**program** Aufbau\_Baum (input, output);

**type**

Knotenzeiger=↑Knoten;

Knoten=**record**

    leftson, righson: Knotenzeiger;

    key: integer;

**end**;

**var**

wurzel: Knotenzeiger;

k:integer;

**Function** NeuerKnoten (var k:integer): Knotenzeiger

**begin**

    new(p); {neuen Knoten schaffen}

    p↑.leftson := nil;

    p↑.rightson := nil;

    p↑.key := k;

    NeuerKnoten := p;

**end**

**Procedure** Einfügen (**var** p:Knotenzeiger,k:integer);

**begin**

**if** k < p↑.key {linker Zweig}

**then begin**

**if** p↑.leftson = nil **then**

        p↑.leftson := NeuerKnoten(k);

**else** Einfügen(p↑.leftson, k);

**end**

**else if** k > p↑.key {rechter Zweig}

**then begin**

**if** p↑.rightson= nil **then**

        p↑.rightson := NeuerKnoten(k);

**else** Einfügen(p↑.righthson, k);

**else** write(`Schlusses existiert schon`);

**end;**

**begin**

    read(k);

    wurzel:= NeuerKnoten (k); {Wurzelknoten}

**while not** eof(input) **do** {bis keine Eingabe mehr}

**begin**

        read(k);

        Einfügen(wurzel,k)

