LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR MATHEMATIK, INFORMATIK UND STATISTIK
INSTITUT FÜR INFORMATIK
LEHRSTUHL FÜR DATENBANKSYSTEME UND DATA MINING

Kapitel 4: Suchen

Einfache Suchverfahren:
Lineare Suche, Binäre Suche, Interpolationssuche
Suchbäume: Binäre Suchbäume, AVL- Bäume,
Splay-Bäume, B-Baum, R-Baum
Hashing

DBS Prof. Dr. Thomas Seidl

Motivation zum Suchen

- Anwendungsbeispiele:
  - Daten zu bestimmtem Schlüssel in einer Datenbank abfragen
  - Warensendung in einem Regallager finden
  - Eintrag in einem Wörterbuch

- In der Regel werden Duplikate ausgeschlossen, also jeder Schlüssel identifiziert maximal einen Datensatz

Dr. Marcus Gossler (https://commons.wikimedia.org/wiki/File:Latin_dictionary.jpg), „Latin dictionary“, https://creativecommons.org/licenses/by-sa/3.0/legalcode

Heinrich Taxis GmbH + Co. KG (https://commons.wikimedia.org/wiki/File:Hochregallager.jpg), https://creativecommons.org/licenses/by-sa/4.0/legalcode

Problemdefinition „Suchen“

- Universum: Die Menge, in der die Schlüsselwerte existieren.
  - Beispiel: European Article Number (EAN-13)

$$|U| = 10^{13}$$

- Eingabe:
  - Folge von *Datensätzen*

$$D_1, D_2, D_3, \dots, D_n = S \subset U$$

  - Jeder Datensatz besitzt eine Schlüsselkomponente $D_i.key$
  - Jeder Datensatz kann außerdem weitere Informationseinheiten enthalten (z.B. Name, Adresse, PLZ, etc.)

- Ausgabe:
  - Falls der Schlüssel nicht vorhanden ist, ist das Resultat leer
  - Sonst liefere Daten zum enthaltenen Schlüssel

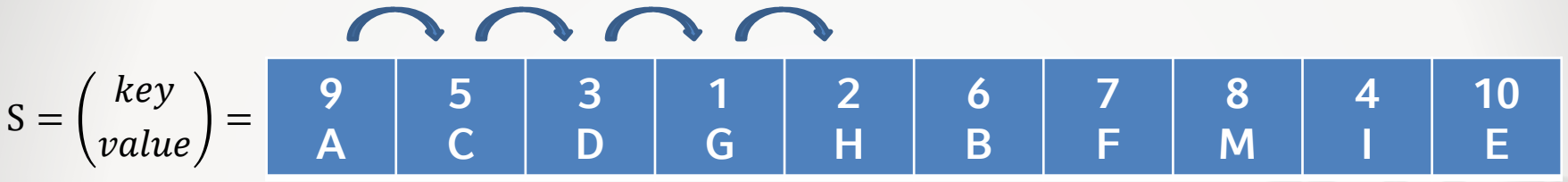
Suche nach nicht-vorhandenen Schlüsseln

- Im Allgemeinen dauert die Suche nach einem Element, welches nicht in S enthalten ist, länger als die Suche nach enthaltenen Elementen.
  - Erfolgreiche Suche: kann oft frühzeitig abgebrochen werden (wenn das Element gefunden wurde)
  - Nicht erfolgreiche Suche: bis zum *Ende* suchen, um sicherzustellen, dass der Schlüssel nicht in S enthalten ist.

Lineare Suche

- Lineare Suche durchläuft S sequenziell

linSearch(S,2)



- Wird auch als sequenzielle Suche bezeichnet
- S kann ungeordnet sein

```
S = [(1,"A"), (2,"C"), (3,"D"), (4,"G"), (5,"H"),
     (6,"B"), (7,"F"), (8, "M"), (9, "I"), (10, "E")]

def linSearch(S, searchkey):
  for key, value in S:
    if key == searchkey:
      return value
```

Lineare Suche: Komplexität

- Annahme: Die Wahrscheinlichkeit, dass der gesuchte Schlüssel an Position $1 \leq i \leq n$ ist, beträgt $\frac{1}{n}$ (jede Position ist gleich wahrscheinlich).

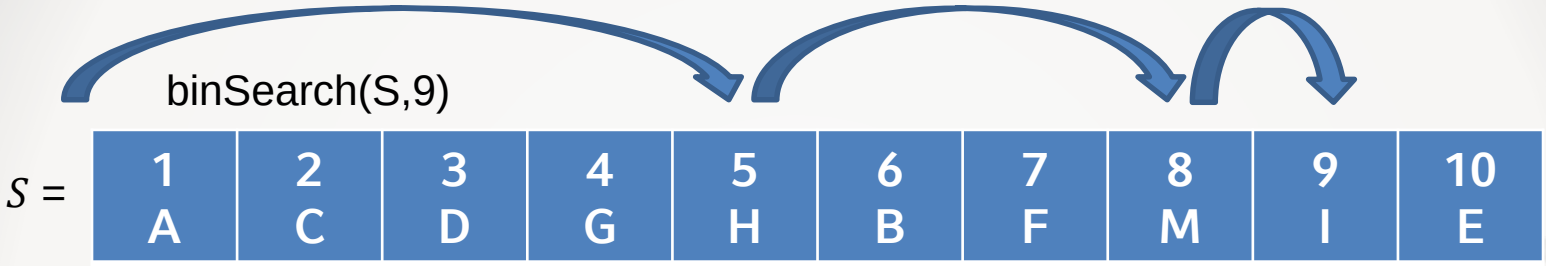
- Falls Schlüssel enthalten ist, werden im Mittel $\frac{n+1}{2}$ Vergleiche benötigt.

- Im schlimmsten Fall wird die gesamte Liste durchlaufen

$$\Rightarrow O(n)$$

Binäre Suche

- Voraussetzung: S ist sortiert



- Binäre Suche halbiert den Suchbereich sukzessive

```
def binSearch(S, searchkey):
  if len(S) > 0:
    mid = len(S)//2
    key, value = S[mid]
    if searchkey == key:
      return value
    elif searchkey < key:
      return binSearch(S[:mid], searchkey)
    else:
      return binSearch(S[mid+1:], searchkey)
```

Binäre Suche: Komplexität

- Falls Liste nicht vorsortiert ist, entsteht Zusatzaufwand $O(n \log n)$ durch Sortieren.
- Daher für allem geeignet bei
  - Vorsortierten Daten.
  - Daten, die selten verändert, aber auf denen häufig gesucht wird.

- Bei jedem Schleifendurchlauf halbiert sich der durchsuchte Bereich

$$T(n) = T\left(\frac{n}{2}\right) + 1$$

- Damit ist die Komplexität $O(\log n)$, auch im Worst-Case.

Anwendungsbeispiel: Suffix-Array

- Problem: Suche nach Teilzeichenketten P (Pattern) in einer Sequenz S (Text, DNA, Signal, …)
- Beispiel: Die Zeichenkette „münchen“ enthält unter anderem
  - „mü“, „ünch“, „chen“, …
- Die Suche nach dem Pattern „apfel“ wird erfolglos bleiben.

- Trivialer Ansatz (wie lineare Suche):
  - Durchlaufe S mit einem Suchfenster P der Größe $|P|$.
  - Teste für jede Position alle Fenstersymbole auf Gleichheit mit S.
  - $O(|P||S|)$, geht das besser?

SuffixArray: Aufbau

S:	M	I	S	S	I	S	S	I	P	P	I	$

Array mit Indexpositionen erstellen
Jede Position repräsentiert ein Suffix:

0	1	2	3	4	5	6	7	8	9	10	11
MISSISSIPPI$	ISSISSIPPI$	SSISSIPPI$	SISSIPPI$	ISSIPPI$	SSIPPI$	SIPPI$	IPPI$	PPI$	PI$	I$	$

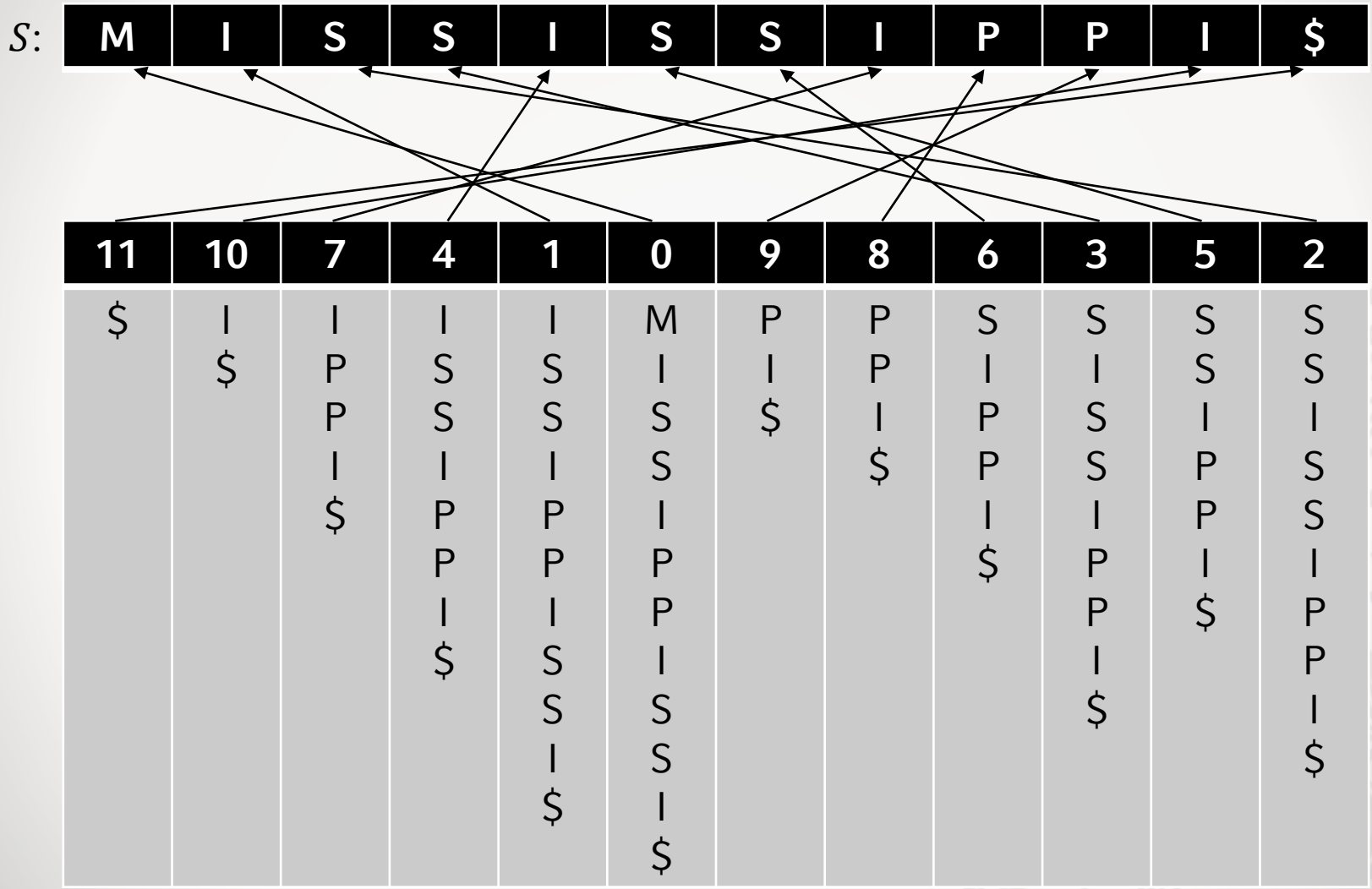
SuffixArray: Aufbau

S:	M	I	S	S	I	S	S	I	P	P	I	$

Generiertes Indexarray lexikographisch sortieren

11	10	7	4	1	0	9	8	6	3	5	2
$	I $	I P P I $	I S S I P P I $	I S S I P P I S S I $	M I S S I P P I S S I $	P I $	P P I $	S I P P I $	S I S S I P P I $	S S I P P I $	S S I S S I P P I $

SuffixArray: Speicherung mit Zeigern



SuffixArray: Suche

- Suche P = SSI

SSI > M...

11	10	7	4	1	0	9	8	6	3	5	2
$	I $	I P P I $	I S S I P P I $	I S S I P P I S S I $	M I S S I P P I S S I $	P I $	P P I $	S I P P I $	S I S S I P P I $	S S I P P I $	S S I S S I P P I $

SuffixArray: Suche

- Suche P = SSI

SSI > SI…

11	10	7	4	1	0	9	8	6	3	5	2
$	I $	I P P I $	I S S I P P I $	I S S I P P I S S I $	M I S S I P P I S S I $	P I $	P P I $	S I P P I $	S I S S I P P I $	S S I P P I $	S S I S S I P P I $

SuffixArray: Suche

- Suche P = SSI

SSI ~ SSI…

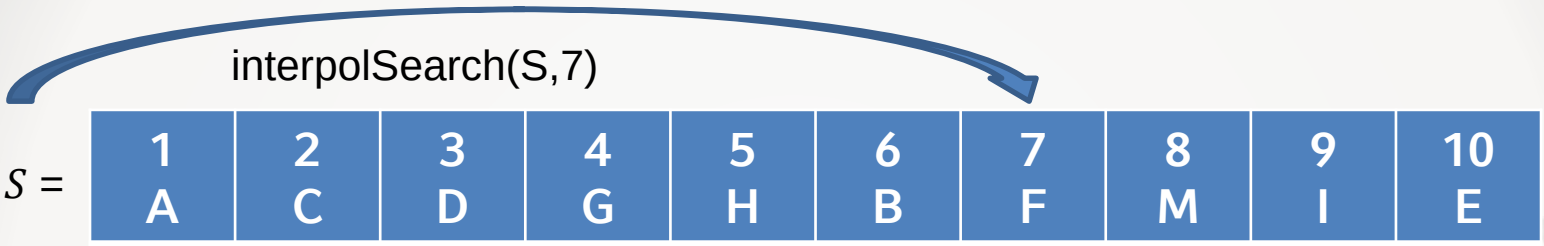
11	10	7	4	1	0	9	8	6	3	5	2
$	I $	I P P I $	I S S I P P I $	I S S I P P I S S I $	M I S S I P P I S S I $	P I $	P P I $	S I P P I $	S I S S I P P I $	S S I P P I $	S S I S S I P P I $

SuffixArray: Komplexität

- $O(\log|S|)$ Schritte für die Suche (binäre Suche)
- $O(|P|)$ Zeichenvergleiche in jedem dieser Schritte
- Insgesamt: $O(|P|\log|S|)$ Zeichenvergleiche insgesamt mit $O(|S|)$ Speicher.

Interpolationssuche

- Voraussetzung: S ist sortiert, Elemente sind gleichverteilt

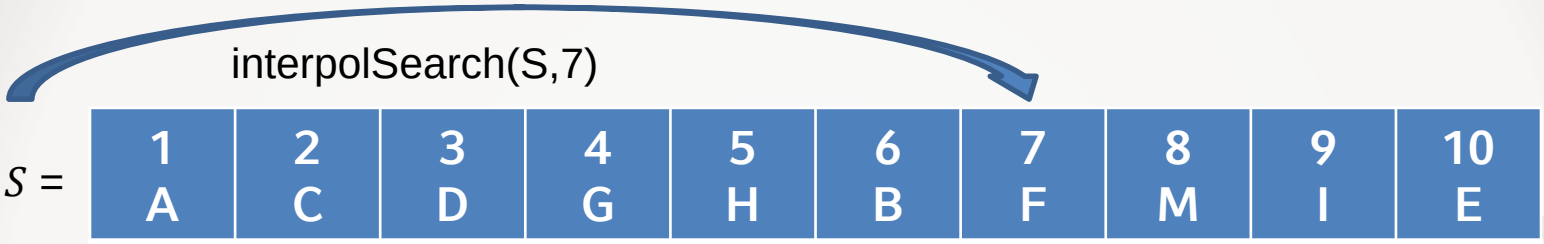


- Schätze gesuchte Position durch lineare Interpolation

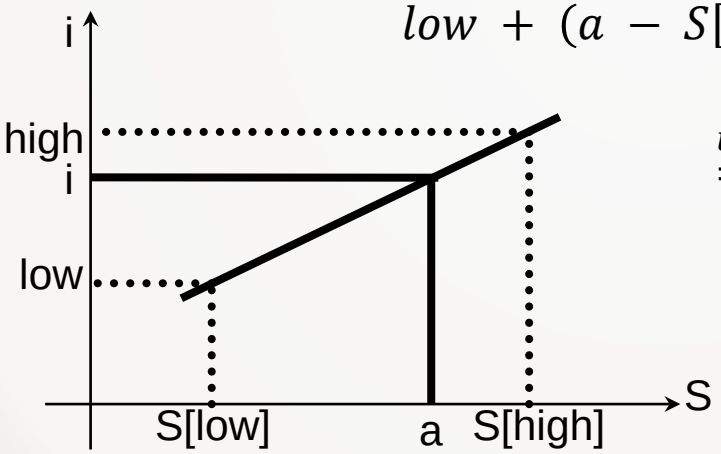
```
def interpolSearch(S, searchkey):
  if S[0][0] <= searchkey <= S[-1][0]:
    i = (searchkey-S[0][0])*(len(S)-1)//(S[-1][0]-S[0][0])
    key, value = S[i]
    if searchkey == key:
      return value
    elif searchkey < key:
      return interpolSearch(S[:i], searchkey)
    else:
      return interpolSearch(S[i+1:], searchkey)
```

Interpolationssuche

- Voraussetzung: S ist sortiert, Elemente sind gleichverteilt



- Schätze gesuchte Position durch lineare Interpolation

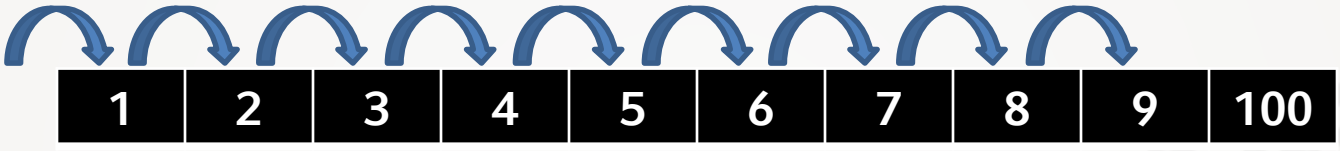


$$low + (a - S[low]) \frac{(high - low)}{(S[high] - S[low])}$$

$$\overset{if\ low=0}{\Longrightarrow} (a - S[0]) \frac{high}{(S[high] - S[0])}$$

Interpolationssuche: Komplexität

- Worst-Case (keine Gleichverteilung der Elemente):
  - $O(n)$, denn betrachte folgendes Beispiel mit interpolSearch(S,9):



- Average-Case Komplexität: $O(\log \log n)$ (ohne Beweis)

Zusammenfassung: Einfache Suche

Methode	Lineare Suche	Binäre Suche	Interpolationssuche
Suche (Avg.)	$\frac{n+1}{2} \in O(n)$	$\lfloor \log_2 n + 1 \rfloor \in O(\log n)$	$O(\log \log n)$
Suche (Worst)	$n \in O(n)$	$\lfloor \log_2 n + 1 \rfloor \in O(\log n)$	$O(n)$
Speicher	$n \in O(n)$	$n \in O(n)$	$O(n)$
Vorteil	Keine Initialisierung	Worst-Case auch $O(\log n)$	Schnelle Suche
Nachteil	Hohe Suchkosten	Sortiertes Array	Sortiertes Array Starke Bedingung an Datenverteilung

Suche in Bäumen

- Bisher Suche in linearen Strukturen (Mengen, Listen)
- Sortierte Arrays nur sinnvoll für statische Mengen, da Einfügen und Entfernen in $O(n)$ liegen.

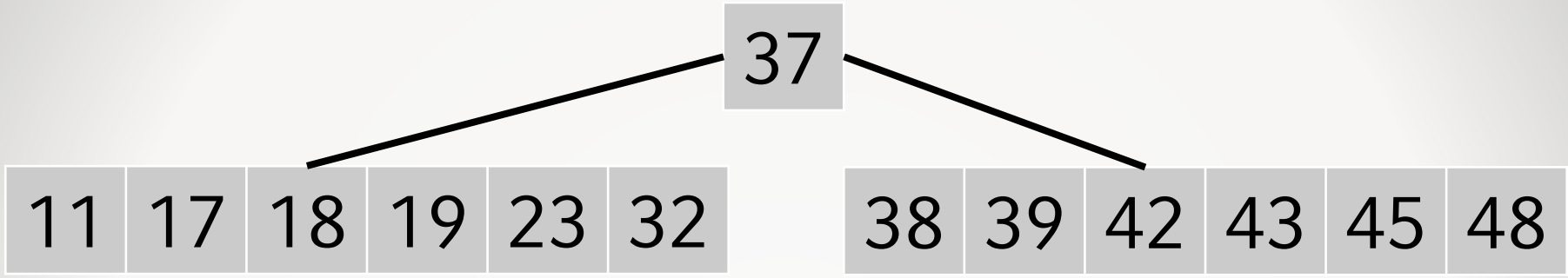
- Nun betrachten wir Bäume, um Daten strukturiert zu speichern.
- Einfügen, Löschen und Suchen von Elementen ist komplexer.
- Effiziente Lösungen für die Verwendung eines Sekundärspeichers.