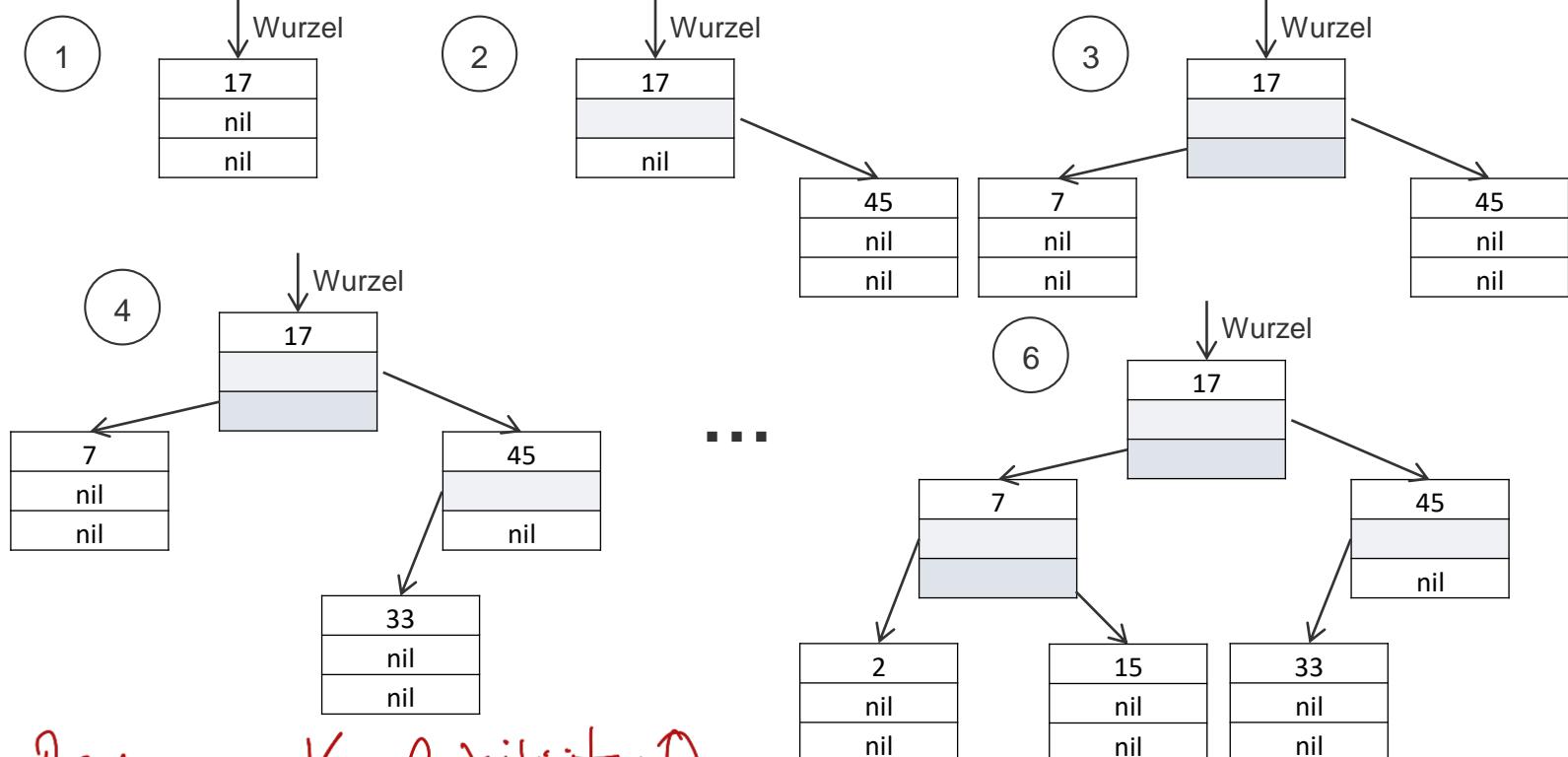
**end**

**end;**

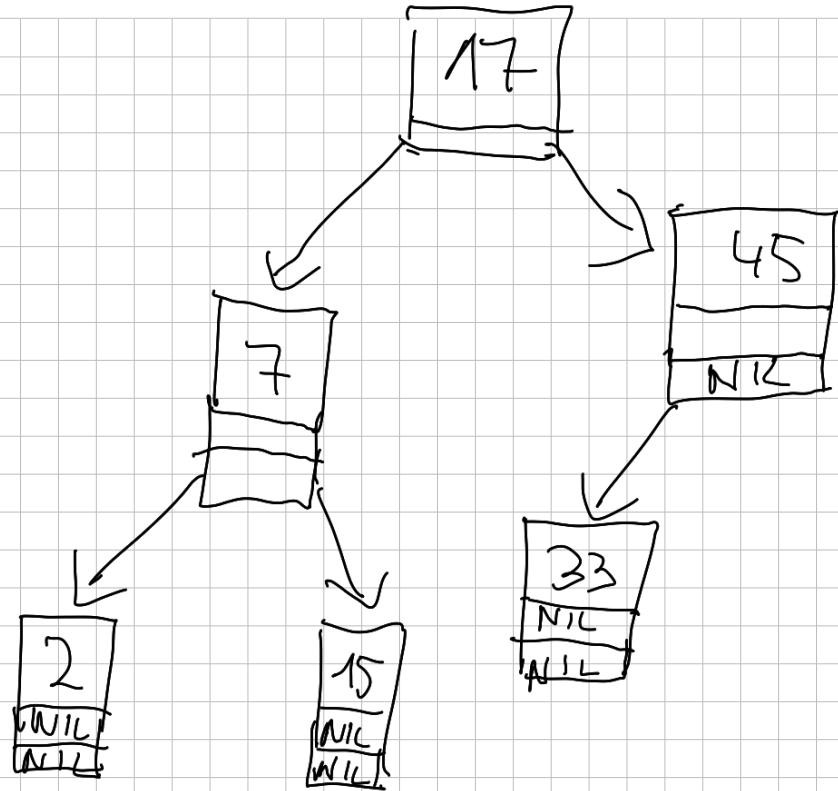
# Baumstrukturen

## Zahlenbeispiel: 17, 45, 7, 33, 2, 15

Je nachdem, welche Zahlen in welcher Reihenfolge eingegeben werden, können sehr unterschiedliche Baumstrukturen entstehen.