Binäre Suchbäume

11	17	18	19	23	32	37	38	39	42	43	45	48

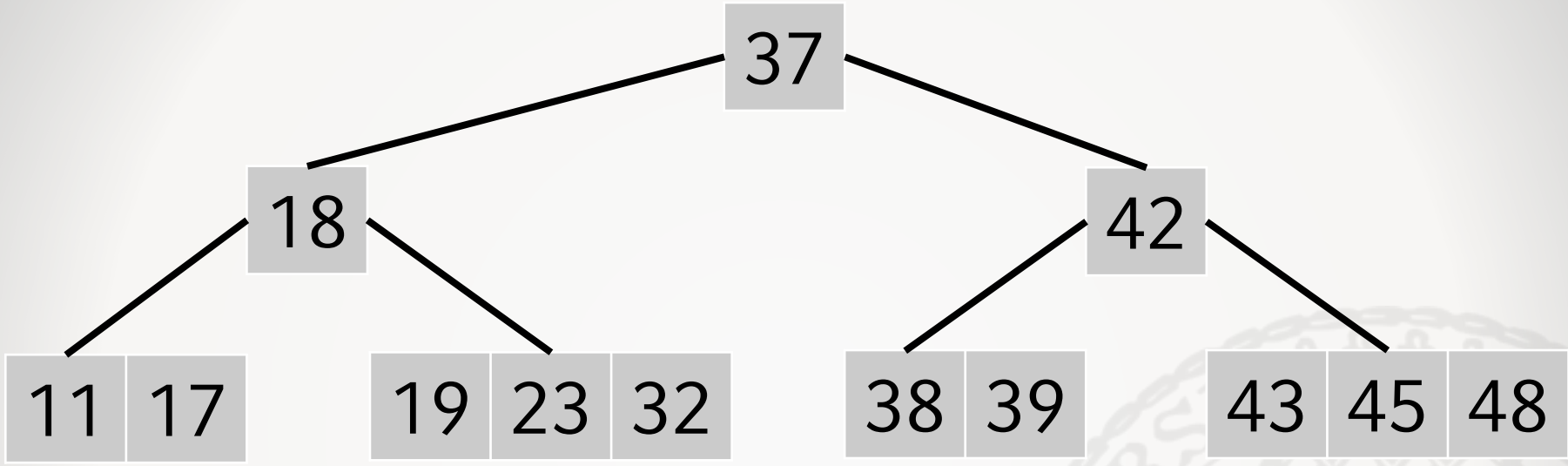
- Start bei der Mitte -> Wurzel
- Aufteilen in linken und rechten Teil (ohne Mitte)

Binäre Suchbäume



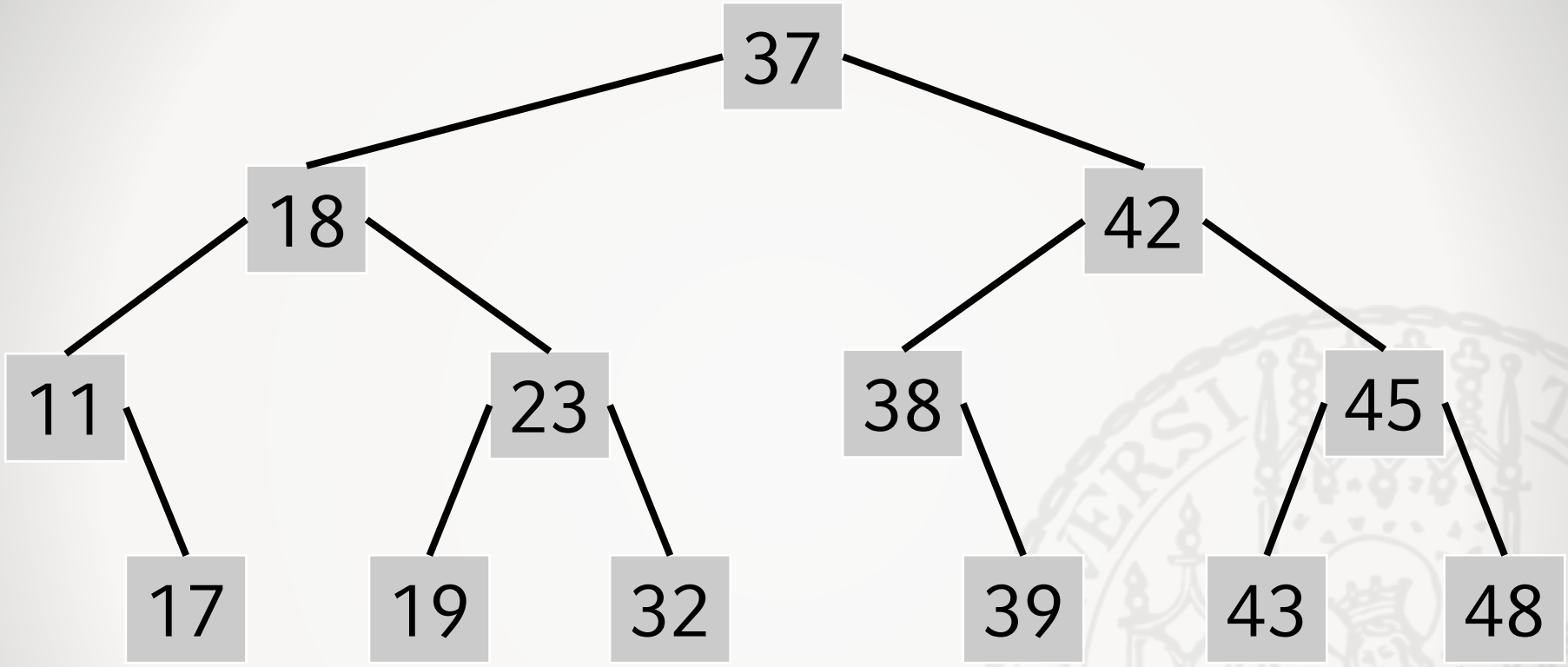
- Start bei der Mitte -> Wurzel
- Aufteilen in linken und rechten Teil (ohne Mitte)

Binäre Suchbäume



- Start bei der Mitte -> Wurzel
- Aufteilen in linken und rechten Teil (ohne Mitte)

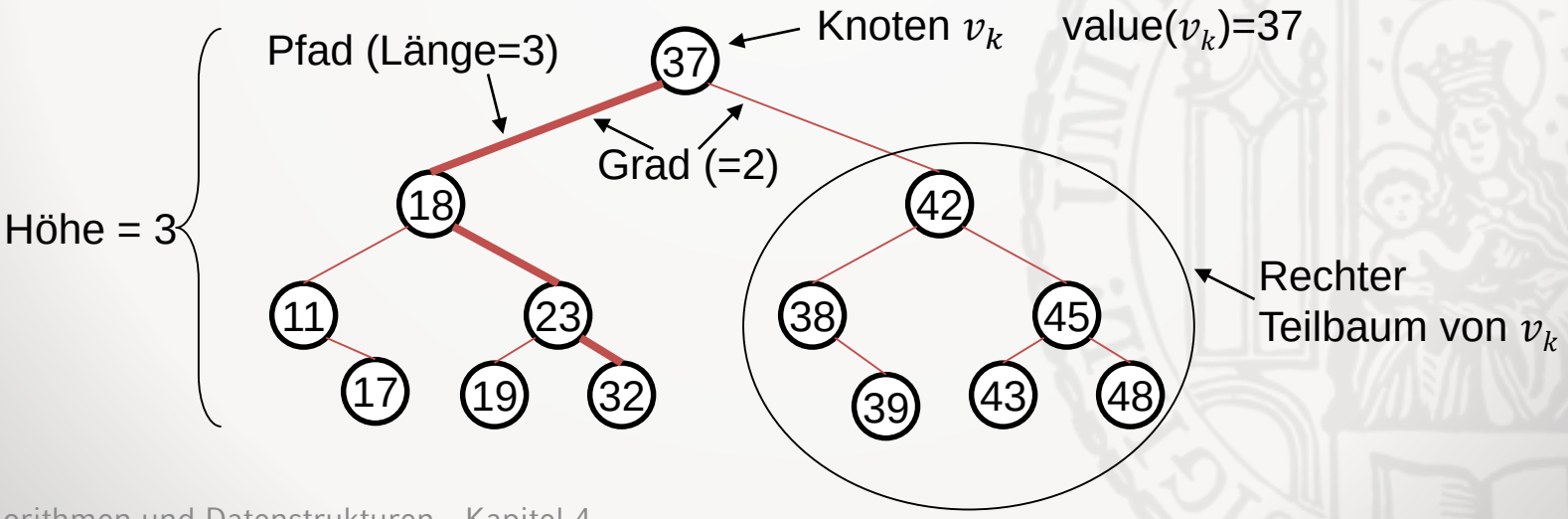
Binäre Suchbäume



- Start bei der Mitte -> Wurzel
- Aufteilen in linken und rechten Teil (ohne Mitte)

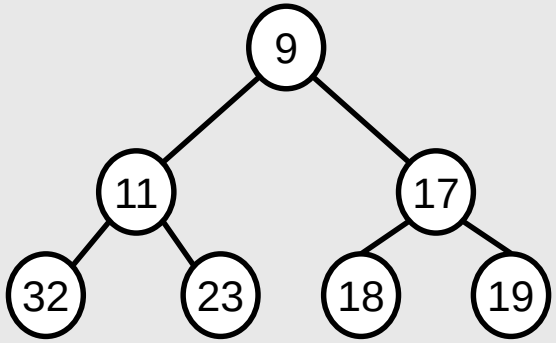
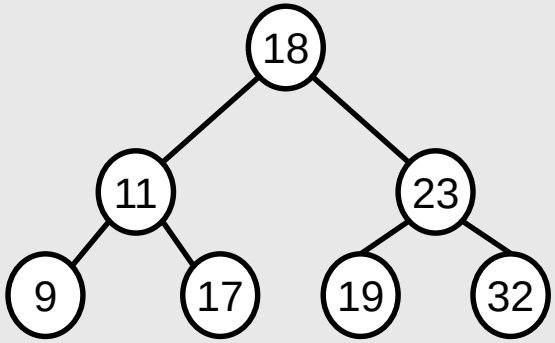
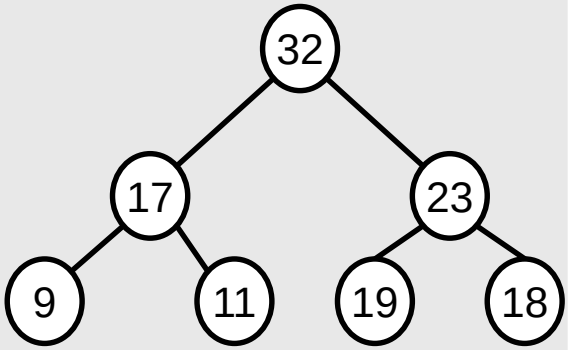
Binäre Suchbäume: Definition

- Ein binärer Suchbaum für eine Menge von Schlüsseln
$$S = \{x_1, x_2, \ldots, x_n\}$$
besteht aus einer Menge beschrifteter Knoten
$$v = \{v_1, v_2, \ldots, v_n\}$$
mit Beschriftungsfunktion $value: v \rightarrow S$
- Die Beschriftungsfunktion bewahrt die Ordnung in der Form: Wenn v_i im linken Teilbaum von v_k liegt und v_j im rechten Teilbaum dann $value(v_i) \leq value(v_k) \leq value(v_j)$



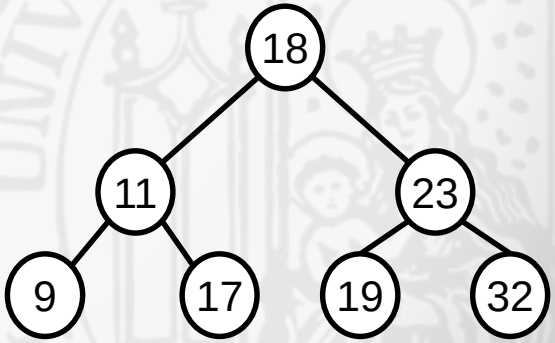
Binärer Suchbaum vs. Heap

- Ein binärer Suchbaum und ein Heap unterscheiden sich durch ihre strukturellen Invarianten:
  - Wenn v_i im linken Teilbaum von v_k liegt und v_j im rechten Teilbaum, dann gilt:

Min-Heap	Binärer Suchbaum	Max-Heap
$value(v_k) \leq value(v_i)$ $value(v_k) \leq value(v_j)$	$value(v_i) \leq value(v_k)$ $value(v_k) \leq value(v_j)$	$value(v_i) \leq value(v_k)$ $value(v_j) \leq value(v_k)$
		

Binärer Suchbaum: Suche

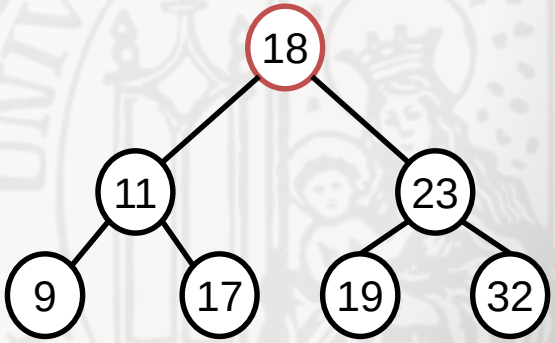
- Idee: Rekursive Erkundung eines Pfades
  - Suche nach Schlüssel s beginnt an der Wurzel $v = root$
  - Falls der aktuelle Knoten v Schlüssel s enthält → s gefunden!
  - Falls nicht:
    - v ist Blatt → s nicht enthalten!
    - Schlüssel ist kleiner als $value(v)$ → Suche im linken Teilbaum
    - Schlüssel ist größer als $value(v)$ → Suche im rechten Teilbaum



Binärer Suchbaum: Suche

- Idee: Rekursive Erkundung eines Pfades
  - Suche nach Schlüssel s beginnt an der Wurzel $v = root$
  - Falls der aktuelle Knoten v Schlüssel s enthält → s gefunden!
  - Falls nicht:
    - v ist Blatt → s nicht enthalten!
    - Schlüssel ist kleiner als $value(v)$ → Suche im linken Teilbaum
    - Schlüssel ist größer als $value(v)$ → Suche im rechten Teilbaum

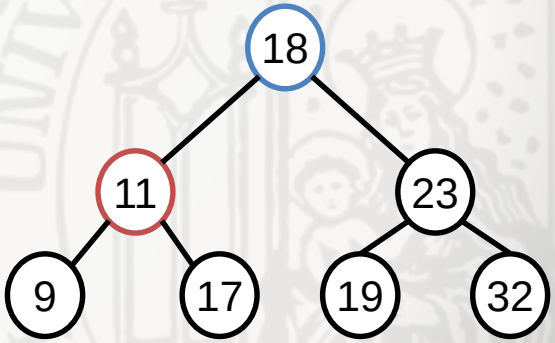
- Beispiel: Suche nach 17



Binärer Suchbaum: Suche

- Idee: Rekursive Erkundung eines Pfades
  - Suche nach Schlüssel s beginnt an der Wurzel $v = root$
  - Falls der aktuelle Knoten v Schlüssel s enthält → s gefunden!
  - Falls nicht:
    - v ist Blatt → s nicht enthalten!
    - Schlüssel ist kleiner als $value(v)$ → Suche im linken Teilbaum
    - Schlüssel ist größer als $value(v)$ → Suche im rechten Teilbaum

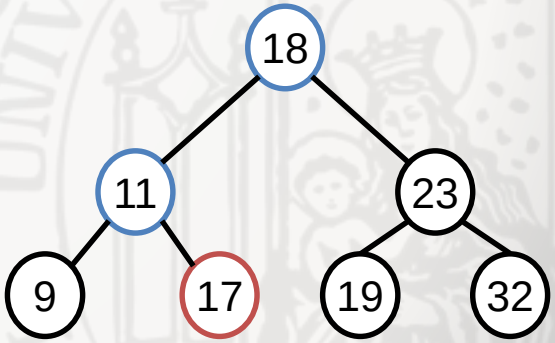
- Beispiel: Suche nach 17



Binärer Suchbaum: Suche

- Idee: Rekursive Erkundung eines Pfades
  - Suche nach Schlüssel s beginnt an der Wurzel $v = root$
  - Falls der aktuelle Knoten v Schlüssel s enthält → s gefunden!
  - Falls nicht:
    - v ist Blatt → s nicht enthalten!
    - Schlüssel ist kleiner als $value(v)$ → Suche im linken Teilbaum
    - Schlüssel ist größer als $value(v)$ → Suche im rechten Teilbaum

- Beispiel: Suche nach 17



Binäre Suchbäume: Implementierung Knoten

- Basierend auf Binärbäumen
- Schlüsselwert muss vergleichbar sein
- Jeder Knoten ist ein Binärbaum mit evtl. Subbäumen.

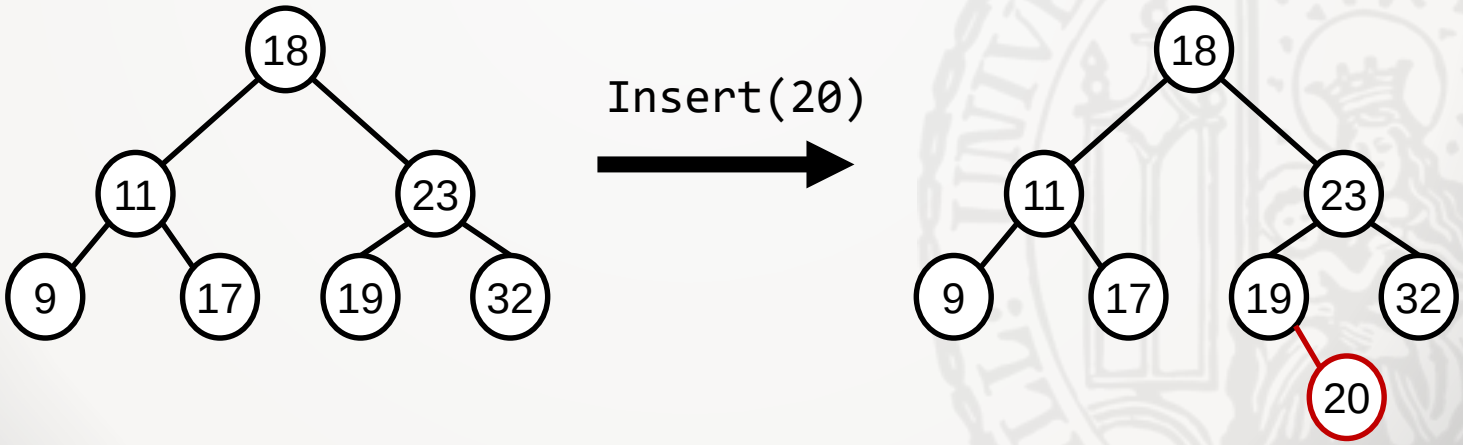
```
class node():
    def __init__(self, key = None, value = None):
        self.key = key
        self.value = value
        self.left = None
        self.right = None
```

Binäre Suchbäume: Suche nach Schlüssel

```
def search(self, key):
  if key == self.key:
    return self.value
  elif key < self.key and self.left != None:
    return self.left.search(key)
  elif key > self.key and self.right != None:
    return self.right.search(key)
```

Binäre Suchbäume: Einfügen Schlüssel+Objekt

- Intuition für *insert(key, value)*
  - Suche den Schlüssel *key* im Baum.
  - Falls *key* schon im Baum existiert, ersetze das vorherige Objekt.
  - Falls nicht, hält die Suche in einem Blatt oder innerem Knoten mit max. einem Nachfolger.
  - Füge einen neuen Knoten mit *(key, value)* als linker oder rechter Folgeknoten bei diesem Knoten ein.