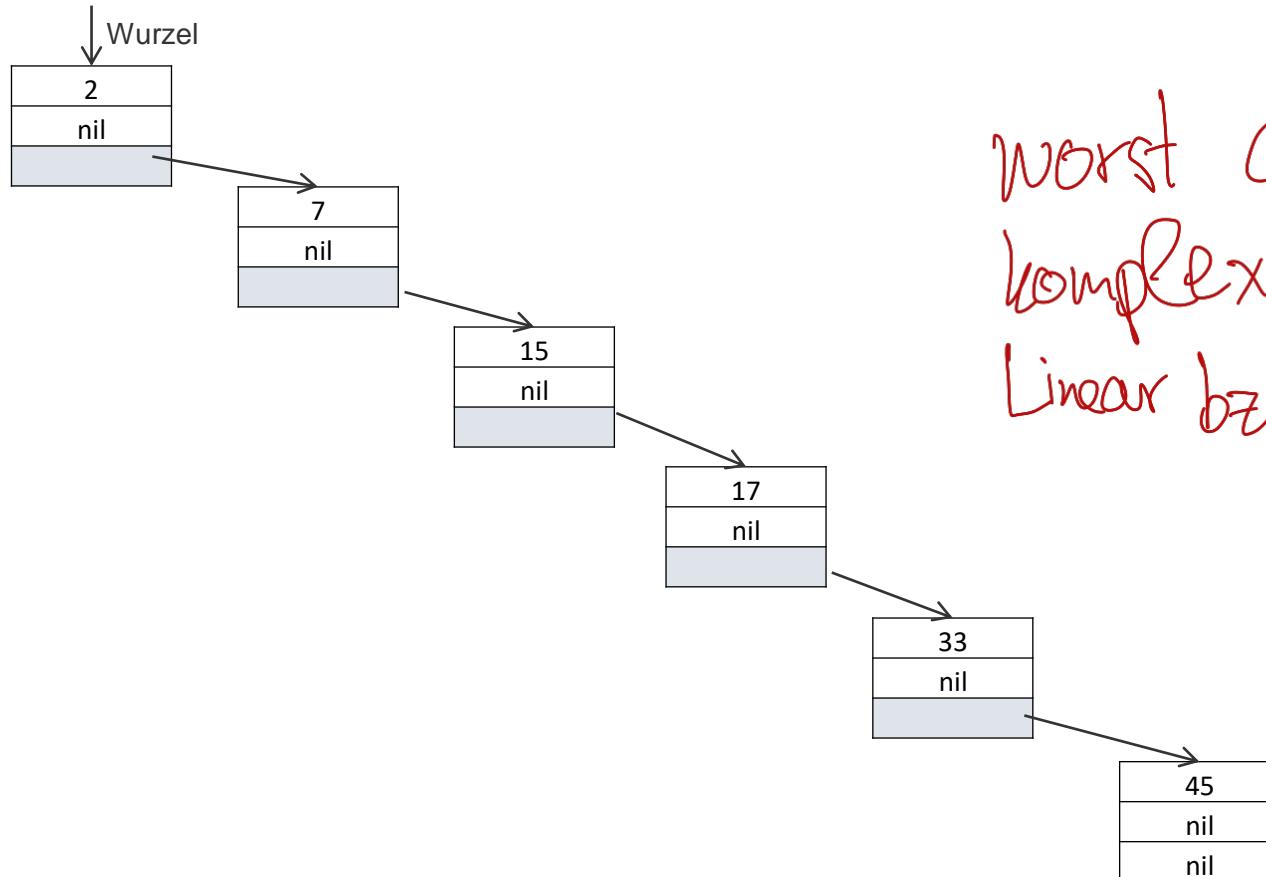
Best case - Komplexität:  $O(\log_2(n))$



# Baumstrukturen

## Zahlenbeispiel: 2, 7, 15, 17, 33, 45

Je nachdem, welche Zahlen in welcher Reihenfolge eingegeben werden, können sehr unterschiedliche Baumstrukturen entstehen.



# Baumstrukturen

## Suchen eines Schlüssels

Auch zum Suchen eines Schlüssels kann hier wieder ein rekursives Vorgehen gewählt werden:

Procedure Suchen ( $p$ : Knotenzeiger;  $X$ : Integer)

begin

if  $p = \text{nil}$

then write('Schlüssel',  $X$ , 'nicht im Baum vorhanden')

else

if  $X < p^1.\text{key}$

then Suchen ( $p^1.\text{leftson}$ ,  $X$ )

else

if  $X > p^1.\text{key}$

then Suchen ( $p^1.\text{rightson}$ ,  $X$ )

else write /'schlüssel (', x, ', im Baum vorhanden' );

end

# Baumstrukturen - Traversierungsreihenfolgen

## Unterschiedliche Untersuchungsreihenfolgen

---

1. **Pre-order** (depth first/Hauptreihenfolge/Tiefensuche) Traversierung  
Von der Wurzel über den linken zum rechten Teilbaum
  
2. **Post-order** (Nebenreihenfolge) Traversierung  
Vom linken über den rechten Teilbaum zur Wurzel
  
3. **In-order** (Symmetrische Reihenfolge) Traversierung  
Vom linken Teilbaum über die Wurzel zum rechten Teilbaum
  
4. **Reverse in-order** (anti-symmetrische Reihenfolge) Traversierung  
Vom rechten Teilbaum über die Wurzel zum linken Teilbaum
  
5. **Level-order** (breadth-first/Breitensuche)  
Von der Wurzel wird Ebene für Ebene durchlaufen