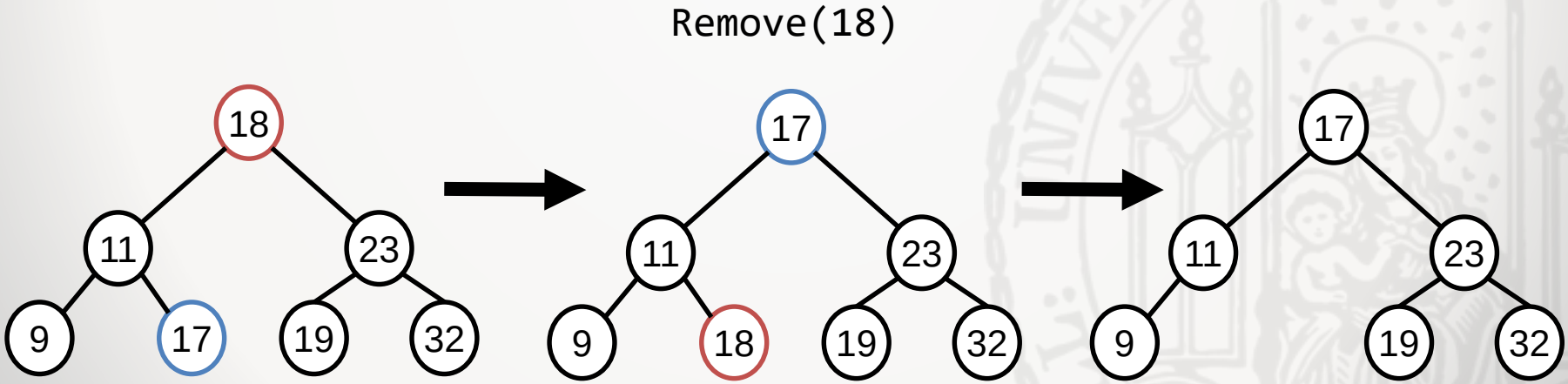


Binäre Suchbäume: Einfügen Schlüssel+Objekt

```
def insert(self, key, value):
  if self.key == None:
    self.key, self.value = key, value
  elif key == self.key:
    self.value = value
  elif key < self.key:
    if self.left == None:
      self.left = Node(key, value)
    else:
      self.left.insert(key, value)
  else:
    if self.right == None:
      self.right = Node(key, value)
    else:
      self.right.insert(key, value)
```

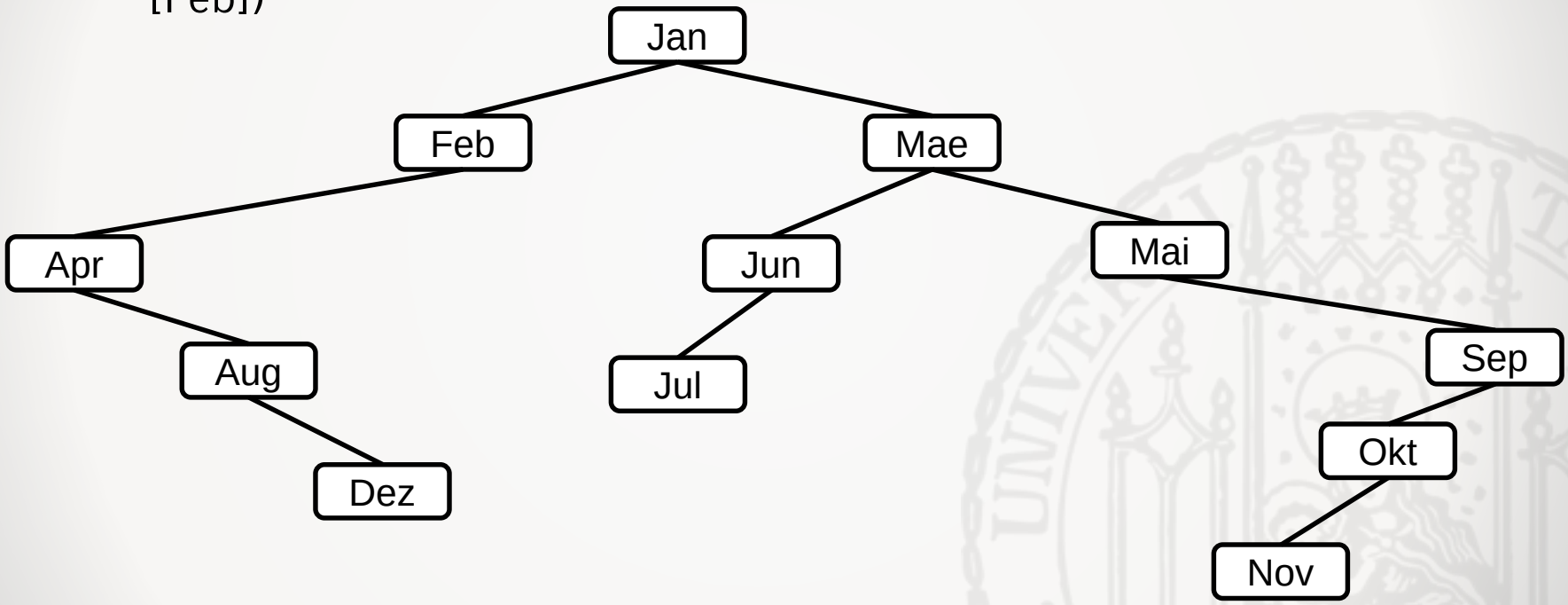
Binäre Suchbäume: Löschen

- Intuition für *remove(key)*
  - Suche den Knoten v mit Schlüssel *key* im Baum.
  - Falls *key* nicht im Baum existiert, passiert nichts.
  - Falls v ein Blatt ist, lösche den Zeiger darauf.
  - Falls v ein innerer Knoten ist, finde rechtesten Knoten v' (größten Schlüssel) im linken Teilbaum von v. Tausche v mit v'. Lösche dann den Blattknoten v.



Suchbäume für lexikografische Schlüssel

- Beispiel: Deutsche Monatsnamen
  - Sortierung lexikographisch
  - Einfügen in kalendarischer Reihenfolge (nicht mehr ausbalanciert [Feb])



  - Ausgabe durch InOrder-Traversierung (siehe Kap. 1):
  - Apr - Aug - Dez - Feb - Jan - Jul - Jun - Mae - Mai - Nov - Okt - Sep

Binäre Suchbäume: Komplexität

- Analyse der Laufzeit Insert und Remove
  - Suchen der entsprechenden Position im Baum.
  - Lokale Änderungen im Baum in O(1).

- Analyse des Suchverfahrens
  - Anzahl Vergleiche entspricht maximale Pfadtiefe des Baumes
  - Sei $h(t)$ die Höhe des Baumes t, dann ist die Komplexität der Suche $O(h(t))$.
  - Wir wissen: Hat t genau n Knoten, dann gilt:
    $$h + 1 \leq n \leq 2^{h+1} - 1$$
  - Damit gilt im Worst-Case Komplexität $O(n)$ und im Best-Case $O(\log n)$.

Binäre Suchbäume: Fazit

- Operationen *insert()*, *get()* und *remove()* haben im optimalen Fall eine gute Komplexität $O(\log n)$.
- Die Operationen *insert()* und *remove()* können den Baum aber entarten lassen zu einer linearen Liste.

- Idee:
  Modifiziere *insert()* und *remove()*, sodass die Teilbäume jedes Knotens ungefähr gleich groß bleiben.
- Diese Eigenschaft nennt man balanciert.

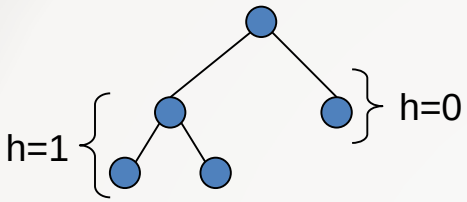
AVL-Bäume

- Historisch erste Variante eines balancierten Baums.
- Name basiert auf den Erfindern: Adelson-Velsky & Landis.

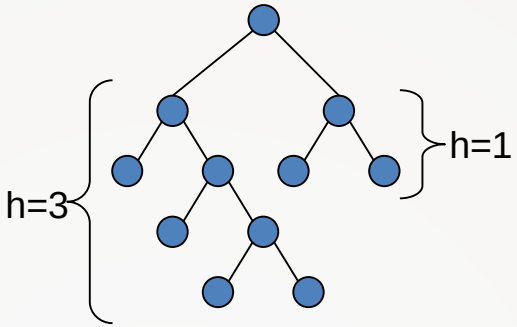
- Definition: Ein AVL-Baum ist ein binärer Suchbaum mit folgender Strukturbedingung:
  Für alle Knoten gilt, dass die Höhen der beiden Teilbäume sich höchstens um eins unterscheiden.

- Die Suche funktioniert exakt so wie bei binären Suchbäumen.
- Nur nach *insert* und *remove* muss eventuell rebalanciert werden.

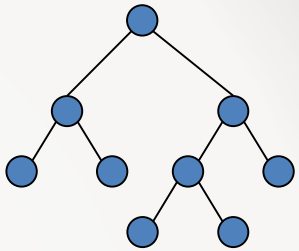
AVL-Bäume: Beispiele



AVL-Baum $|\Delta h| \leq 1$



kein AVL Baum $|\Delta h| = 2$

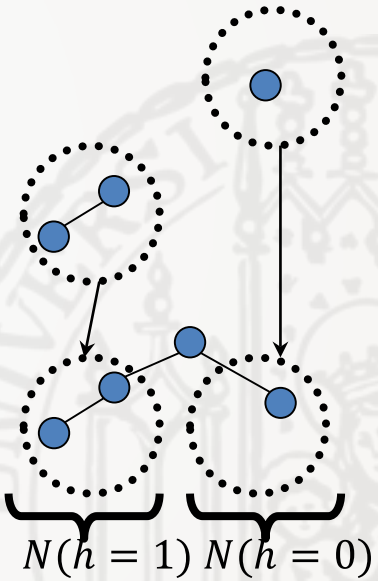


AVL-Baum $|\Delta h| \leq 1$

- Untersuchung der Komplexität
  - Die Operation Search hängt weiterhin von der Höhe des Baums ab.
  - Frage: Wie hoch kann ein AVL-Baum für eine gegebene Knotenanzahl n maximal werden?
  - Oder: Aus wie vielen Knoten muss ein AVL-Baum der Höhe h mindestens bestehen?

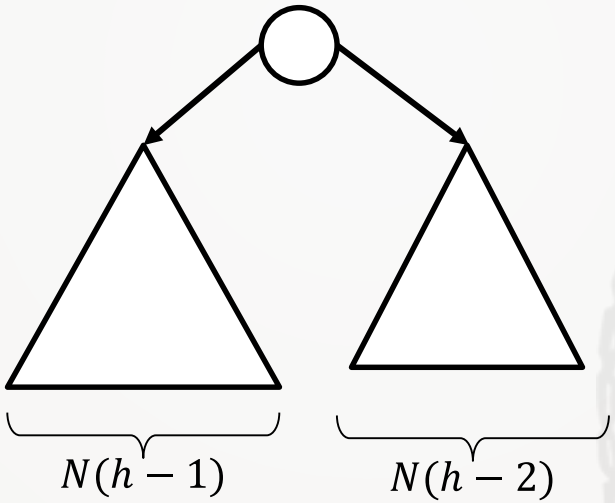
AVL-Bäume: Anzahl der Knoten

- Gesucht ist die minimale Knotenanzahl.
- Betrachte minimal gefüllte Bäume. $N(h)$ sei die minimale Anzahl Knoten eines AVL-Baums der Höhe h. Wir betrachten kleine Basisfälle:
  - Höhe $h = 0$: $N(h) = 1$
    Wurzel
  - Höhe $h = 1$: $N(h) = 2$
    nur ein Zweig gefüllt
  - Höhe $h = 2$: $N(h) = 4$
    Wurzel mit einem min. Baum $h = 1$
    und einem min. Baum $h = 2$



AVL-Bäume: Anzahl der Knoten

- Für beliebigen minimal gefüllten AVL-Baum der Höhe $h \geq 2$ gilt:
  - Die Wurzel besitzt zwei Teilbäume
  - Ein Teilbaum hat die Höhe $h - 1$
  - Der andere Teilbaum hat die Höhe $h - 2$



AVL-Bäume: Anzahl der Knoten

- Für beliebigen minimal gefüllten AVL-Baum gilt damit:

$$N(h) = \begin{cases} 1 & ,h = 0 \\ 2 & ,h = 1 \\ N(h-1) + N(h-2) + 1 & ,h > 1 \end{cases}$$

  - $N(h) = 1, 2, 4, 7, 12, 20, 33, 54$
- Zur Erinnerung die Fibonacci-Reihe:

$$fib(h) = \begin{cases} 0 & ,h = 0 \\ 1 & ,h = 1 \\ fib(h-1) + fib(h-2) & ,h > 1 \end{cases}$$

  - $fib(h) = 0, 1, 1, 2, 3, 5, 8, 13, 21, 34$

- Beweisbar: $N(h) = fib(h+3) - 1$

- Daher heißen AVL-Bäume auch Fibonacci-Bäume.

AVL-Bäume: Höhe in Abhängigkeit der Knotenanzahl

- $N(h) = fib(h+3) - 1$
- Formel von Moivre-Binet: $fib(h) = \frac{\varphi^h - \psi^h}{\sqrt{5}}$ mit $\varphi = \frac{1+\sqrt{5}}{2}$, $\psi = \frac{1-\sqrt{5}}{2}$
- Für große h gilt: $fib(h) \approx \frac{\varphi^h}{\sqrt{5}}$
- Damit gilt für $N(h) \leq n$:

$$fib(h+3) - 1 \leq n$$

$$\Leftrightarrow \frac{\varphi^{h+3}}{\sqrt{5}} \leq n+1$$

$$\Leftrightarrow h + 3 - \log_\varphi \sqrt{5} \leq \log_\varphi (n+1)$$

$$\Leftrightarrow h \leq \log_\varphi (n+1) + const$$

$$\Leftrightarrow h \leq \frac{\log_2 (n+1)}{\log_2 \varphi} = 1.4404 \cdot \log_2 (n+1) + const$$

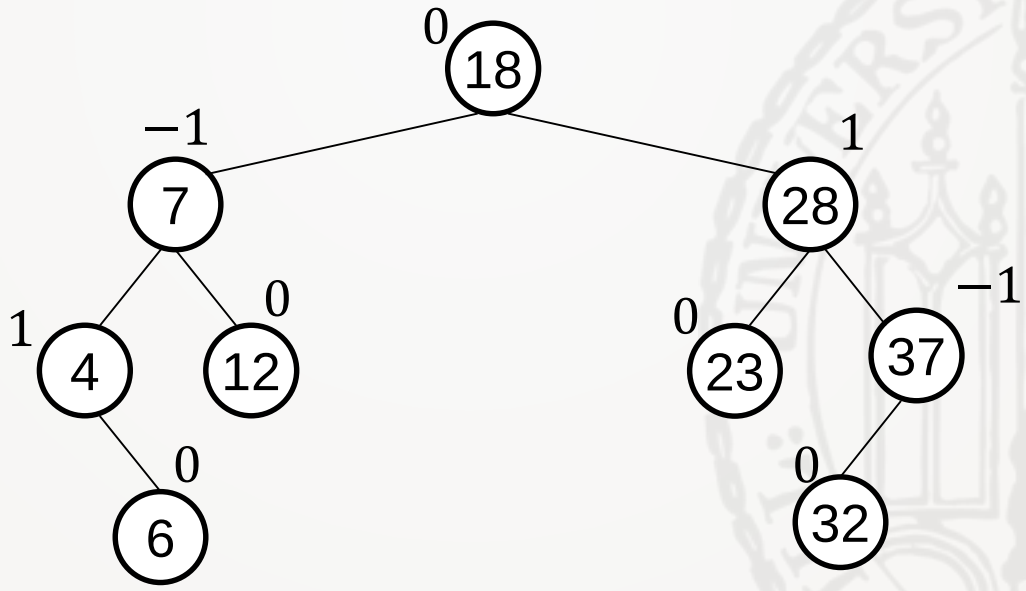
- Ein AVL-Baum ist maximal 44% höher als ein maximal ausgeglichener binärer Suchbaum (Suchkomplexität).

AVL-Bäume: Balance

- Wie müssen die Operationen Einfügen und Löschen verändert werden, damit die Balance eines AVL-Baums gewährleistet wird?

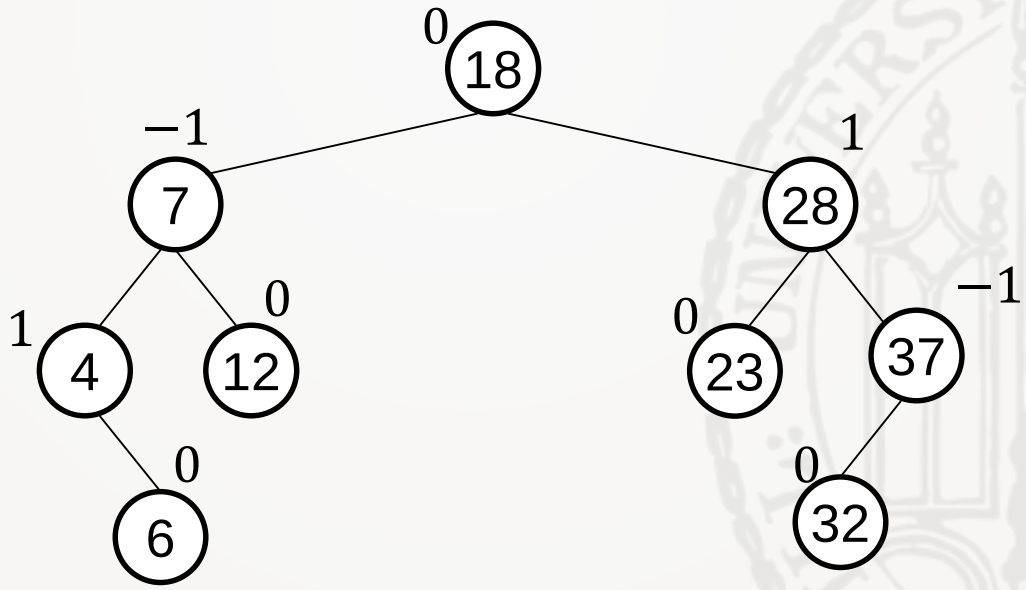
- Wir speichern bei jedem Knoten die Höhendifferenz (Balance b) der beiden Teilbäume:

$$b = \text{Höhe(rechter Teilbaum)} - \text{Höhe(Linker Teilbaum)}$$



AVL-Bäume: Einfügen

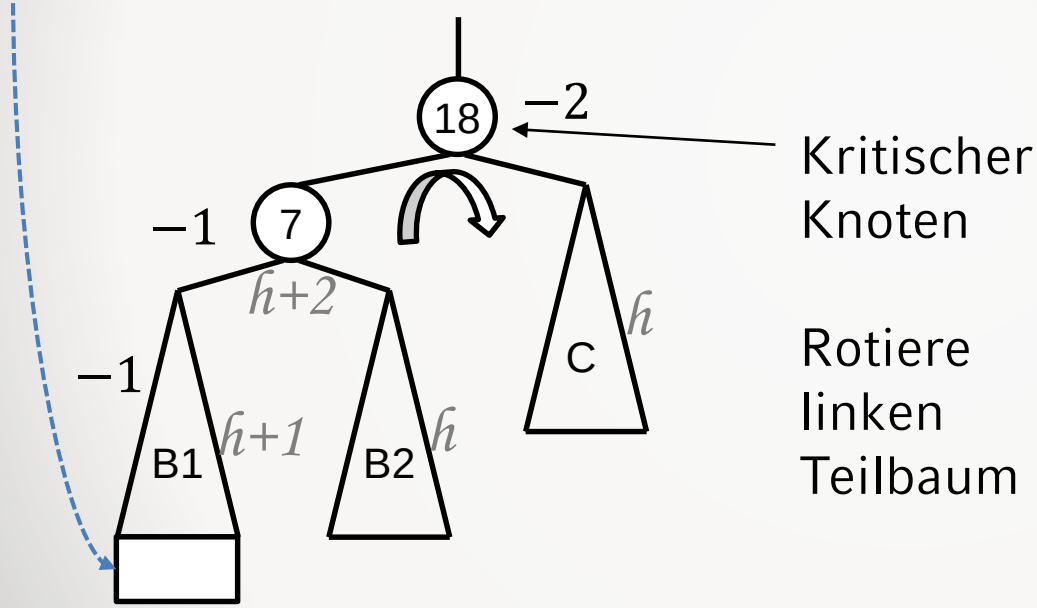
- Zuerst normales Einfügen wie bei binären Bäumen.
- Beim Einfügen kann sich nur die Balance b von Knoten ändern, die auf dem Suchpfad liegen.
- Wird das AVL-Kriterium verletzt, gehe den Suchpfad zurück und aktualisiere die Balance.