# Kapitel 5: Datenstrukturen

## 5.3. Balancierte Suchbäume

Shtëpia  
thëe

## 5.3. Balancierte Suchbäume

### Warum balancierte Suchbäume?

---

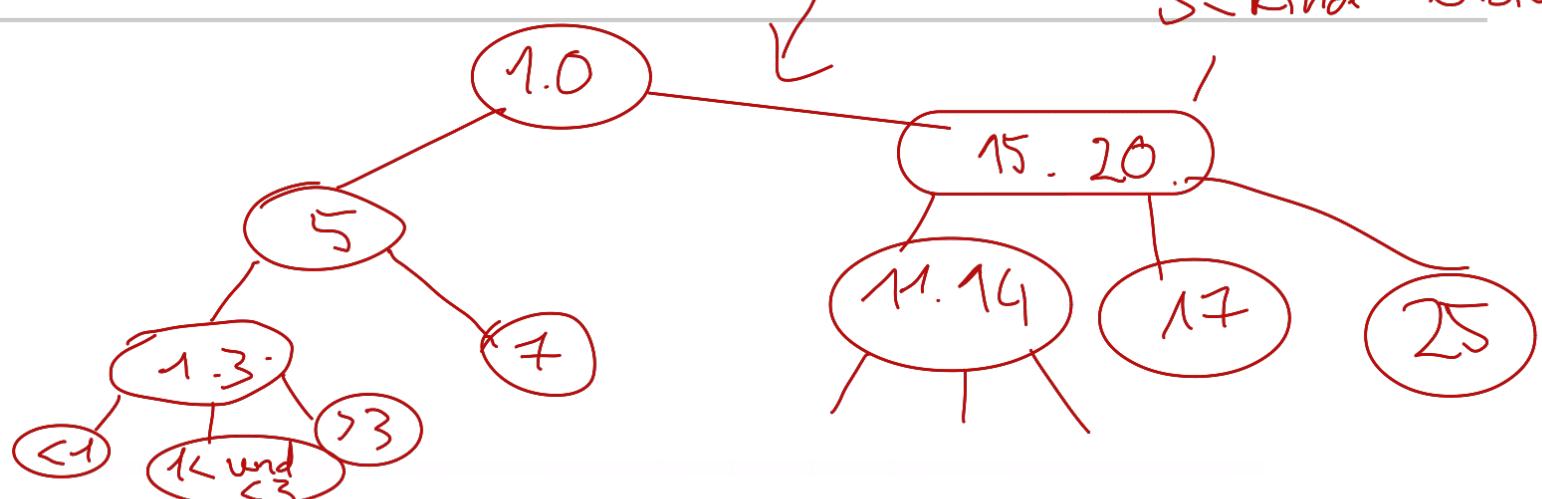
Je nach Eingabe können binäre Suchbäume ggf. zu sehr unsymmetrischen Strukturen führen (z. B. aufsteigende oder absteigende Eingabe), was zu einer schlechten Worst Case Performance führt.

Bei idealer Balance des Suchbaumes beträgt die Höhe des Baumes  $\log_2 N$  und es werden maximal  $\log_2 N$  Vergleiche zur Suche eines Elements benötigt.

Der im weiteren dargestellte Ansatz des 2-3 Suchbaumes stellt einen Kompromiss zwischen (aufwendig zu implementierender) perfekter Balance und garantiert logarithmischer Performance dar.

## 5.3. Balancierte Suchbäume

### Der 2-3-Suchbaum: Definition



Ein 2-3 Suchbaum ist entweder leer oder ein:

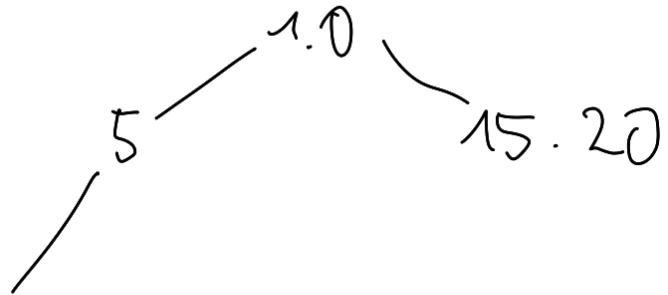
- 2-Kind-Knoten mit einem Schlüssel und einer linken Referenz zu einem 2-3 Suchbaum mit kleineren Schlüsseln und einer rechten Referenz zu einem 2-3 Suchbaum mit größeren Schlüsseln
- 3-Kind-Knoten mit 2 Schlüsseln und einer linken Referenz zu einem 2-3 Suchbaum mit kleinerem Schlüssel, einer mittleren Referenz zu einem 2-3 Suchbaum mit Schlüsseln zwischen den Schlüsseln des Knotens und einer rechten Referenz zu einem 2-3 Suchbaum mit größerem Schlüssel.