AVL-Bäume: Einfachrotation

- Rechtsrotation (rechts-rechts)

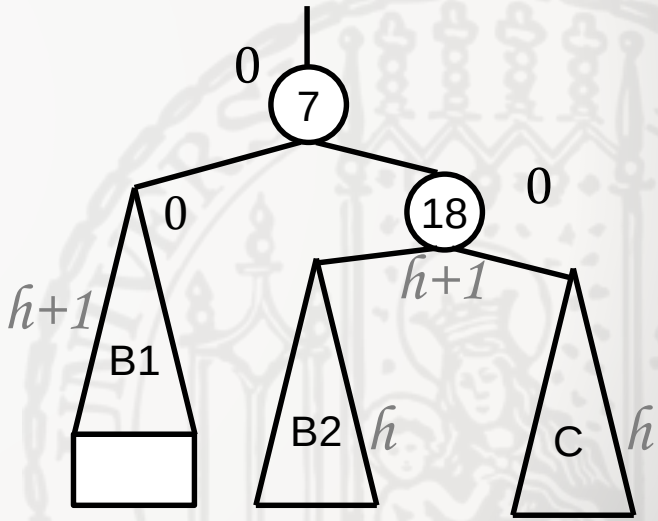
Beispiel: Einfügung war in Teilbaum „links links“ (Balance = −2)



Kritischer Knoten

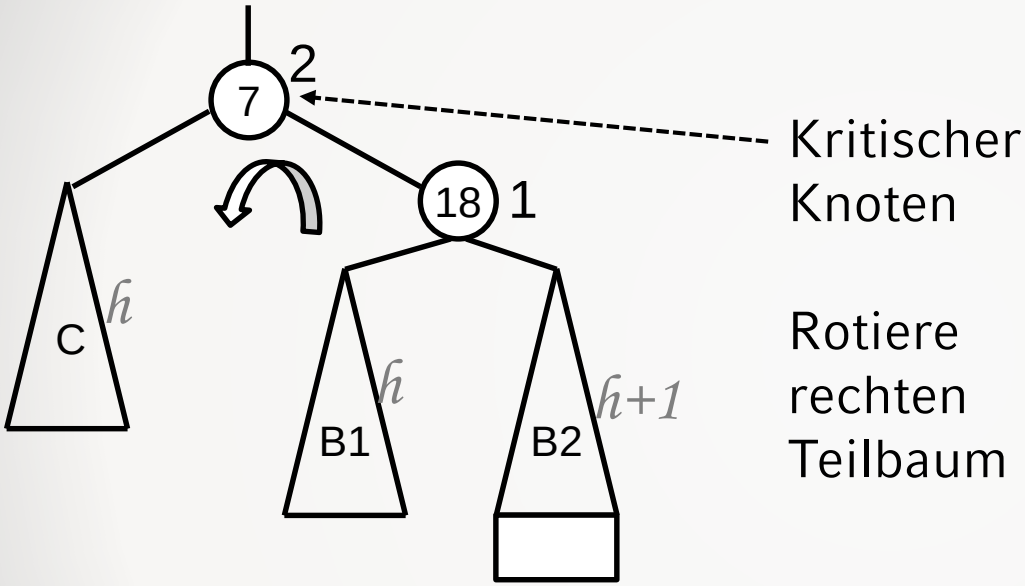
Rotiere linken Teilbaum

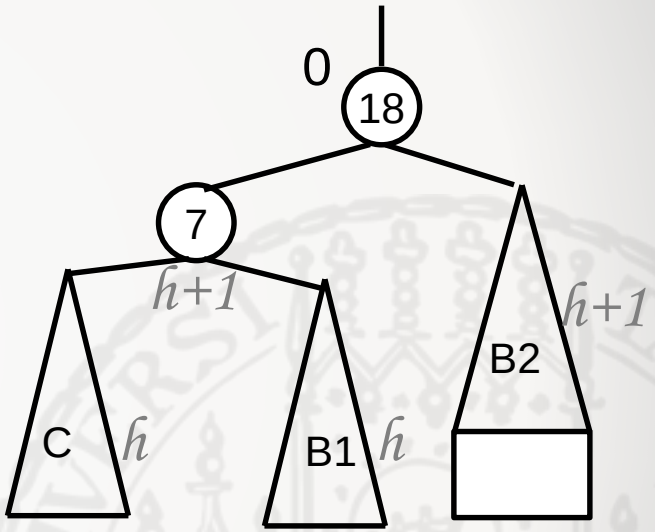
Baum ist nach der Rotation wieder balanciert



AVL-Bäume: Einfachrotation

- Linksrotation (links-links)

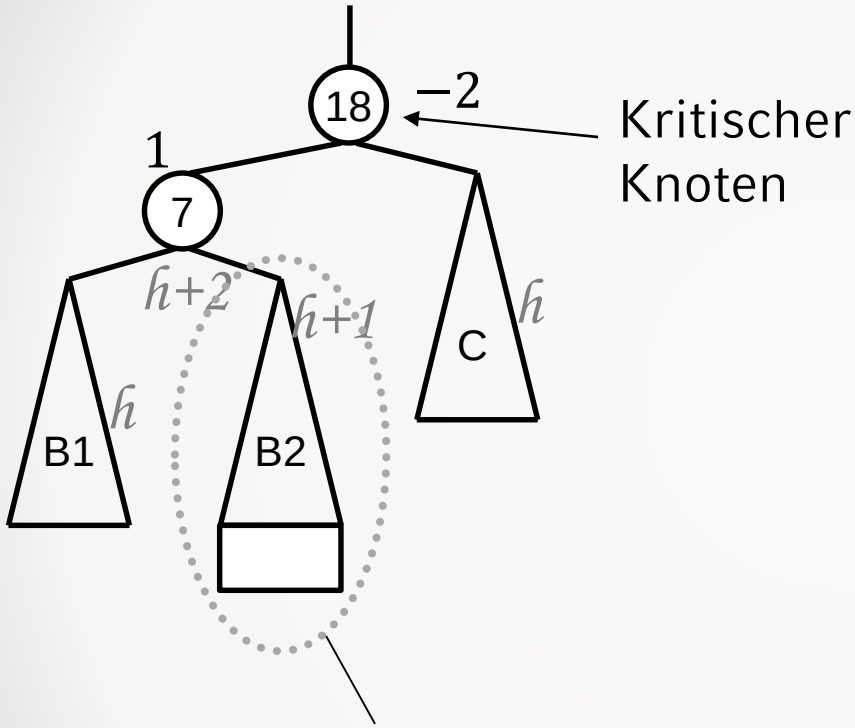


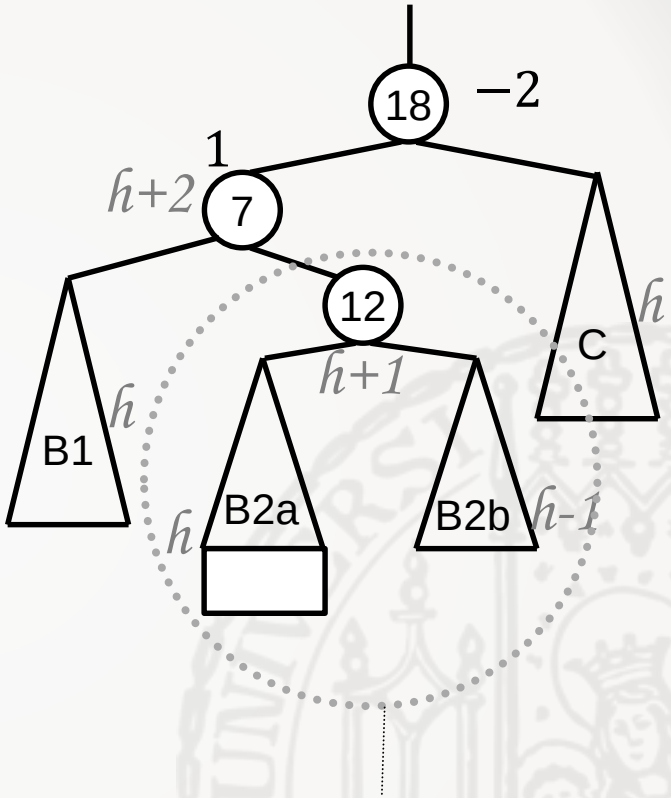


Symmetrisch zur Rechtsrotation

AVL-Bäume: Doppelrotation

- LR-Rotation (links-rechts)



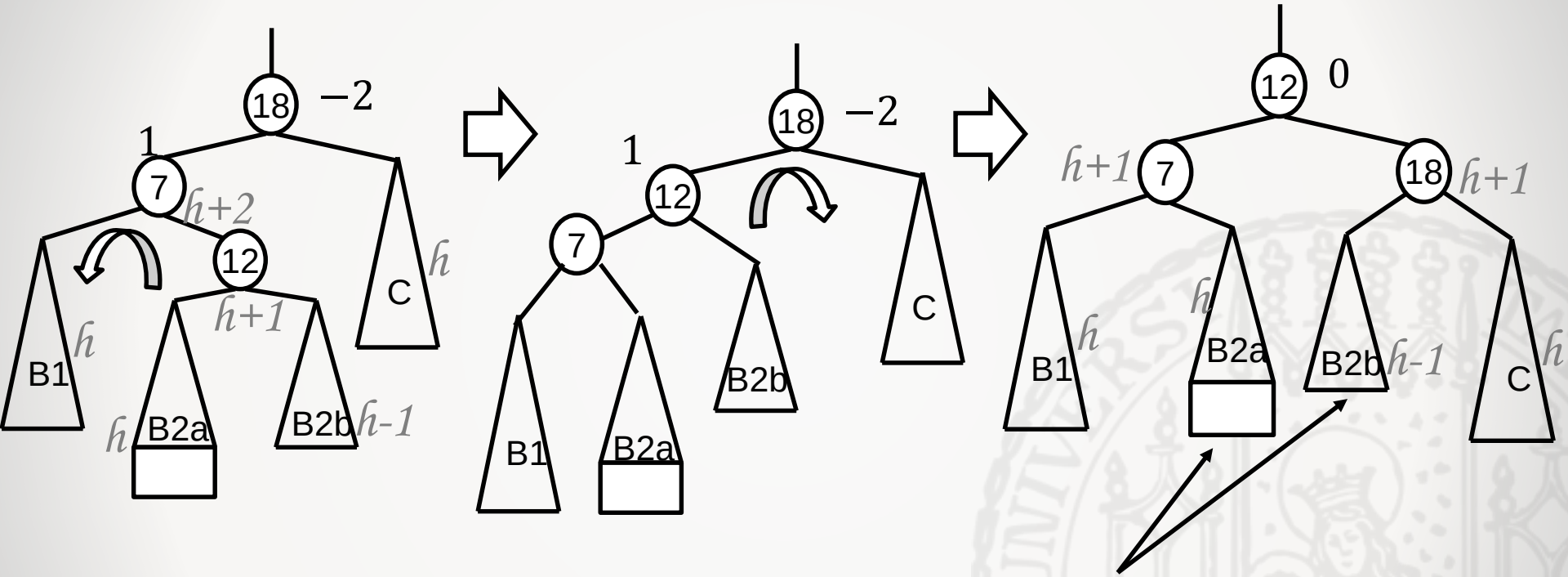


Eine einfache Rotation ist nicht mehr ausreichend, da der problematische Baum innen liegt

→ der Baum B2 muss näher betrachtet werden

AVL-Bäume: Doppelrotation

- LR-Rotation (links-rechts)



Wie man sieht, ist es dabei egal, ob der neue Knoten im Teilbaum B2a oder B2b eingefügt wurde
Die RL-Rotation geht analog zur LR-Rotation (symmetrischer Fall)

AVL-Bäume: Komplexität beim Einfügen

- Die Rotationen stellen das AVL-Kriterium im rebalancierten Unterbaum wieder her und sie bewahren die Sortierreihenfolge

- Wenn ein Baum rebalanciert wird, ist der entsprechende Unterbaum danach immer genauso hoch wie vor dem Einfügen.

  → der restliche Baum bleibt konstant und muss nicht überprüft werden

  → beim Einfügen eines Knotens benötigt man höchstens eine Rotation zur Rebalancierung.

Aufwand:

Einfügen	+	Rotieren		
$O(h)$	+	$O(1)$	=	$O(\log(n))$

AVL-Bäume: Löschen

Vorgehensweise

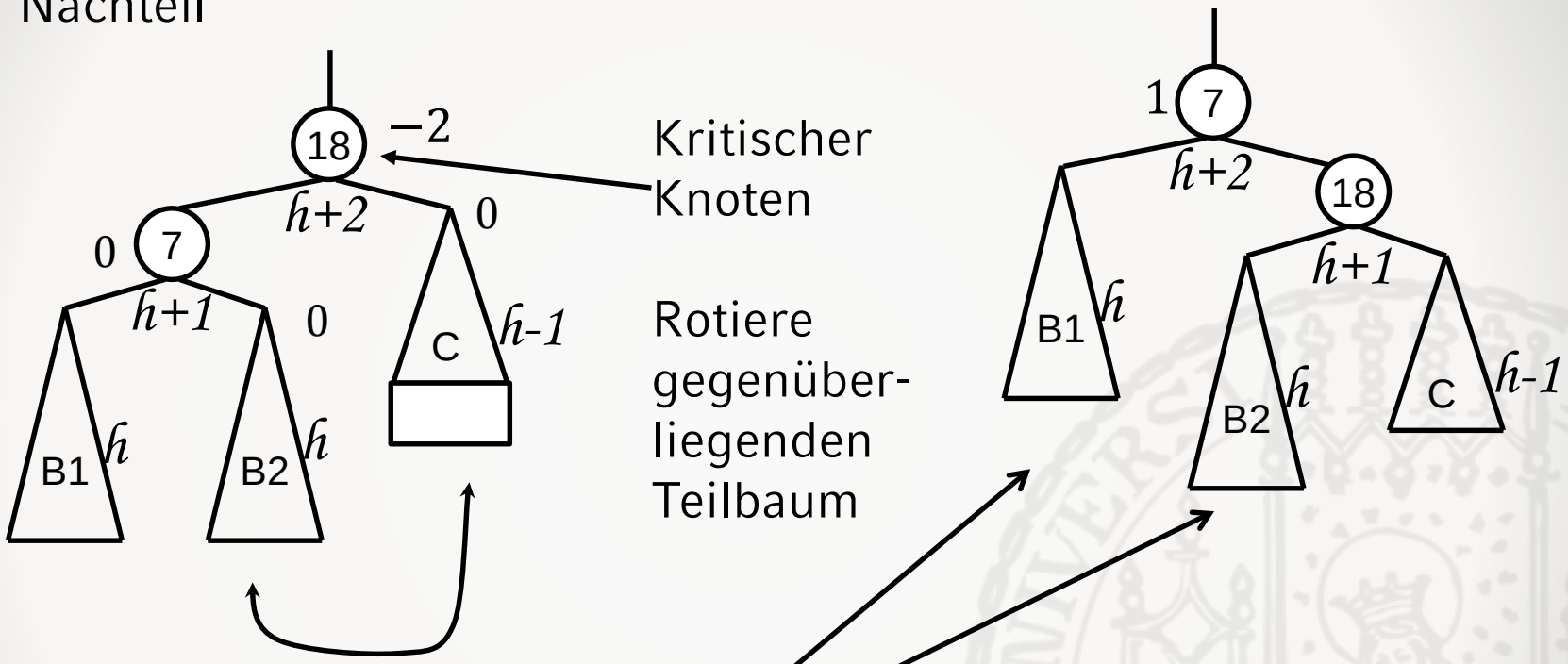
- Zuerst „normales“ Löschen wie bei binären Bäumen
- Nur für Knoten auf diesem Pfad kann das AVL-Kriterium verletzt werden (wie beim Einfügen)

Ablauf:

- Nach dem „normalen“ Löschen den kritischen Knoten bestimmen (nächster Vorgänger zum tatsächlich entfernten Knoten mit Balance $b = \pm 2$)
- Dieser ist Ausgangspunkt der Reorganisation (hier Rotation genannt)
- Rotationstyp wird bestimmt, als ob im gegenüberliegenden Unterbaum ein Knoten eingefügt worden wäre

AVL-Bäume: Löschen

- Nachteil



- Wie man sieht, ist der linke Teilbaum danach nicht mehr vollkommen ausbalanciert
- D.h., AVL-Balance wird zum Teil durch Abnahme von vollkommenen Teilbaumbalancen erkauft.

AVL-Bäume: Komplexität beim Löschen

- Beim Löschen eines Knotens wird
  - das AVL-Kriterium wiederhergestellt, die Sortierreihenfolge bleibt erhalten
  - kann es vorkommen, dass der re-balancierte Unterbaum nicht die gleiche Höhe wie vor dem Löschen besitzt

  $\rightarrow$ auf dem weiteren Pfad zur Wurzel kann es zu weiteren Re-Balancierungen (des obigen Typs, also immer im anderen Unterbaum) kommen

  $\rightarrow$ beim Löschen werden maximal h Rotationen benötigt

Aufwand:

Entfernen	+	Rotieren	
$O(h)$	+	$O(h)$	$= O(\log(n))$

Splay-Bäume

- Problem bei AVL-Bäumen:
  Basieren auf Prämisse der Gleichverteilung der Anfragen.
- Bei Nicht-Gleichverteilung (einige Anfragen treten häufiger auf), ist es wünschenswert, wenn sich der Baum an diese anpasst.
  $\rightarrow$ Splay-Bäume

- Splay-Bäume sind selbstoptimierende Binärbäume, für die keine Balancierung notwendig ist.

- Grundidee:
  - Bei jeder Suche nach einem Schlüssel wird dieser durch Rotationen zur Wurzel des Suchbaums.
  - Nachfolgende Operationen lassen den Schlüssel schrittweise tiefer in den Baum wandern.
  - Wird regelmäßig der gleiche Schlüssel angefragt, so wandert er nicht besonders tief in den Baum und kann somit schneller gefunden werden.

Splay-Bäume: Eigenschaften

- Splay-Bäume basieren auf den normalen Operationen Suchen, Einfügen und Löschen.
- Nur Suchen und Einfügen sind mit der Operation Splay gekoppelt.
- Splay platziert das betreffende Element als Wurzel des Baums.
- Splay-Bäume haben keine strukturelle Invariante wie AVL-Bäume, welche für deren Effizienz verantwortlich ist. Einzig die Splay-Operation führt zu einer heuristischen Restrukturierung.

Splay-Bäume: Operationen

- Suchen
  - Normale Binärsuche im Suchbaum
  - Endet in Knoten x mit Schlüssel k
  - Bei Erfolg: Wende Operation Splay auf Knoten x an
  - Sonst: NOP (no-operation)

- Einfügen
  - Normale Binärsuche im Suchbaum
  - Einfügen eines Knotens als Blatt
  - Wende Splay auf diesen Knoten an

- Löschen
  - Normale Binärsuche im Suchbaum
  - Entferne den gefundenen Knoten wie im Binärbaum

Splay-Bäume: Splay-Operation

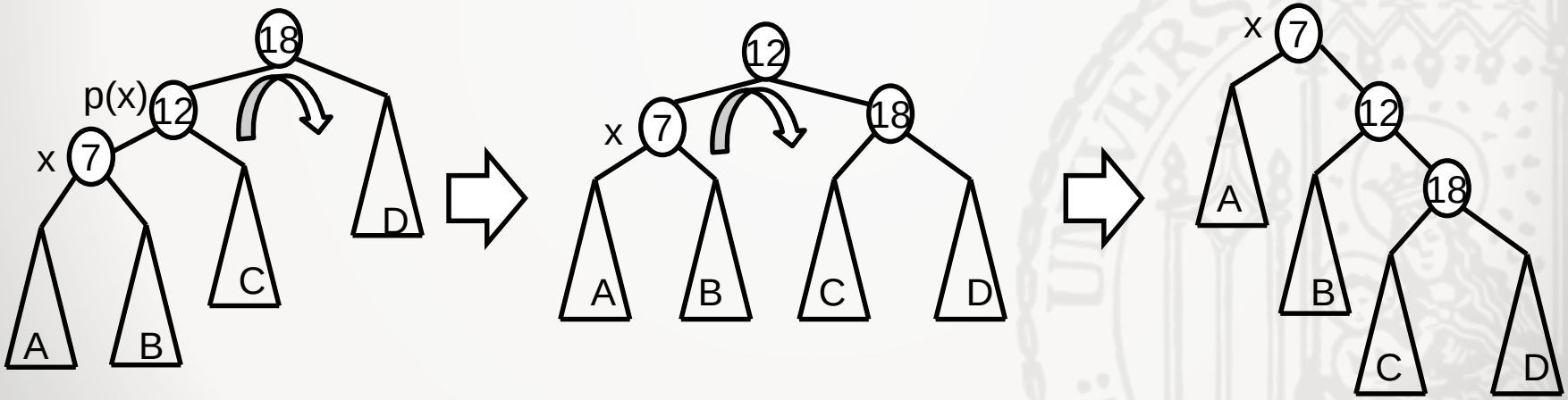
- Der Splay repositioniert einen gegebenen Baumknoten als Wurzel.
- Umsetzung: Sukzessives Rotieren, bis der Knoten die Wurzel ist.

- Zwei Varianten
  - Bottom-Up Splay Tree
    - Suchphase: Steige von Wurzel zum gesuchten Knoten ab.
    - Splayphase: Rotiere auf dem Weg zurück den Knoten nach oben.

  - Top-Down Splay Tree
    - Behandle alle Knoten auf dem Pfad nur einmal.
    - Ziehe die Knoten dazu von der Wurzel aus nach oben, anstatt im Pfad abzusteigen.
    - Kleinere Knoten werden nach links rotiert, größere nach rechts, bis der gesuchte Knoten als Wurzel vorliegt.

Top-Down Splay-Operation

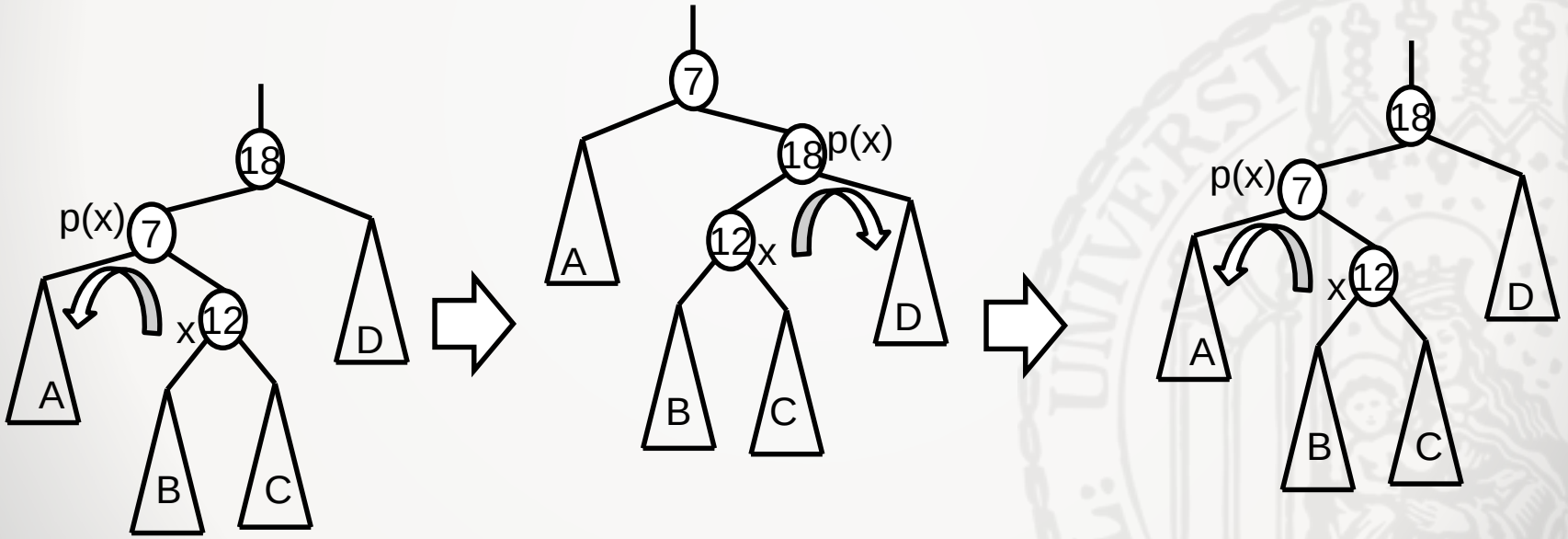
- Beispiel
  - Knoten 7 liegt im linken Unterbaum: rotiere 18 nach rechts
  - Knoten 7 liegt im linken Unterbaum: rotiere 12 nach rechts



Top-Down Splay-Operation

- Beispiel
  - Knoten 12 liegt im linken Unterbaum: rotiere 18 nach rechts
  - Knoten 12 liegt im rechten Unterbaum: rotiere 7 nach links

Geht nicht, wir brauchen doch eine Doppelrotation!!!



Splay-Bäume: Komplexität

- Ein Splay-Baum mit n Knoten hat eine amortisierte Zeitkomplexität von $O(\log n)$.
- Amortisierte Komplexität berechnet die durchschnittliche Komplexität über eine Worst-Case-Sequenz von Operationen im Gegensatz zu einer reinen Worst-Case-Abschätzung aller Operationen.

- Es lässt sich zeigen, dass sich Splay-Bäume asymptotisch wie optimale Suchbäume verhalten.
  - *Optimal* ist ein Suchbaum, wenn die erwartete Anzahl an Vergleichen für eine Suchanfrage minimal ist.

Splay-Bäume: Zusammenfassung

- Gut geeignet zur Umsetzung von Caches oder Garbage Collection.
- keine Strukturinvariante wie bei AVL, d.h. sie sind speichereffizienter als diese, da die Knoten keine zusätzlichen Informationen speichern müssen(z.B. den Balancegrad).
- Geringerer Programmieraufwand.

Verwendung von Sekundärspeicher

Motivation

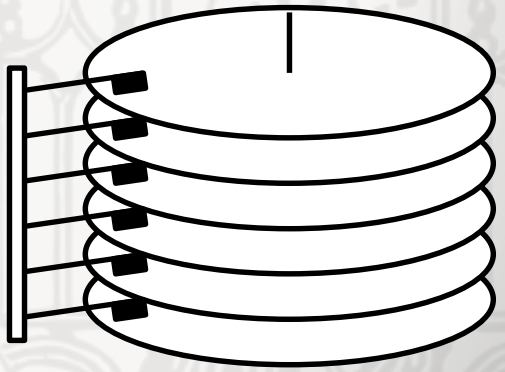
- Falls Daten persistent gespeichert werden müssen
- Falls Datenmenge zu groß für Hauptspeicher

Sekundärspeicher/Festplatte

- Festplatte besteht aus übereinanderliegenden rotierenden Platten mit magnetischen/optischen Oberflächen, die in Spuren und Sektoren eingeteilt sind

Zugriffszeit Festplatten

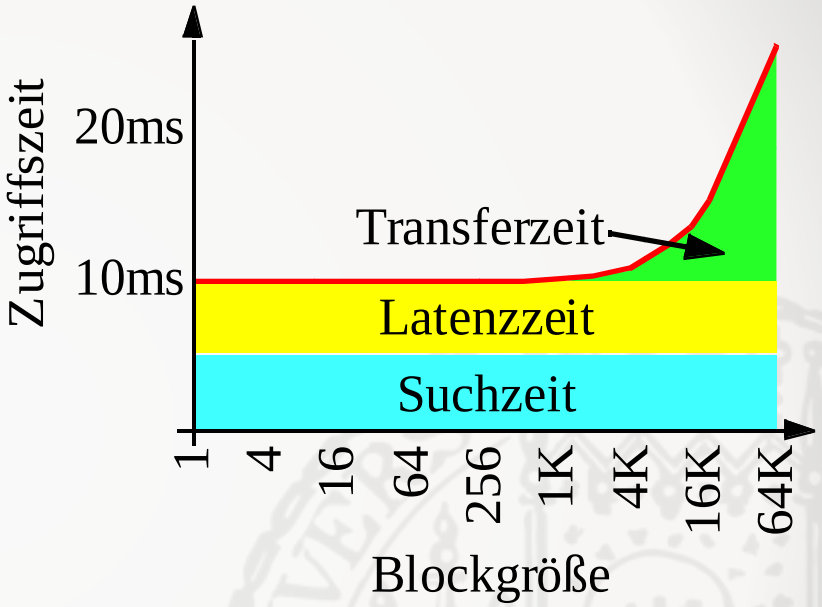
- **Suchzeit** [ms]: Armpositionierung (Translation)
- **Latenzzeit** [ms]: Rotation bis Blockanfang
- **Transferzeit** [ms/MB]: Übertragung der Daten



Verwendung von Sekundärspeicher

Blockgrößen

- Größere Transfereinheiten sind günstiger
- Gebräuchlich sind Seiten der Größe 4kB oder 8kB



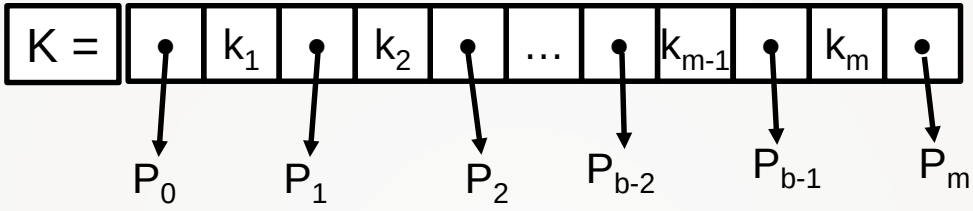
Problem

- Seitenzugriffe sind teurer als Vergleichsoperationen

- Ziel: Möglichst viele ähnliche Schlüssel auf einer Seite (Block) speichern

Mehrwegbäume

Knoten haben $n \geq 2$ Nachfolger

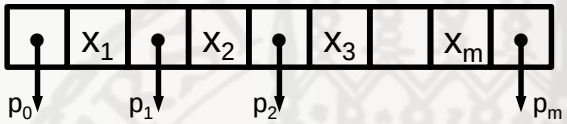


Knoten $K = (P_0, k_1, P_1, k_2, P_2, \ldots, P_{m-1}, k_m, P_m)$ eines n-Wege-Suchbaums B besteht aus:

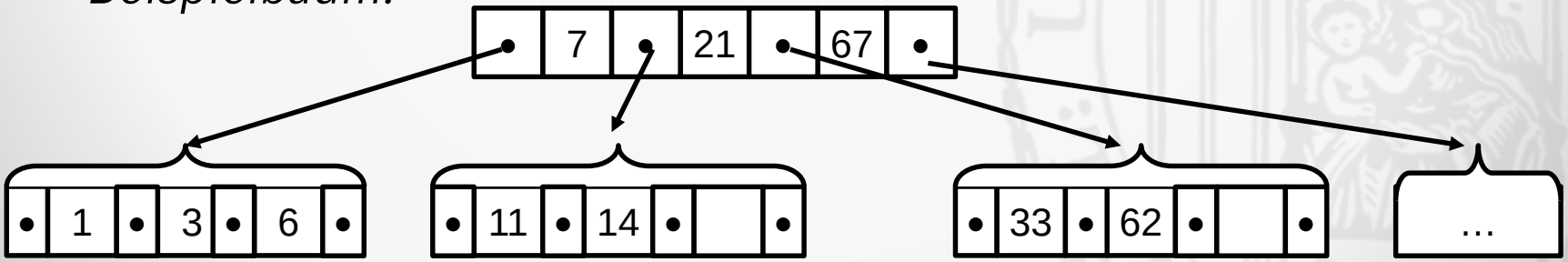
- Grad: $m = grad(K) \leq n$
- Schlüssel: $k_i \ (1 \leq i \leq m)$
- Zeiger: P_i auf die Unterbäume $(0 \leq i \leq m)$

B-Baum: Suchbaumeigenschaft

- Innerhalb einer Seite sind die Schlüssel bzgl. der auf ihnen definierten Ordnung sortiert, sie liegen logisch jeweils zwischen zwei Verweisen auf ein Kind.
- Das bedeutet: Sei $K(p)$ die Menge der Schlüssel in dem von p referenzierten Teilbaum, m sei die Anzahl der in der Seite gespeicherten Schlüssel. Dann gilt:
  - $\forall y \in K(p_0)\colon\ y < x_1$
  - $\forall y \in K(p_i), 1 < i < m-1\colon\ x_i < y < x_{i+1}$
  - $\forall y \in K(p_m)\colon\ x_m \leq y$

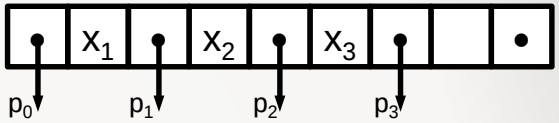


- *Beispielbaum:*

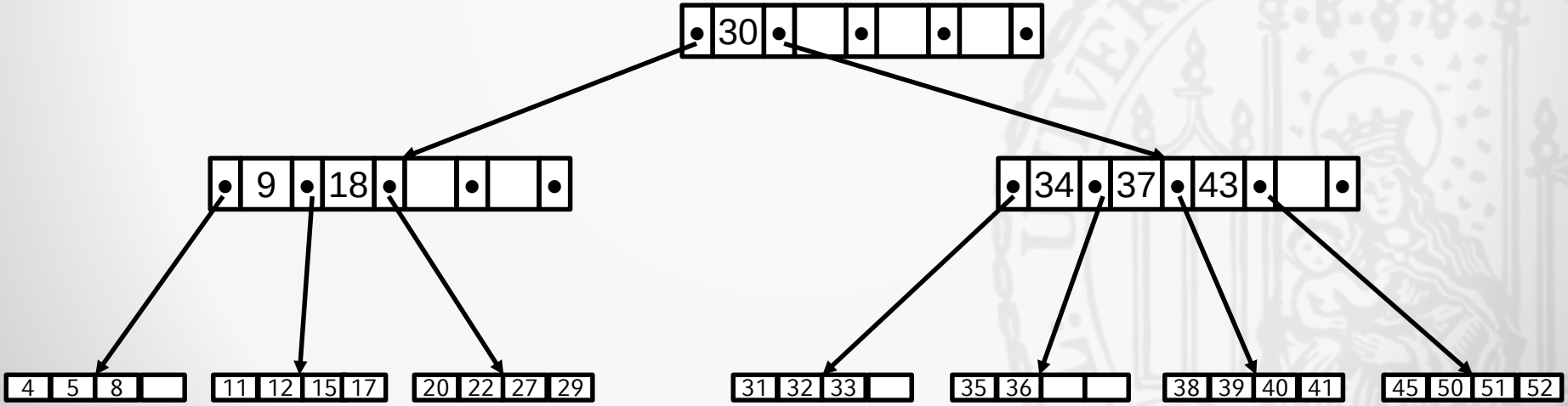


B-Baum: Beispiel

- Die Ordnung $k \in \mathbb{N}$ bestimmt man aus der Größe einer Plattenseite
  - Beispielgrößen: Seite 4 kByte, Objekt 42 Byte, Zeiger 8 Byte
  - 2k Objekte + (2k + 1) Zeiger = 4096 Byte
  - Damit $2k \approx (4096 - 8) / 50 = 81$, also $k = 40$.

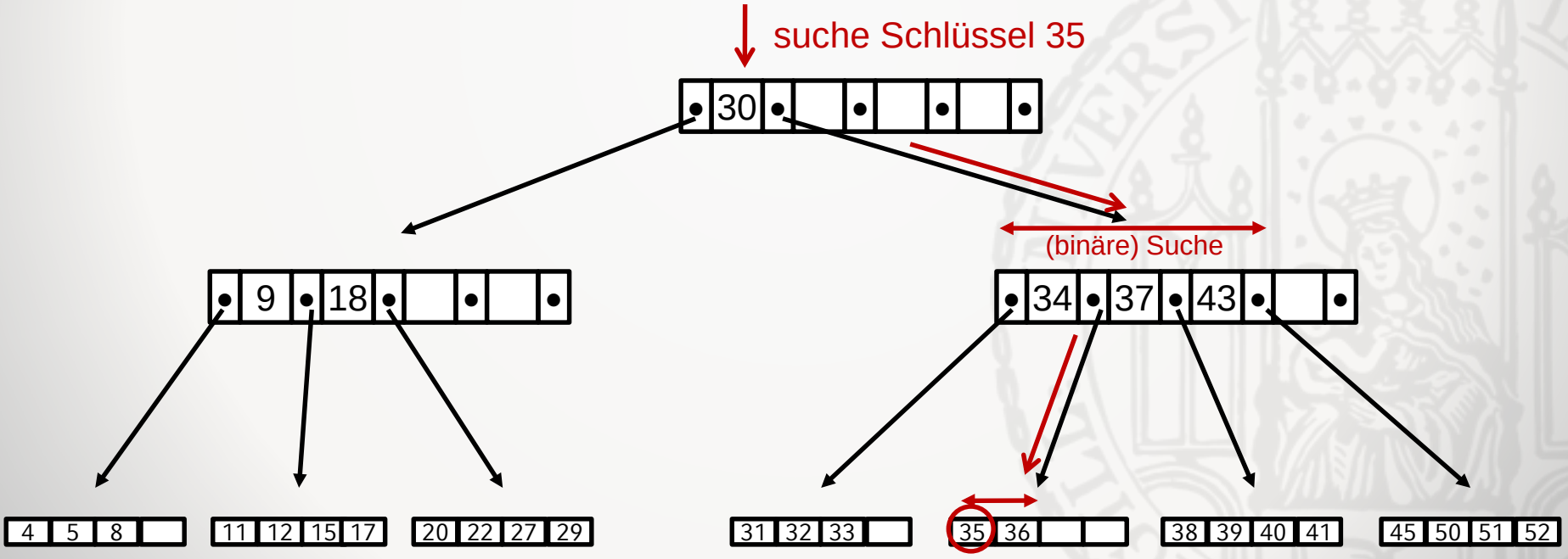


- Beispiel: $k = 2, h = 2$



B-Baum: Suche

- Suchen eines Objektes anhand eines Suchschlüssels
  - Beginne Suche in der Wurzel
  - (binäre) Suche auf jeweiligem Knoten
    - Falls gefunden: Rückgabe des Objektes
    - Sonst: im entsprechenden Teilbaum rekursiv weitersuchen
    - Falls keine Teilbäume existieren (Blattebene): Misserfolgsmeldung



B-Baum: Höhenabschätzung

Wurzel: $1 \leq \#\text{Schlüssel} \leq 2k, \quad 2 \leq \#\text{Verweise} \leq 2k+1$
Knoten: $k \leq \#\text{Schlüssel} \leq 2k, \quad k+1 \leq \#\text{Verweise} \leq 2k+1$

Wie viele Schlüssel sind mindestens in einem B-Baum der Ordnung k enthalten? – Wichtig: im B-Baum sind alle Blätter auf der selben Höhe.

Höhe	Schlüssel (pro Ebene)	Verweise (pro Ebene)	$k = 10$ (gesamt)	$k = 100$ (gesamt)
0	1	2	1	1
1	$2 * k$	$2(k+1)$	$20 + 1 = 21$	201
2	$2(k+1) * k$	$2(k+1) * (k+1)$	$220 + 21 = 241$	20.401
3	$2(k+1)^2 * k$	$2(k+1)^2 * (k+1)$	$2420 + 241 = 2861$	2.060.601
4	$2(k+1)^3 * k$	$2(k+1)^4$	$26620 + 2861 = 29.481$	208.120.801

Damit hat ein Baum mit 1000 Schlüsseln maximal die Höhe 1. Ein Baum mit 1.000.000 Schlüsseln wird nie höher als Höhe 2.

B-Baum: Höhenabschätzung

Wurzel: $1 \leq$ #Schlüssel $\leq 2k$, $2 \leq$ #Verweise $\leq 2k+1$
Knoten: $k \leq$ #Schlüssel $\leq 2k$, $k+1 \leq$ #Verweise $\leq 2k+1$

Höhe	Schlüssel	Verweise	k beliebig
0	1	2	1
1	$2 * k$	$2(k+1)$	$2k+1$
2	$2(k+1) * k$	$2(k+1) * (k+1)$	$2k\big((k+1)+1\big)+1$
3	$2(k+1)^2 * k$	$2(k+1)^2 * (k+1)$	$2k\big((k+1)^2+(k+1)+1\big)+1$
4	$2(k+1)^3 * k$	$2(k+1)^4$	$2k\big((k+1)^3+(k+1)^2+(k+1)+1\big)+1$

Die minimale Schlüsselzahl s_{min} eines Baumes der Höhe $h \geq 0$:

$$s \geq s_{min} = 1 + 2k \sum_{i=0}^{h-1} (k+1)^i = 2(k+1)^h - 1$$

Damit ist die maximale Höhe logarithmisch abhängig von der Anzahl der Schlüssel s:

$$h \leq \log_{(k+1)} \frac{s+1}{2} \approx \log_{k+1} s$$

B-Baum: Höhenabschätzung

Wurzel: $1 \leq$ #Schlüssel $\leq 2k$, $2 \leq$ #Verweise $\leq 2k+1$
Knoten: $k \leq$ #Schlüssel $\leq 2k$, $k+1 \leq$ #Verweise $\leq 2k+1$

Wie viele Schlüssel sind <u>maximal</u> in einem B-Baum der Ordnung k enthalten?

Höhe	Schlüssel	Verweise	$k = 10$	$k = 100$
0	$2k$	$2k+1$	20	200
1	$2k * (2k+1)$	$(2k+1)^2$	$420 + 20$ $= 440$	40.400
2	$2k * (2k+1)^2$	$(2k+1)^3$	$8.820 + 440$ $= 9.260$	8.120.600
3	$2k * (2k+1)^3$	$(2k+1)^4$	$185.220 + 9.260$ $= 194.480$	1.632.240.800
4	$2k * (2k+1)^4$	$(2k+1)^5$	$3.889.620$ $+194.489$ $= 4.084.109$	328.080.401.000

B-Baum: Höhenabschätzung

Wurzel: $1 \leq$ #Schlüssel $\leq 2k$, $2 \leq$ #Verweise $\leq 2k+1$
Knoten: $k \leq$ #Schlüssel $\leq 2k$, $k+1 \leq$ #Verweise $\leq 2k+1$

Höhe	Schlüssel	Verweise	k beliebig
0	$2k$	$2k+1$	$2k$
1	$2k*(2k+1)$	$(2k+1)^2$	$2k\big(1+(2k+1)\big)$
2	$2k*(2k+1)^2$	$(2k+1)^3$	$2k(1+(2k+1)+(2k+1)^2)$
3	$2k*(2k+1)^3$	$(2k+1)^4$	$2k(1+(2k+1)+(2k+1)^2+(2k+1)^3)$
4	$2k*(2k+1)^4$	$(2k+1)^5$	$2k(1+\cdots+(2k+1)^4)$

Die maximale Schlüsselzahl s_{max} eines Baumes der Höhe $h \geq 0$:

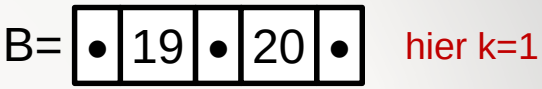
$$s \leq s_{max} = 2k\sum_{i=0}^{h}(2k+1)^i = (2k+1)^{h+1}-1$$

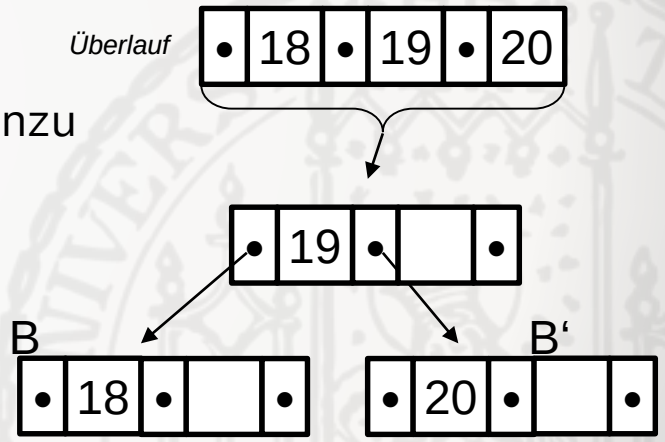
Damit ist die minimale Höhe logarithmisch abhängig von der Anzahl der Schlüssel s:

$$h \geq \log_{2k+1}(s+1)-1 \approx \log_{2k+1} s$$

B-Baum: Einfügen

- Durchlaufe den Baum und suche das Blatt B, in welches der neue Schlüssel gehört
- Füge x sortiert dem Blatt hinzu
- Wenn hierdurch das Blatt $B = (x_1, \ldots, x_{2k+1})$ überläuft → Ausgleich / Split
  - Erzeuge ein neues Blatt B'
  - Verteile Schlüssel auf altes und neues Blatt $B = (x_1, \ldots, x_k)$ und $B' = (x_{k+2}, \ldots, x_{2k+1})$
  - Füge den Schlüssel x_{k+1} dem Vorgänger hinzu ggf. erzeuge neuen Vorgänger: x_{k+1} dient als Trennschlüssel für B und B'
- Vorgänger kann auch überlaufen, ggf. rekursiv bis zur Wurzel weiter splitten
- Wenn die Wurzel überläuft
  - Wurzel teilen
  - Mittlerer Schlüssel x_{k+1} wird neue Wurzel mit zwei Nachfolgern