# 2-3 Suchbaum

## Suche eines Eintrages

Suche der Schlüssel 17 und 4 im 2-3 Suchbaum



# 2-3 Suchbaum

## Einfügen in 2-Kind-Knoten

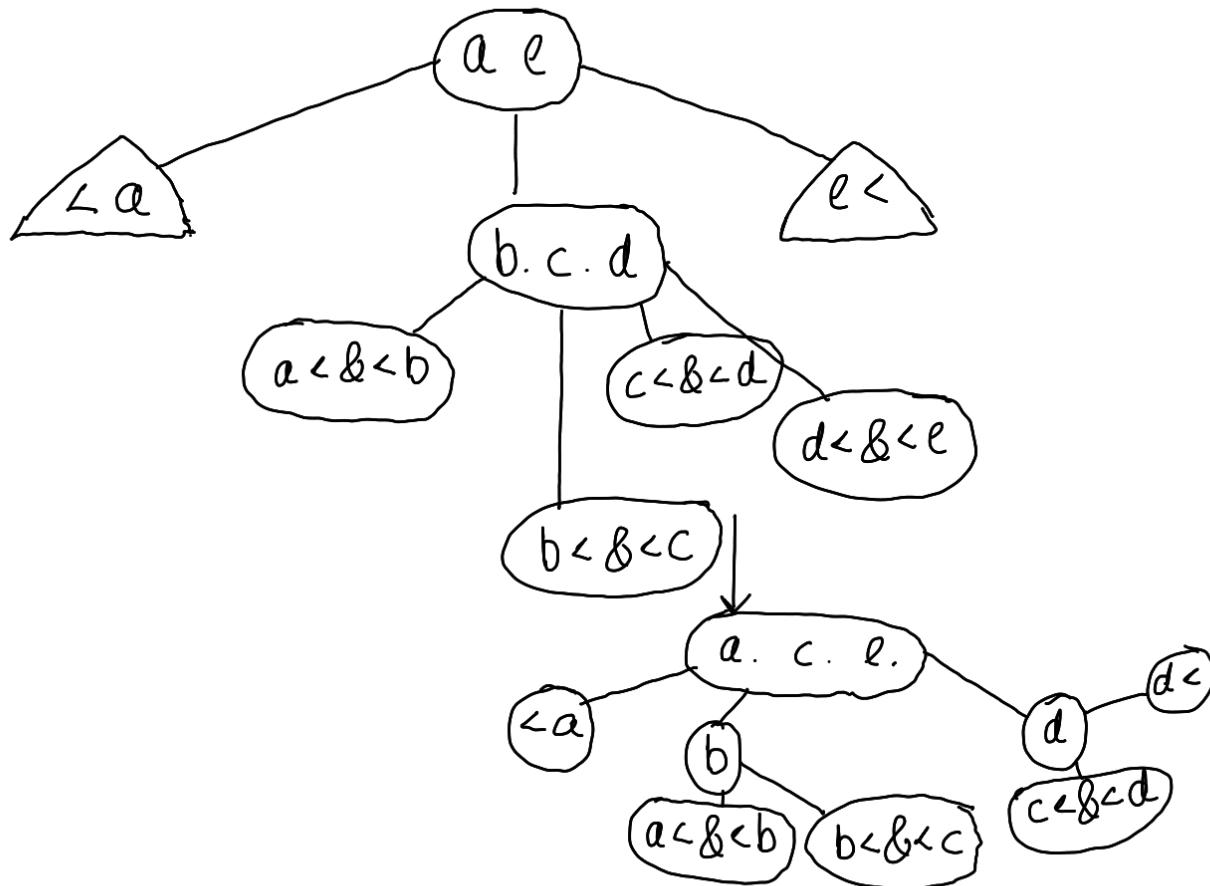
---

Einfügen des Schlüssels 18 im 2-3 Suchbaum

## 2-3 Suchbaum

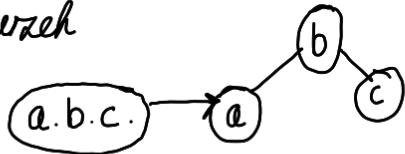
### Einfügen in 3-Kind-Knoten (I)

Das teilen eines 4er-Knoten ist eine lokale Operation:

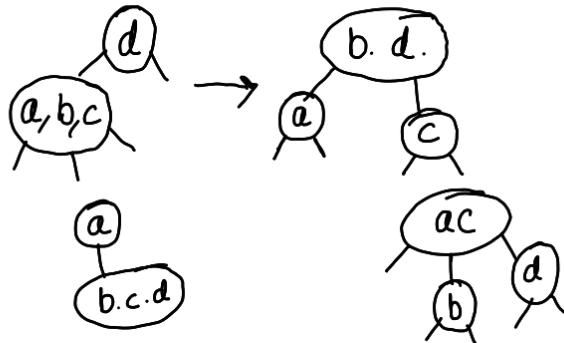


# 2-3 Suchbaum Einfügen in 3-Kind-Knoten (II)

Wurzel

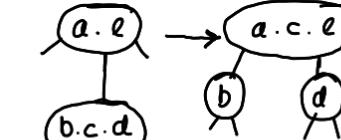
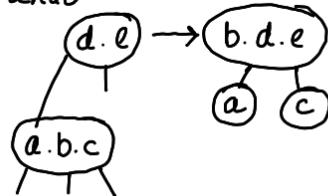


2. Kind - Knoten

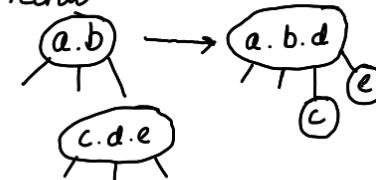


3. Kind - Knoten

links



rechts



## 5.3. Balancierte Suchbäume

### Gruppenarbeit zum Aufbau balancierter Suchbäume

#### Aufgabe:

Bauen Sie jeweils den 2-3 Suchbaum bei folgender Eingabe schrittweise auf:

1) 14, 3, 20, 18, 9, 17, 25, 5, 12, 1

2) 1, 3, 5, 9, 12, 14, 17, 18, 20, 25

Zeit: 10 Minuten

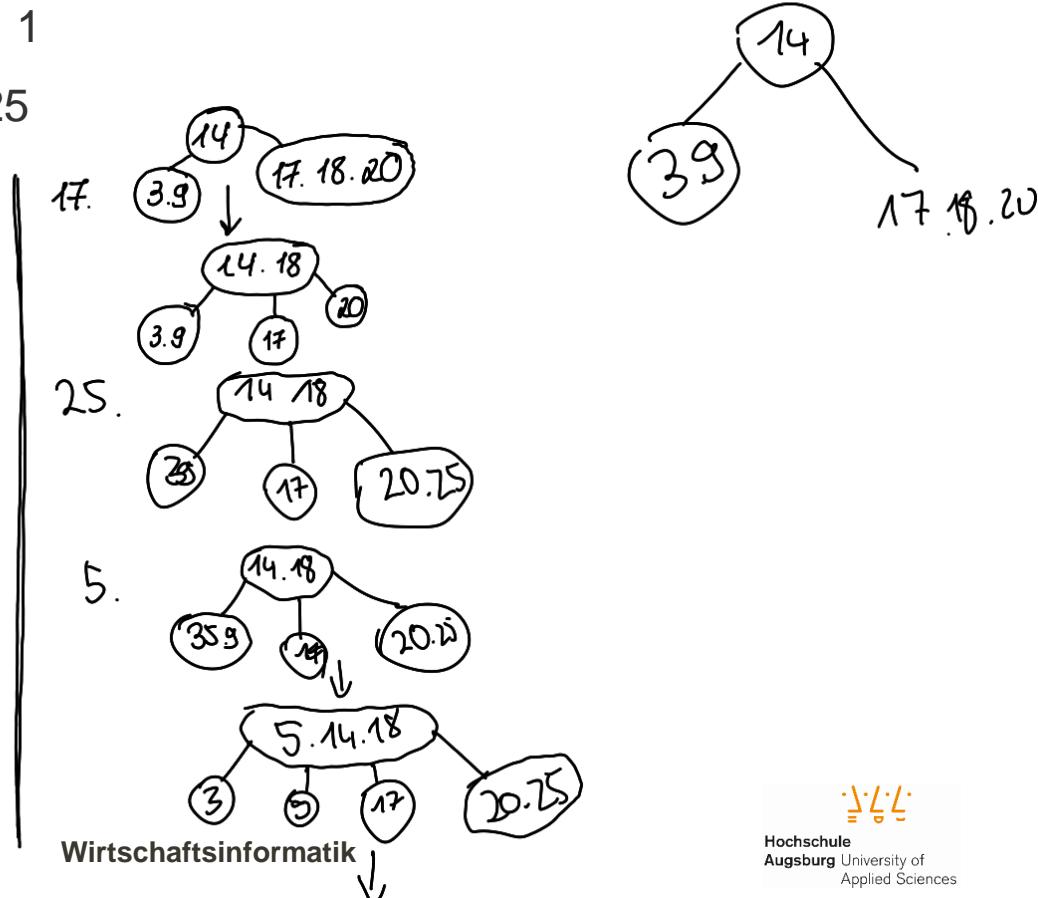
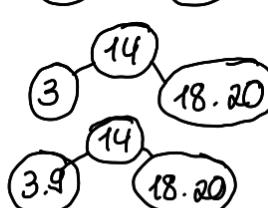
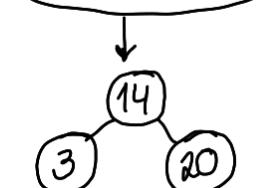
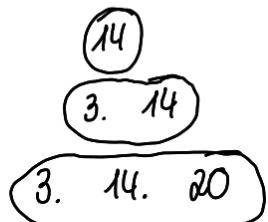
14 :

3 :

20 :

18:

9:



## 5.3. Balancierte Suchbäume

### Ergebnis Gruppenarbeit

---



# Kapitel 5: Datenstrukturen

## 5.4. Hashtabellen

---

## 5.4. Hashtabellen

### Idee der Hashtabelle

---

- Speichert man Elemente in einem Array oder in einer Liste, so muss man i.A. alle Positionen durchsuchen, um das geeignete Element zu finden
- Wenn man anhand des Elementes selbst auf die Position, an der das Element gespeichert werden soll, schließen kann, lässt sich das Verfahren optimieren
- Effizient wäre dies umzusetzen, wenn sich jedes Objekt eindeutig einem Index in einem Array zuordnen lassen würde
- Einfügen, löschen und suchen: Man muss nur an der entsprechenden Stelle nachsehen und das Element dort einfügen, löschen, oder zurückliefern (Suche)

## 5.4. Hashtabellen

### Warum Hashtabellen?

---

## Verfahren zum Speichern von Datensätzen

- Array  $\rightarrow$  Suchzeit  $\sim \Theta(n)$
- Verkettete Liste (einfach/doppelt)
  - $\rightarrow$  Suchzeit  $\sim \Theta(n)$
- Bäume (z.B. 2-3 Bäume)
  - $\rightarrow$  Suchzeit  $\sim \Theta(\log n)$
- Hashing ??
  - $\rightarrow$  Ziel:  $\Theta(1)$



int Hash = 0; M = 101, m = 31;

String s = "Augsburg"

for (int i=0; i < s.length(); i++)

Hash = (k + Hash + s.charAt(i) % m);

Beispiel: Schlüssel Hash

## 5.4. Hashtabellen

### Wie funktioniert Hashing?

- **Hashfunktion** berechnet den Arrayindex aus dem Suchschlüssel
- Dies kann zu Kollisionen führen, wenn unterschiedliche Suchschlüssel auf den identischen Arrayindex verweisen
- Mit der **Kollisionsauflösung** wird dieses Problem gelöst
- Somit wird mit Hashverfahren versucht, einen optimalen Kompromiss zwischen Rechenzeit und Speicherbedarf zu bilden

#	N A M E
0	Schirrmacher
1	Fredersdorf
2	Kuhls
3	Foege
4	Noll
5	Horn
6	Mietling
7	Bollmeyer
8	Behrens
9	
10	
11	
12	Tautz
13	Peters
14	
15	
16	Markiewicz
17	Stabler
18	Schneider
19	Ahrensl
20	Meier
21	Ahrens2
22	Celik
23	Bauschulte
24	Hartung
25	Wiedebusch
26	Kawuloc
27	Szyperrek
28	Juveczok
29	
30	Urbansky
31	Lammers
32	
33	Kessler
34	
35	



# Hashfunktionen

## Positive Integer: modulares Hashing

1. Im ersten Schritte gilt es festzulegen, wie groß die maximale Schlüssel-menge M ist, die in dem zum Speichern der Daten verwendeten Array abgelegt werden
2. Dividiert man nun den Schlüsselwert durch M-1 und nimmt den Rest der Division (Modulo-Operation), so ist dieser immer ein Wert zwischen 0 und M-1 und kann damit zur Adressierung des Arrays verwendet werden
3. Eine ideale Verteilung wird erreicht, wenn für M eine Primzahl verwendet wird. Ansonsten ist die Wahrscheinlichkeit groß, dass nicht alle Bits des Schlüssels berücksichtigt werden und damit keine optimal gleichmäßige Verteilung über das Array gegeben ist.