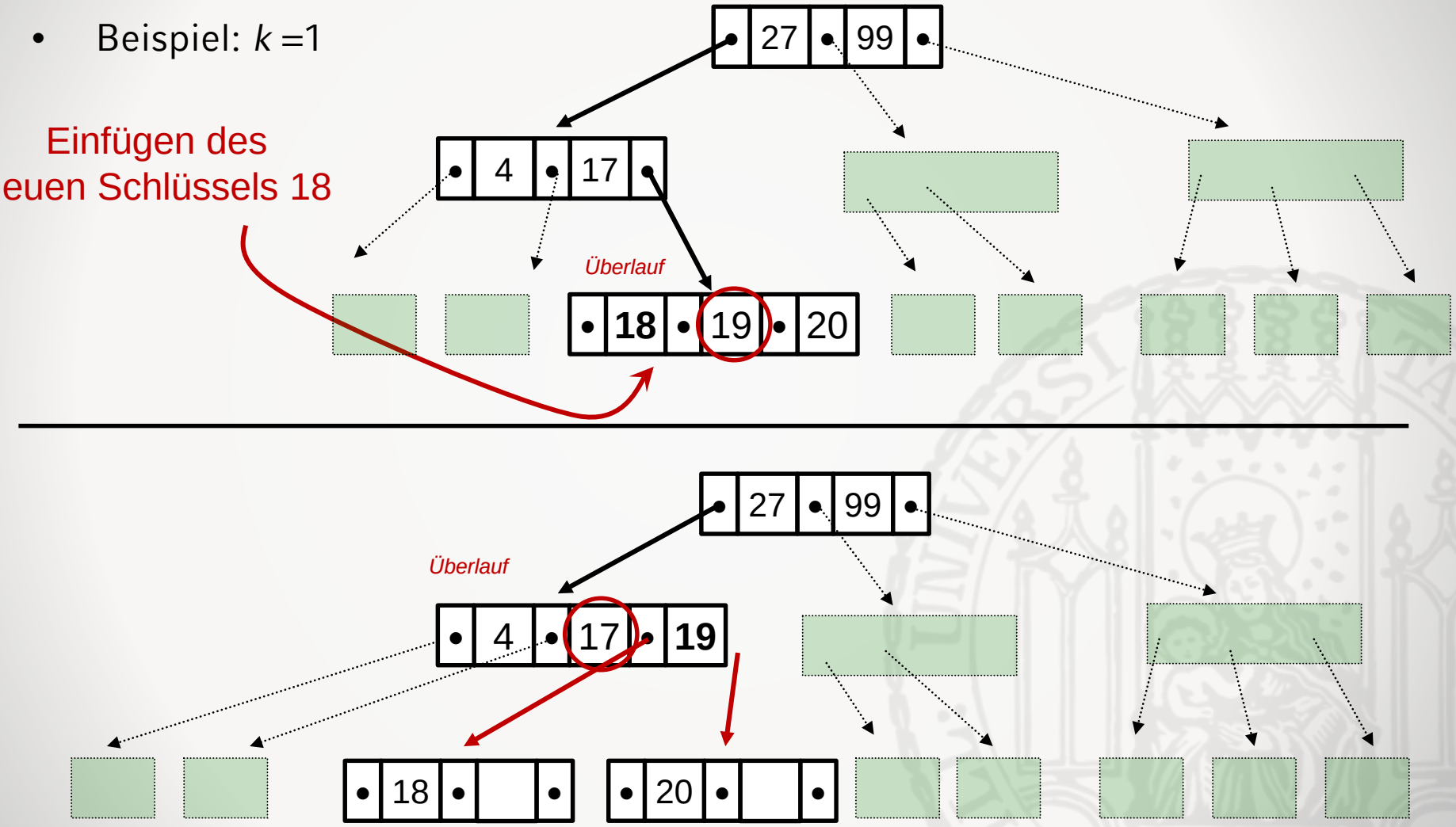




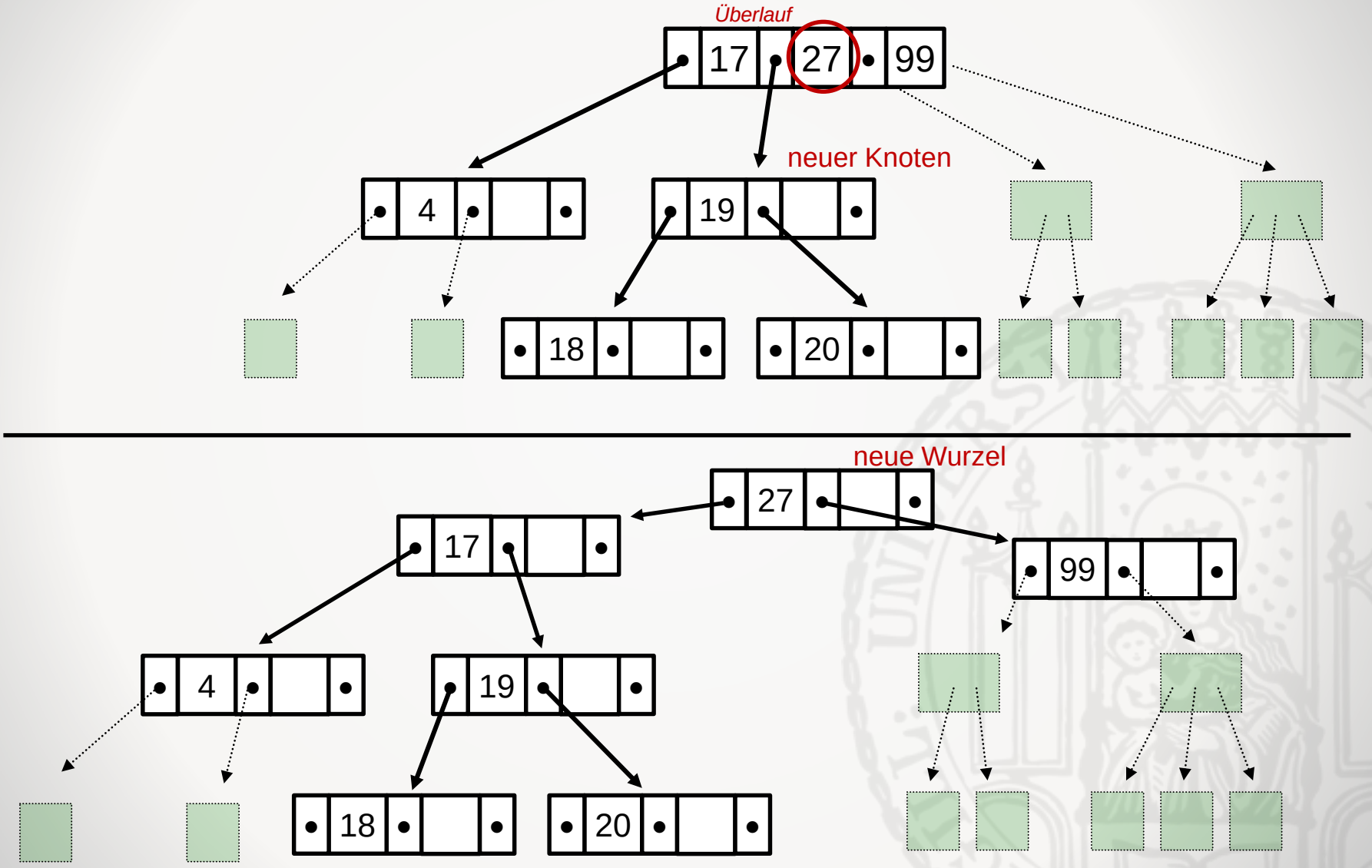
B-Baum: Einfügen

- Beispiel: $k = 1$

Einfügen des neuen Schlüssels 18



B-Baum: Einfügen



B-Baum: Löschen

- Suche den Knoten N, welcher den zu löschenden Schlüssel x enthält
- Falls B ein innerer Knoten
  - $\rightarrow$ Suche den größten Schlüssel x' im Teilbaum links des Schlüssel x
  - $\rightarrow$ Ersetze x im Knoten B durch x'
  - $\rightarrow$ Lösche x' aus seinem ursprünglichen Blatt B'
- Falls N ein Blatt ist, lösche den Schlüssel aus dem Blatt
  - $\rightarrow$ Hierbei ist es möglich, dass N nun weniger als k Schlüssel enthält
  - $\rightarrow$ Reorganisation unter Einbeziehung der Nachbarknoten
  - $\rightarrow$ Bemerkung: Nur die Wurzel hat keine Nachbarknoten und darf weniger als k Schlüssel enthalten

B-Baum: Unterlauf beim Löschen

- Nach einer Löschoperation sei der Knoten N unterläufig
- Fall 1: N ist die Wurzel
  - Wurzel wird gelöscht, wenn diese keine Schlüssel mehr beinhaltet → Baum ist leer
- Fall 2: N hat einen Nachbarn M mit mehr als k Schlüssel
  - Dann **Ausgleich** von N durch die Schlüssel x_i aus dem Nachbarknoten M unter Einbeziehung des Vorgängers
- Fall 3: N hat einen Nachbarn M mit genau k Elementen
  - Dann **Verschmelze** N und M inklusive dem zugehörigen Schlüssel im Vorgänger x zu einem Knoten
  - Entferne x aus dem Vorgänger
  - Im Vorgänger bleibt noch ein Zeiger auf den verschmolzenen Knoten bestehen

B-Baum: Unterlauf – Ausgleich

Aus dem Knoten $N = (x_1 \dots x_k)$ soll der Schlüssel x_i entfernt werden

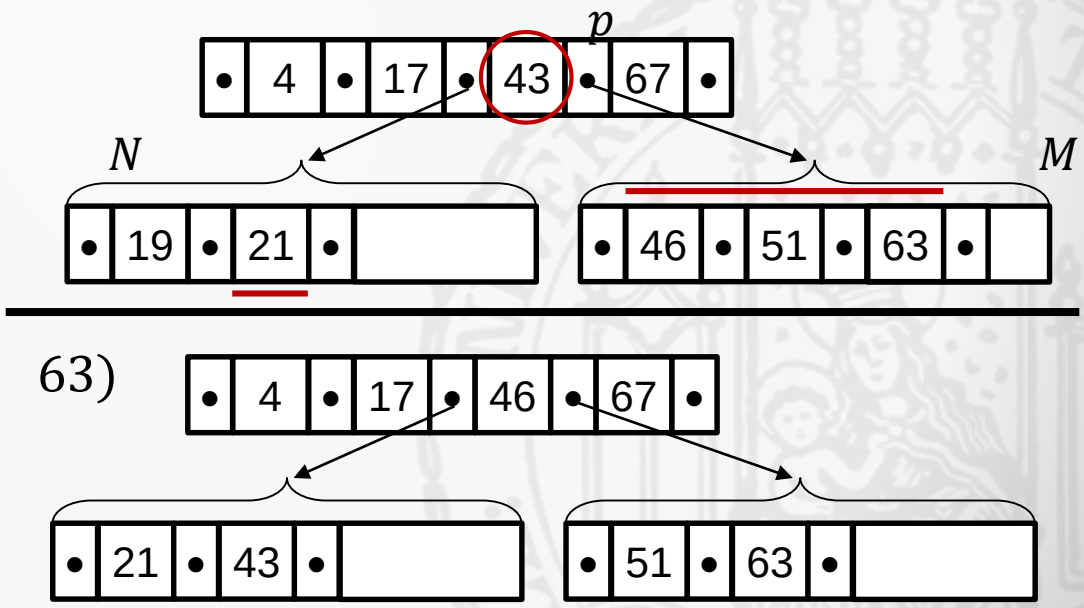
- Sei $M = (x'_1 \dots x'_k)$ ein Nachbarknoten mit **mehr** als k Schlüsseln $(n > k)$
- O.B.d.A sei M rechts von N und p der Trennschlüssel im Vorgänger
- Verteile die Schlüssel $x_1 \dots x_a, p, x'_1 \dots x'_n$ auf die Knoten M und N
- Ersetze den Schlüssel p im Vorgänger durch den mittleren Schlüssel
- M und N haben nun jeweils min. k Schlüssel

Beispiel:

B-Baum mit $k = 2$

Lösche Schlüssel 19

Ausgleich (21, 43, 46, 51, 63)



B-Baum: Unterlauf – Verschmelzen

Aus dem Knoten $N = (x_1 \dots x_k)$ soll der Schlüssel x_i entfernt werden

- Sei $M = (x'_1 \dots x'_k)$ ein Nachbarknoten mit **genau** k Schlüsseln
- O.B.d.A sei M rechts von N und p der Trennschlüssel im Vorgänger V
- Verschmelze die Knoten N und M zu M', füge p zu K' hinzu und lösche N
- Entferne p sowie den Verweis auf N aus dem Vorgänger V ggf. rekursiv bis zur Wurzel (enthält diese danach keine Schlüssel mehr, so wird das einzige Kind zur neuen Wurzel)

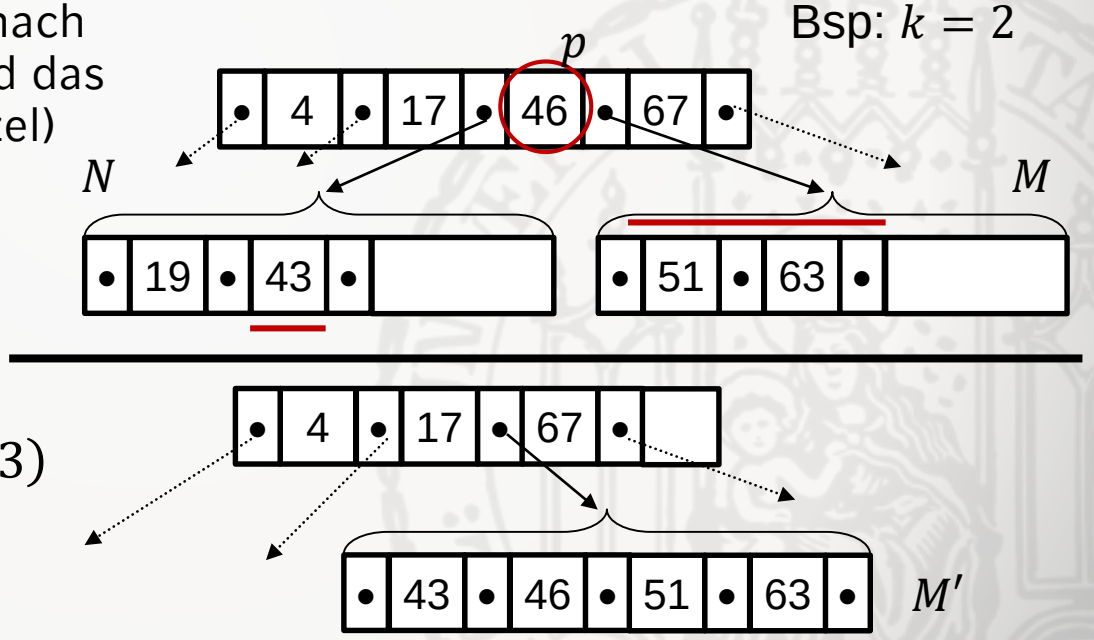
Beispiel:

B-Baum mit $k = 2$

Lösche Schlüssel 19

Verschmelze $(43, 46, 51, 63)$

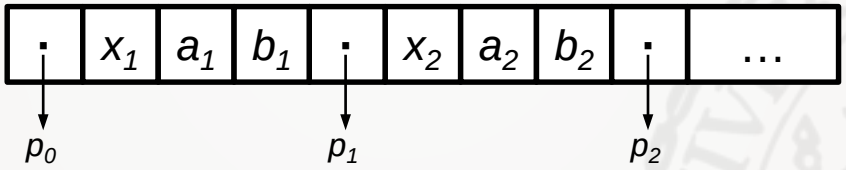
Entferne $(p = 46)$



Verschmelzen = inverse Operation zum Split

Praktische Anwendung: B+- Baum

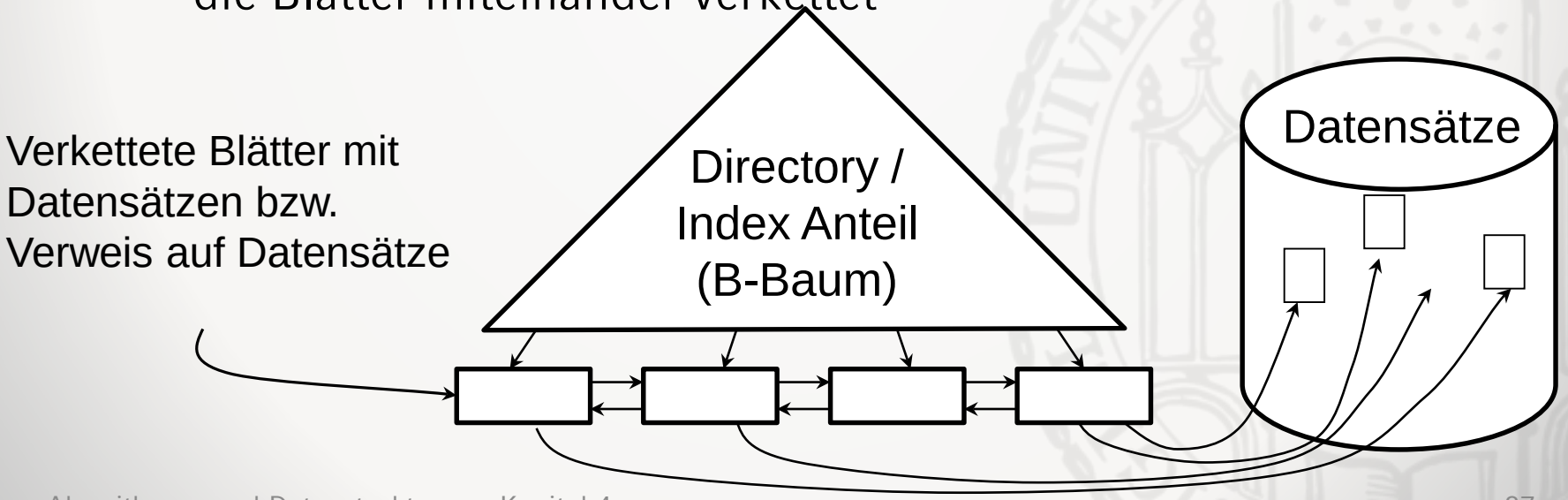
- In der Praxis will man neben den eigentlichen Schlüsseln oft zusätzlich noch weitere Daten (z.B. Attribute oder Verweise auf weitere Datensätze) speichern.
  - Bsp.: Zu einer Matrikelnummer soll jeweils noch Name, Studiengang, Anschrift, etc. abgelegt werden.
- Lösung: Speichere in den Knoten des B-Baums jeweils den Schlüssel x_i und dessen Attribute $a_i, b_i, c_i,$ …
  Bsp.:



- Problem: Durch mehr Daten in den inneren Knoten (Seiten) sinkt der Verzweigungsgrad und die Baumhöhe steigt → kontraproduktiv
- Lösung: Eigentliche Daten in die Blätter, im Baum darüber nur „Wegweiser“ → Konzept des B+-Baums

B^+-Baum

- Ein B^+-Baum ist abgeleitet vom B-Baum und hat zwei Knotentypen
  - Innere Knoten enthalten keine Daten (nur Wegweiserfunktion)
  - Nur Blätter enthalten Datensätze (oder Schlüssel und Verweise auf Datensätze)
  - Als Trennschlüssel (Separatoren, Wegweiser) nutzt man z.B. die Schlüssel selbst oder ausreichend lange Präfixe
  - Für ein effizientes Durchlaufen großer Bereiche der Daten sind die Blätter miteinander verkettet

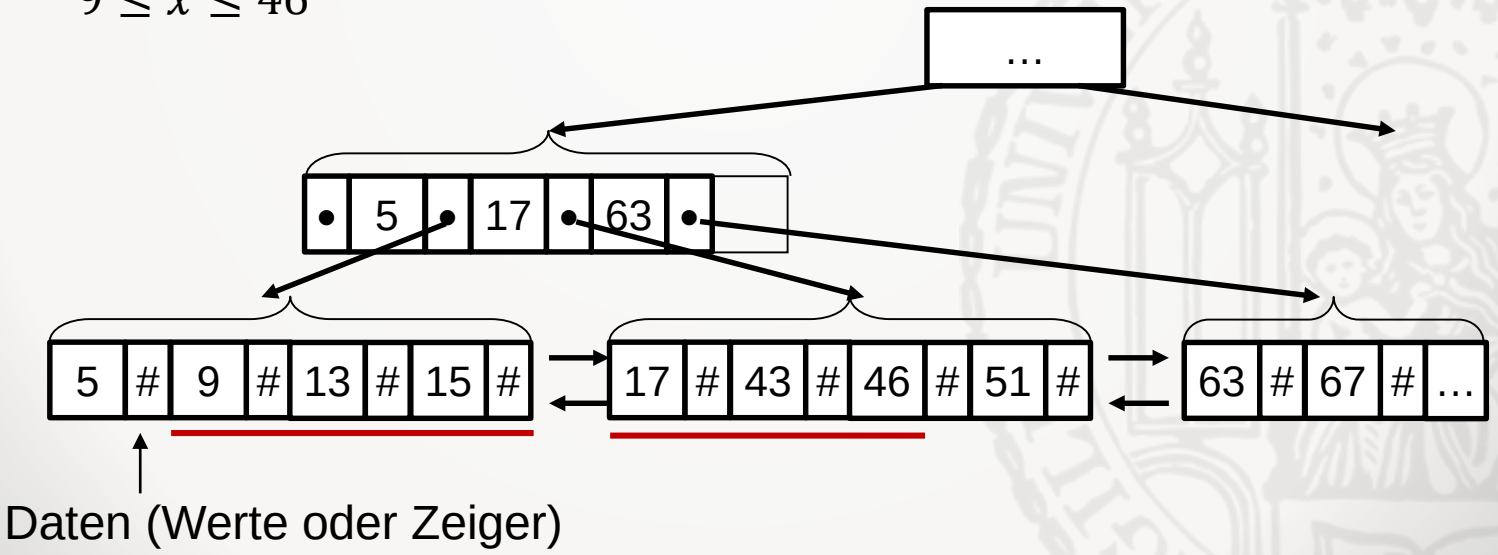


B+-Baum: Bereichsanfrage

- Neben exakten Suchanfragen müssen Datenbanken oft Bereichsanfragen (= Intervallanfragen) realisieren
  Beispiel in SQL:
```
SELECT * FROM TableOfValues
WHERE value BETWEEN ‘9‘ AND ’46‘
```

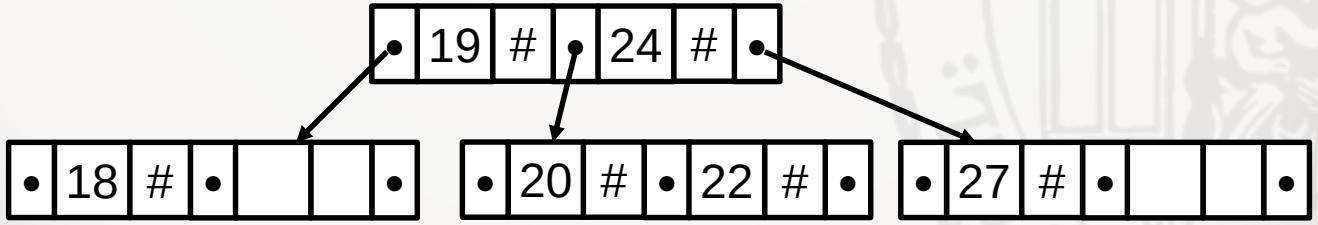
- Verkettung der Blätter ermöglicht effiziente Bereichsabfragen
  Beispiel Bereichsanfrage

$$9 \leq x \leq 46$$



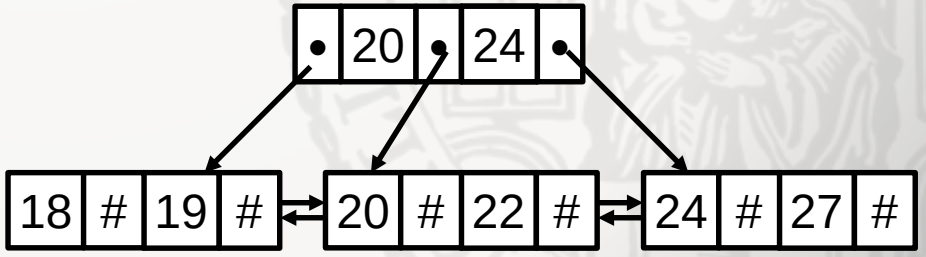
B-Baum: Vergleich Verzweigungsgrad

- Datensätze (#) stehen entweder direkt in den Knoten (ggf. bei Primärindex) oder sind ausgelagert und über Verweise referenziert (z.B. bei Sekundärindex)
- Bei der Größenabschätzung für B-Bäume sind diese Daten noch nicht berücksichtigt
- Beispiel: Seite 4096 B, Datenpointer 8 B, Knotenpointer 4 B, Schlüssel 16 Bytes
  - B-Baum: pro Knoten maximal $2k + 1$ Knotenpointer, $2k$ Datenpointer, $2k$ Schlüssel
    - Dann muss $(2k + 1) \cdot 4 + 2k \cdot 8 + 2k \cdot 16 \leq 4096 \Rightarrow 56k \leq 4092 \Rightarrow k \leq 73.1$, also k maximal 73
    - Verzweigungsgrad maximal $2k + 1 = 147$



B⁺-Baum: Vergleich Verzweigungsgrad

- Wie beim B-Baum:
  Seite 4096 B, Datenpointer 8 B, Knotenpointer 4 B, Schlüssel 16 B
- Zusätzlich: Verkettungspointer (Blätter) = Knotenpointer = 4 Bytes
- B⁺-Baum:
  - pro innerem Knoten (Directory) maximal $2k + 1$ Knotenpointer, $2k$ Schlüssel
  - pro Blatt maximal $2k$ Datenpointer, $2k$ Schlüssel, 2 Verkettungspointer
  - Innerer Knoten:
    $(2k + 1) \cdot 4 + 2k \cdot 16 \leq 4096 \Rightarrow 40k \leq 4092 \Rightarrow k \leq 102{,}3$
    Also k maximal 102, Verzweigungsgrad maximal $2k + 1 = 205$
  - Blätter:
    $2k \cdot 8 + 2k \cdot 16 + 2 \cdot 4 \leq 4096 \Rightarrow 48k \leq 4088 \Rightarrow k \leq 85.1$
    - k maximal 85
    - Füllgrad 170 maximal



B-Bäume: Zusammenfassung

- Grundidee
  - Hierarchische Strukturierung der Daten zur schnellen Suche
  - Effiziente Nutzung der Seiten des Sekundärspeichers
  - B-Bäume sind Mehrwegsuchbäume

- Unterschied zu Hashing (siehe unten)
  - dynamische Datenstruktur: kann effizient wachsen und schrumpfen
  - unterstützt Bereichsanfragen durch Ordnungserhaltung

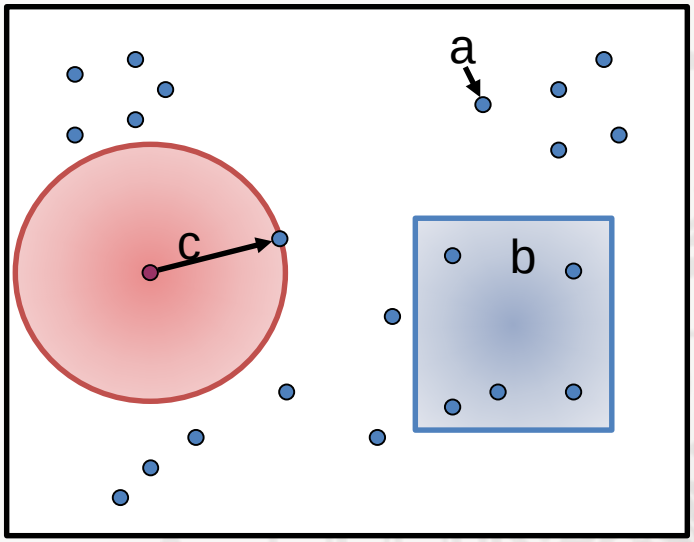
- Offenes Rätsel: Wofür steht das „B“?
  - **B**ushy Tree, **B**road Tree (Hinweis auf hohen Verzweigungsgrad)
  - **B**alanced Tree (technische Beschreibung)
  - Prof. Rudolf **B**ayer, PhD (mit Ed McCreight Erfinder der B-Bäume, 1970)
  - The **B**oeing Company (Bayer arbeitete im Forschungslabor in Seattle)
  - **B**arbara (Vorname von Bayers Ehefrau)
  - **B**anyan Tree (australischer Baum, wächst durch Wurzelteilung).
  - **B**inary Tree ? (falsch: Mehrwegebäume – richtig: binäre Suche in Knoten)

Räumliche Daten

Beispiel Punktdaten in 2D

- 2D Punktdaten können durch einen B-Baum über (x, y) indiziert werden

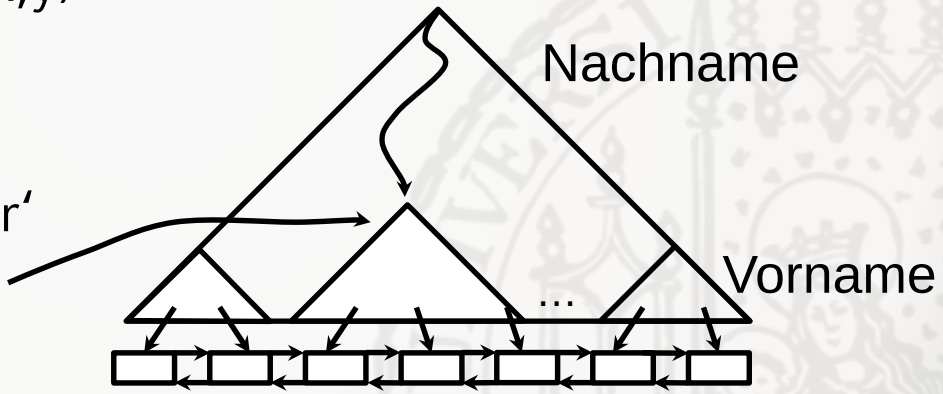
- Beispiele für Anfragen sind
  a) Punktanfragen
  b) Alle Objekte in einem bestimmten Bereich
  c) Nächster Nachbar



Zusammengesetzte Schlüssel (Composite)

Einsatzgebiet

- Bäume können auf zusammengesetzte Schlüssel erweitert werden
  - Dabei wird zuerst nach der ersten und dann nach der zweiten Komponente sortiert (lexikografische Ordnung)
  - Beispiel: Nachname, Vorname
    Punktdaten (x,y)



Anwendung

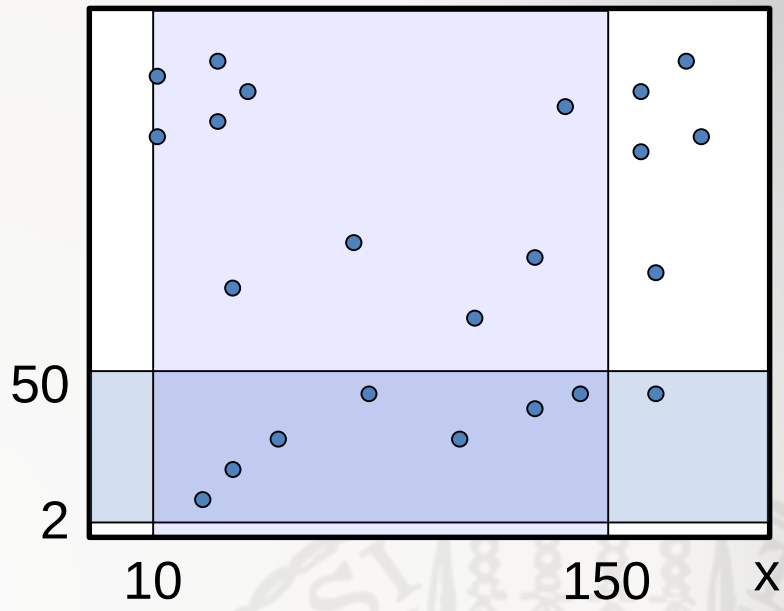
- Suche Nachname = ‚Müller‘ durchläuft alle Blätter des Teilbaumes unter ‚Müller‘
- UND Vorname = ‚Hans‘ endet in einem Blatt

Räumliche Daten

Beispiel: Bereichsanfrage

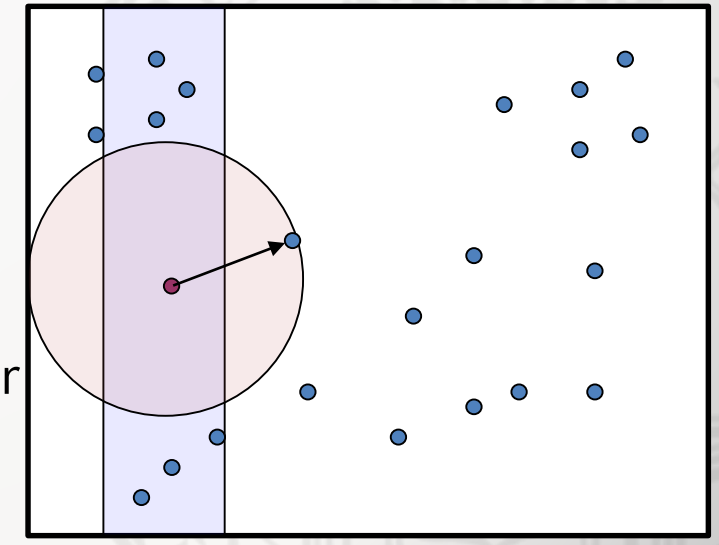
Annahme: B-Baum auf (x, y)

- $x > 10$ AND $x < 150$
- $y > 2$ AND $y < 50$
- Die Selektion nach x schränkt den Suchbereich nicht gut ein

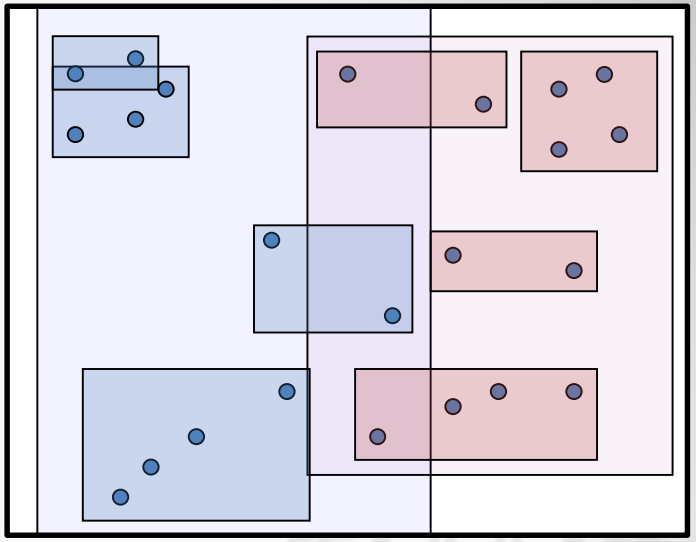


Beispiel: Nächster-Nachbarn-Anfrage (NN)

- Suche nach NN in der x-Umgebung würde nicht das gewünschte Resultat bringen
- NN bzgl. x muss nicht NN bzgl. x,y sein, d.h. schlechte Unterstützung der Suche durch Hauptsortierung nach x

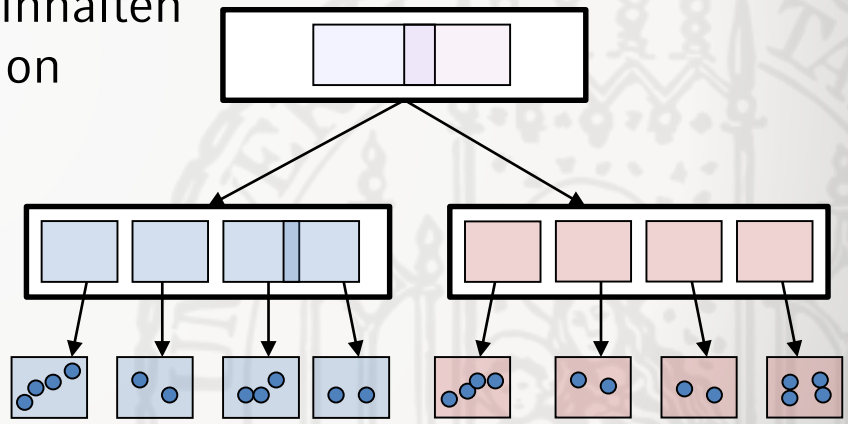


R-Baum



Struktur eines R-Baumes (Guttman, 1984)

- Zwei Knotentypen wie in B^+-Baum
  - Blattknoten enthalten Punktdaten
  - Innere Knoten enthalten Verweise auf Nachfolgerknoten sowie deren MBRs
  - MBRs (Minimal Bounding Regions): kleinste Rechtecke, welches alle Punkte im darunterliegenden Teilbaum beinhalten
  - MBRs haben also Wegweiserfunktion
  - MBRs können Punkte, aber auch geometrische Objekte enthalten



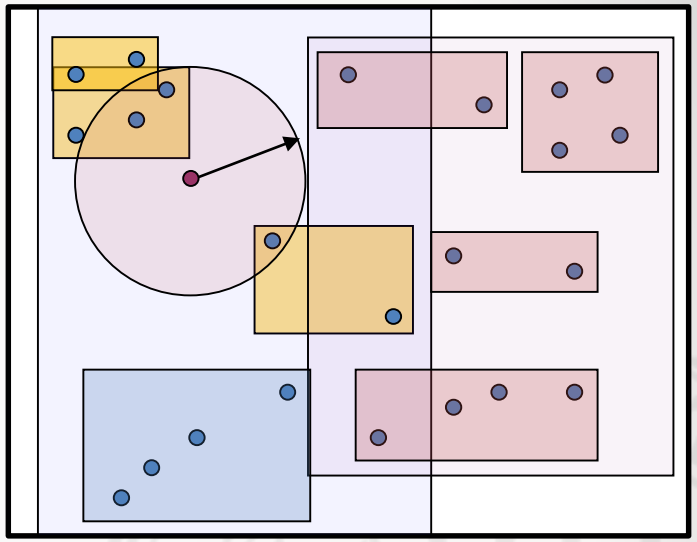
Aufbau

- Balance analog zum B-Baum
- Regionen können überlappen

- Beim Suchen ggf. Besuch mehrerer Teilbäume, auch bei Punktanfragen

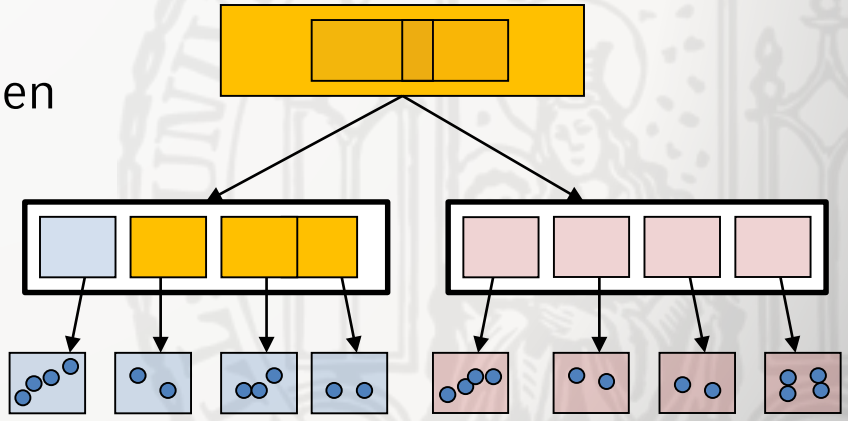
R-Baum: Bereichsanfragen

- Nicht alle MBRs müssen durchsucht werden.
  - „Pruning“ des Suchraums
- Distanz zu Rechtecken:
  - Abschätzung über minimale Distanz von Punkt zu Rechtecken.



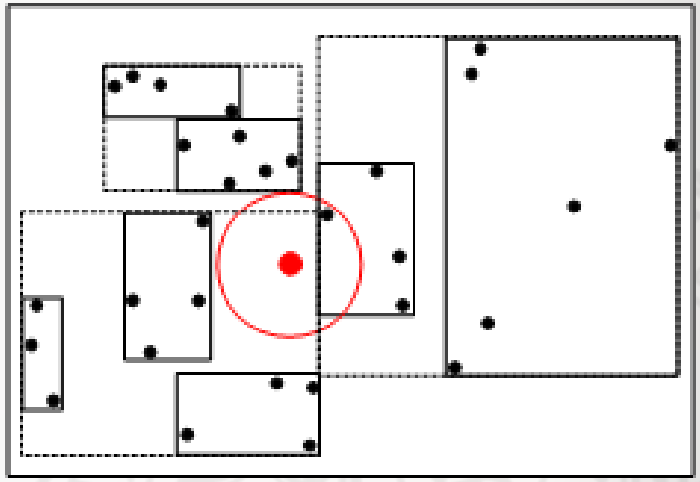
Eingabe: *Punkt p, Radius r*

1. Starten bei Wurzel
2. Tiefensuche auf Baum
3. Untersuche jeweils Kindknoten von Nichtblattknoten, deren Rechteck den Bereich um p schneidet (Bestimmung mittels Abstand)
4. Überprüfe in Blattknoten, ob
   - Abstand des Punkts p zu einem der Rechtecke ≤ Radius



R-Baum: Nächste-Nachbarn-Anfragen

- Nächste-Nachbar-Anfragen basieren auf sukzessiver Verkleinerung von Bereichsanfragen.
- Abbruch, wenn nur noch ein Element in der Bereichsanfragenumgebung.

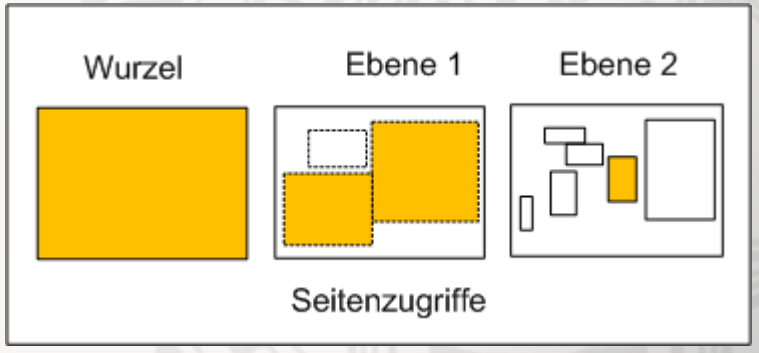


Eingabe: *Punkt p*

Initialisierung: resultdist = ∞, result = ⊥

Start mit: NN-Suche(p, Wurzel)

```
procedure NN-Suche (p:Punkt, n: Knoten)
if n ist Blattknoten then
    for i = 1 to n.Anzahl_Rechtecke do
        if dist(p, n.RE[i]) ≤ resultdist then
            result := n.RE[i];
            resultdist := dist(p,n.RE[i]);
else // n ist Directoryknoten //
    for i = 1 to n.Anzahl_Kinder do
        if dist(p, n.RE[i]) ≤ resultdist then
            NN-Suche (p,n.Kind[i]);
```



Zusammenfassung: Mehrwegbäume

- B-Bäume
  - Verbesserung der binären Bäume für die Speicherung auf Festplatten
  - Bereichsabfragen durch verkettete Blätter (B^+-Bäume)
  - B^+-Bäume sind die für den praktischen Einsatz wichtigste Variante des B-Baums

- R-Bäume
  - Mehrdimensionale Daten
  - Bereichsanfrangen, Nächster-Nachbar-Anfragen

- Hochdimensionale Daten
  - R-Bäume sind für hochdimensionale Daten ungeeignet, wegen starker Überlappung der Wegweiser
  - Dieses Problem der hohen Dimensionen tritt insbesondere beim Data Mining auf

Suche in konstanter Zeit

- Bisher: Statt lineare Suche erlauben Bäume für viele Anwendungen durch geeignete Strukturierung, den Suchaufwand auf $O(\log n)$ zu reduzieren.

- Einfügen, Löschen und Zugriff in $O(\log n)$.

- Wenn wir den Suchraum noch cleverer strukturieren, können wir dann noch schneller werden?

Bitvektor-Darstellung für Mengen

- Geeignet für kleine Universen U
  $N = |U|$ vorgegebene maximale Anzahl von Elementen
  $S \subseteq U = \{0,1,\dots,N-1\}$
  Suche hier nur als „Ist-Enthalten“-Test

  Verwende Schlüssel ‚i‘ als Index im Bitvektor (= Array von Bits)

  Bitvektor: Bit[i] = 1 wenn $i \in S$
  Bit[i] = 0 wenn $i \notin S$
- Beispiel: $N = \{0,1,2,3,4\}$, $M_1 = \{0,2,3\}$, $M_2 = \{0,1\}$

$$Bit(M_1) = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}, \qquad Bit(M_2) = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Bitvektor-Darstellung: Komplexität

- Operationen
  - Insert, Delete $O(1)$ setze/lösche entsprechendes Bit
  - Search $O(1)$ teste entsprechendes Bit
  - Initialize $O(N)$ setze ALLE Bits des Arrays auf 0

- Speicherbedarf
  - Anzahl Bits $O(N)$ maximale Anzahl Elemente

- Problem bei Bitvektor
  - Initialisierung kostet $O(N)$
  - Verbesserung durch spezielle Array-Implementierung
  - Ziel: Initialisierung $O(1)$

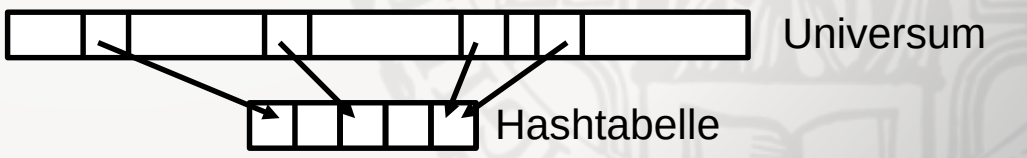
Hashing

- Ziel:
  Zeitkomplexität Suche $O(1)$ - wie bei Bitvektor-Darstellung
  Initialisierung $O(1)$

- Ausgangspunkt
  Bei Bitvektor-Darstellung wird der Schlüsselwert direkt als Index in einem Array verwendet

- Grundidee
  Oft hat man ein sehr großes Universum (z.B. Strings)
  Aber nur eine kleine Objektmenge (z.B. Straßennamen einer Stadt)
  Für die ein kleines Array ausreichend würde

- Idee
  Bilde verschiedene Schlüssel auf dieselben Indexwerte ab.
  Dadurch Kollisionen möglich



Hashing

- Grundbegriffe:
  - U ist das Universum aller Schlüssel
  - $S \subseteq U$ die Menge der zu speichernden Schlüssel mit $n = |S|$
  - T die Hash-Tabelle der Größe m

- Hashfunktion h:
  - Berechnung des Indexwertes zu einem Schlüsselwert x
  - Schlüsseltransformation: $h: U \rightarrow \{0, \ldots, m-1\}$
  - $h(x)$ ist der Hash-Wert von x

- Hashing wird angewendet wenn:
  - $|U|$ sehr groß ist
  - $|S| \ll |U|$ - Anzahl der zu speichernden Elemente ist viel kleiner als die Größe des Universums

Anwendung von Hashing als Prüfziffer

- IBAN (International Bank Account Number):
  Aufbau einer deutschen IBAN

D	E	x	x	b	b	b	b	b	b	b	b	k	k	k	k	k	k	k	k	k	k
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

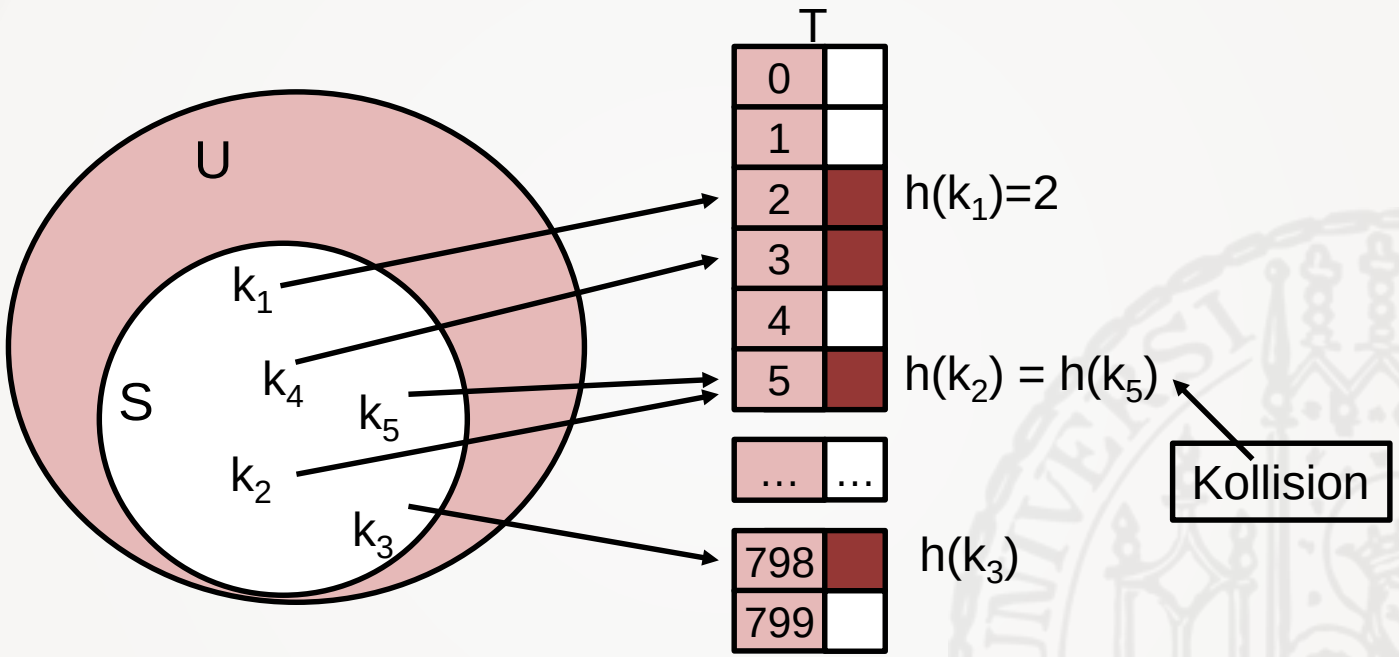
Land | Prüf-ziffern | Bankleitzahl | Kontonummer

Ländercode beachten:
$A \to 10, B \to 11, \ldots$

- Berechnung der Prüfziffer:
  $$\text{xx} = 98 - bbbbbbbbkkkkkkkkkk131400 \; mod \; 97$$
- Universum: 10^{18} Bank-/Kontonummern (theoretisch) möglich
- Hashwerte: $02 \leq \text{xx} \leq 98$
- Durch die schnelle Berechnung können viele Fehler bereits bei Eingabe gemeldet werden (z.B. Zahlendreher)

Hashing-Prinzip

- Grafische Darstellung - Beispiel: Studenten



- Gesucht:
  - Hashfunktion, welche die Matrikelnummern möglichst gleichmäßig auf die 800 Einträge der Hash-Tabelle abbildet

Hashfunktion

- Dient zur Abbildung auf eine Hash-Tabelle
  - Hash-Tabelle **T** hat m Plätze (Slots, Buckets)
    - In der Regel $m \ll |U|$ daher Kollisionen möglich
  - Speichern von $|S| = n$ Elementen ($n < m$)
  - Belegungsfaktor $\alpha = n/m$

- Anforderung an eine Hashfunktion
  - $h: domain(K) \rightarrow \{0, 1, \ldots, m-1\}$ soll surjektiv sein.
  - $h(K)$ soll effizient berechenbar sein, idealerweise in $O(1)$.
  - $h(K)$ soll die Schlüssel möglichst gleichmäßig über den Adressraum verteilen, um dadurch Kollisionen zu vermeiden (Hashing = Streuspeicherung).
  - $h(K)$ soll unabhängig von der Ordnung der K sein in dem Sinne, dass in der Domain „nahe beieinander liegende“ Schlüssel auf nicht nahe beieinander liegende Adressen abgebildet werden.

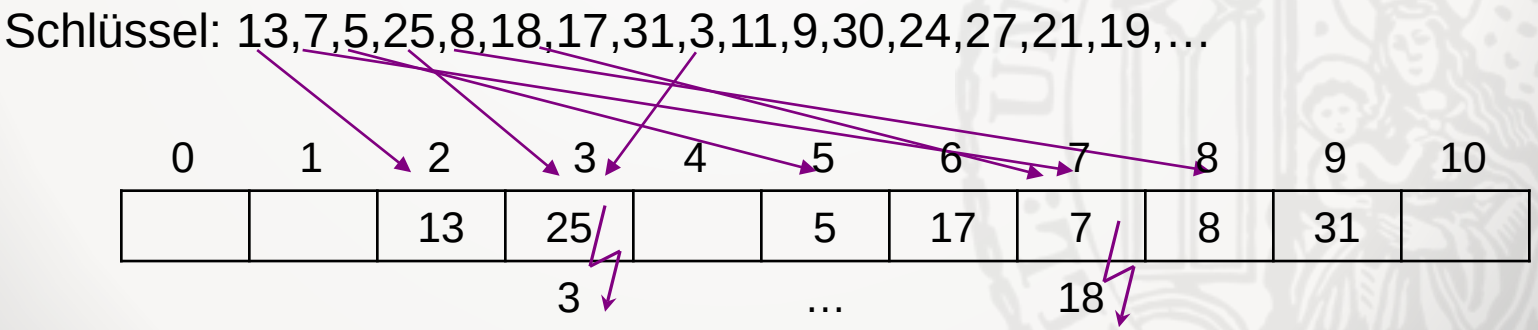
Hashfunktion: Divisionsmethode

- Hashfunktion:
  - $h(k) = K \; mod \; m$ für numerische Schlüssel
  - $h(k) = ord(K) \; mod \; m$ für nicht-numerische Schlüssel

- Konkretes Beispiel für ganzzahlige Schlüssel:

  $h: domain(K) \rightarrow \{0,1, \ldots, m-1\}$ mit $h(K) = K \; mod \; m$

- Sei m=11



Beispiel: Divisionsmethode

- Für Zeichenketten: Benutze die *ord*-Funktion zur Abbildung auf ganzzahlige Werte, z.B.
- Sei m=17:

$$h : \text{STRING} \mapsto \left(\sum_{i=1}^{\text{len(STRING)}} ord(\text{STRING}[i]) \right) \bmod m$$

JAN	→ 25 mod 17 = 8	MAI	→ 23 mod 17 = 6	SEP	→ 40 mod 17 = 6
FEB	→ 13 mod 17 = 13	JUN	→ 45 mod 17 = 11	OKT	→ 46 mod 17 = 12
MAR	→ 32 mod 17 = 15	JUL	→ 43 mod 17 = 9	NOV	→ 51 mod 17 = 0
APR	→ 35 mod 17 = 1	AUG	→ 29 mod 17 = 12	DEZ	→ 35 mod 17 = 1

- Wie sollte m aussehen?
  - $m = 2^d$ → einfach zu berechnen
    K *mod* 2^d liefert die letzten d Bits der Binärzahl K → Widerspruch zur Unabhängigkeit von K
  - m gerade → $h(K)$ gerade ⇔ K gerade → Widerspruch zur Unabhängigkeit von K
  - m Primzahl → hat sich erfahrungsgemäß bewährt

Beispiel Hashfunktion

- Einsortieren der Monatsnamen in die Symboltabelle

$$h(c) = \left(N(c_1) + N(c_2) + N(c_3)\right) \bmod 17$$

0	November
1	April, Dezember
2	März
3	
4	
5	
6	Mai, September
7	
8	Januar

9	Juli
10	
11	Juni
12	August, Oktober
13	Februar
14	
15	
16	

3 Kollisionen

Perfekte Hashfunktion

- Eine Hashfunktion ist perfekt:
  - wenn für $h : U \to \{0, \dots, m-1\}$ mit $S = \{k_1, \dots, k_n\} \subseteq U$ gilt
    $$h(k_i) = h(k_j) \Leftrightarrow i = j$$
  - also für die Menge S keine Kollisionen auftreten
- Eine Hashfunktion ist minimal:
  - wenn $m = n$ ist, also nur genau so viele Plätze wie Elemente benötigt werden
- Im Allgemeinen können perfekte Hashfunktionen nur ermittelt werden, wenn alle einzufügenden Elemente und deren Anzahl (also S) im Voraus bekannt sind (static Dictionary)
  → In der Praxis meist nicht gegeben!

Kollisionen beim Hashing

- Verteilungsverhalten von Hashfunktionen
  - Untersuchung mit Hilfe von Wahrscheinlichkeitsrechnung
  - S sei ein Ereignisraum
  - E ein Ereignis $E \subseteq S$
  - P sei eine Wahrscheinlichkeitsverteilung

- Beispiel: Gleichverteilung
  - einfache Münzwürfe: $S = \{\text{Kopf}, \text{Zahl}\}$
  - Wahrscheinlichkeit für Kopf $P(\text{Kopf}) = \frac{1}{2}$

  - n faire Münzwürfe: $S = \{\text{Kopf}, \text{Zahl}\}^n$
  - Wahrscheinlichkeit für n-mal Kopf $P(n\text{−mal Kopf}) = \left(\frac{1}{2}\right)^n$
    (Produkt der einzelnen Wahrscheinlichkeiten)

Kollisionen beim Hashing

- Analogie zum Geburtstagsproblem (-paradoxon)
  - Wie groß ist die Wahrscheinlichkeit, dass mindestens 2 von n Leuten am gleichen Tag Geburtstag haben?
  - $m = 365$ Größe der Hash-Tabelle (Tage), n = Anzahl Personen

- Eintragen des Geburtstages in die Hash-Tabelle
  - $p(i, m)$ = Wahrscheinlichkeit, dass für das i-te Element eine Kollision auftritt
  - $p(1, m) = 0$ da keine Zelle belegt
  - $p(2, m) = 1/m$ da 1 Zellen belegt
  - ...
  - $p(i, m) = (i - 1)/m$ da (i-1) Zellen belegt

Kollisionen beim Hashing

- Eintragen des Geburtstages in die Hash-Tabelle
  - Wahrscheinlichkeit für keine einzige Kollision bei n Einträgen in eine Hash-Tabelle mit m Plätzen ist das Produkt der einzelnen Wahrscheinlichkeiten

$$P(NoCol|n,m) = \prod_{i=1}^{n}\left(1 - p(i,m)\right) = \prod_{i=0}^{n-1}\left(1 - \frac{i}{m}\right)$$

  - Die Wahrscheinlichkeit, dass es mindestens zu einer Kollision kommt, ist somit

$$P(Col|n,m) = 1 - P(NoCol|n,m)$$

Kollisionen beim Hashing

- Kollisionen bei Geburtstagstabelle

Anzahl Personen n	$P(Col\|n,m)$
10	0,11695
20	0,41144
…	
22	0,47570
23	0,50730
24	0,53835
…	
30	0,70632
40	0,89123
50	0,97037

- Schon bei einer Belegung von $23/365 = 6\%$ kommt es zu 50% zu mindestens einer Kollision
- Daher Strategie für Kollisionen wichtig
- Wann ist eine Hashfunktion gut?
- Wie groß muss eine Hash-Tabelle in Abhängigkeit zu der Anzahl Elemente sein?

Kollisionen beim Hashing

- Wie muss m in Abhängigkeit zu n wachsen, damit $P(NoCol|n,m)$ konstant bleibt?

$$P(NoCol|n,m) = \prod_{i=0}^{n-1}\left(1-\frac{i}{m}\right)$$

- Durch Anwendung der Logarithmus-Rechenregel kann ein Produkt in eine Summe umgewandelt werden:

$$ab = e^{\ln(ab)} = e^{\ln a + \ln b}$$

$$P(NoCol|n,m) = \exp\left(\sum_{i=0}^{n-1}\ln\left(1-\frac{i}{m}\right)\right)$$

- Logarithmus: $\ln(1-\varepsilon) \approx -\varepsilon$
- Da $n \ll m$ gilt: $\ln\left(1-\frac{i}{m}\right) \approx -\left(\frac{i}{m}\right)$

Kollisionen beim Hashing

- Auflösen der Gleichung

$$P(NoCol|n,m) \approx \exp\left(-\sum_{i=0}^{n-1} \ln\left(\frac{i}{m}\right)\right) = \exp\left(-\frac{n(n-1)}{2m}\right) \approx \exp\left(-\frac{n^2}{2m}\right)$$

- Ergebnis:
  Kollisionswahrscheinlichkeit bleibt konstant wenn
  m (Größe der Hash-Tabelle) quadratisch mit n (Zahl der Elemente) wächst

Hashing: Umgang mit Kollisionen

- Kollisionen treten auf, wenn zwei Schlüssel den selben Hashwert erhalten und an die gleiche Stelle gespeichert werden müssen.

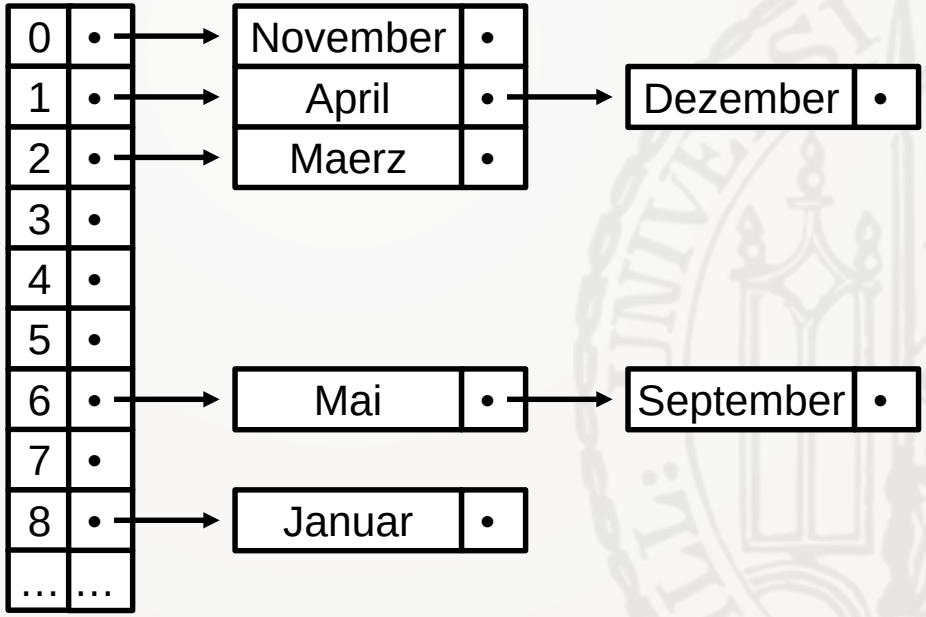
- Kollisionen sind lassen sich nicht vermeiden, deswegen gibt es entsprechende Methoden zur Behandlung.

- Tritt eine Kollision auf, so gibt es zwei populäre Auflösungsstrategien:
  - **Offenes Hashing mit geschlossener Adressierung**
  - **Geschlossenes Hashing mit offener Adressierung**

    Achtung: In der Literatur gerne als Offenes/Geschlossenes Hashing abgekürzt und dann teils vertauscht benutzt!

Offenes Hashing mit geschlossener Adressierung

- Speicherung der Schlüssel außerhalb der Tabelle, z.B. als verkettete Liste.
- Bei Kollisionen werden Elemente unter der selben Adresse abgelegt.
- Die externe Speicherstruktur hat großen Einfluss auf Effektivität und Effizienz.



Geschlossenes Hashing mit offener Adressierung

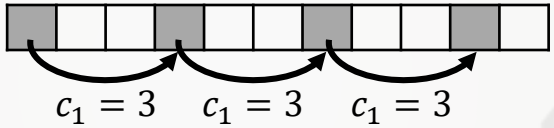
- Bei Kollision wird mittels bestimmter Sondierungsverfahren eine freie Adresse gesucht.
- Jede Adresse der