Schlüssel	M-1=100	M-1=101
685	85	79
921	21	12
455	55	51
379	79	76
772	72	65
845	45	37
680	80	74
229	29	27
302	2	100
322	22	19
169	69	68
280	80	78
263	63	61
877	77	69
372	72	69
536	36	31
366	66	63
280	80	78
202	2	0
444	44	40
108	8	7
115	15	14



# Hashfunktionen

## Strings: modulares Hashing auch möglich

---

- In diesem Fall wird der String wie ein riesiger Integer behandelt, indem die ASCII Werte der einzelnen Char Werte ausgelesen werden
- Um alle Buchstaben des Strings unabhängig von ihrer Stelle im String gleichwertig zu behandeln, werden sie mit einer Konstanten K multipliziert und char für char aufaddiert:
- Der folgende Code wendet dies beispielsweise für  $K = 47$  und eine Arraygröße  $M = 100$  an:
- Dabei ergeben sich beispielsweise folgende Hash-Werte:

# Hashing mit Verkettung

## Problemstellung

---

Das Hashverfahren teilt sich grundsätzlich in die zwei Schritte auf:

1. Hashfunktion zur Umwandlung von Schlüsseln in Array-Indizes
2. **Kollisionsauflösung** für den Fall, dass mehrere Schlüssel in den identischen Index aufgelöst werden:

Eine effektive und einfach zu implementierende Kollisionsauflösung ist das Hashing mit Verkettung.



# Hashing mit Verkettung

## Grundprinzip

Nutze ein Array mit  $M < N$   
verketteten Listen:

- Hash:

Umwandlung Schlüssel in Integer  
zwischen 0 und  $m - 1$

- Einfügen:

Füge Schlüssel an Anfang von  
inter Liste

- Suche:

# Hashing mit linearer Sondierung

## Grundprinzip

---

Bei den Verfahren des Hashing mit offener Adressierung wird beim Eintreten einer Kollision in dem Array nach dem nächsten freien Arrayindex gesucht. Bei der lineare Sondierung wird im Falle einer Kollision schrittweise der nächste Eintrag geprüft mit folgenden Alternativen:

1. Schlüssel = Suchschlüssel  $\rightarrow$  erfolgreiche Suche
2. Schlüssel  $\neq$  Suchschlüssel  $\rightarrow$  beim nächsten  
Eintrag weiter suchen

Die Implementierung erfolgt typischerweise mit parallelen Arrays für Schlüssel und Werte. Der Index zum Identifizieren der Daten im Arrays wird über den Index durch die Hashfunktion ermittelt.

Key	Hash	Value
S	2	0
E	0	1
A	0	2
R	4	3

# Hashing mit linearer Sondierung

## Konkretes Beispiel

Schlüssel	Hash	Wert	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S	6	0							5							.	.	
									0							.	.	
E	10	1											5		.	.	.	
												1			.	.	.	
A	4	2														2	.	
															3			
R	14	3																
C	5	4																
H	4	5								A								
E	10	6																
X	15	7																
A	4	8																
M	1	9																
P	14	10																
L	6	11																
E	10	12																



# Hashing mit linearer Sondierung

## Performance

Annahme

- „Perfekte“ Hash-Funktion
- m Tabelleneinträge
- n Anzahl der gespeicherten Einträge

Dann gilt:

- Anzahl Vergleiche S, um einen Eintrag zu finden

$$\text{Belegungsfaktor } S \approx \frac{1}{n} \cdot \ln\left(\frac{1}{1-\mu}\right) \quad \mu = \frac{n}{m}$$

⇒ um eine akzeptable Performance zu erreichen,  
sollte der Lastfaktor  $\mu$  nicht  $> 0,5$  sein  
⇒ Automatische Anpassung der Arrays wenn  
 $0,125 < \mu < 0,5$

# Hash-Verfahren

## Anmerkungen

---

Hashing ist ein in der Praxis häufig eingesetztes Verfahren, um auf Objekte schnell zugreifen zu können. Beispiele:

- Verwaltung von Objekten in OOP
- Verwendung von Punkten, Kreisen, Flächen
- Verwaltung von Punkten, Kreisen, Flächen in Algorithmen der Computergrafik

Eine Datenstruktur „Hash-Tabelle“ ist in vielen Programmiersprachen in Form von Bibliotheken verfügbar. Beispiele:

- java: `java.util.Hashtable`

Es gibt eine Vielzahl unterschiedlicher Hash-Verfahren und Hash-Funktionen (z. B. optimiert für bestimmte Anwendungsbereiche)

Eine aussagekräftige Komplexitätsanalyse ist oft schwierig, weil man die Häufigkeit von Kollisionen nur abschätzen kann, wenn man die Anwendung hinreichend gut kennt.

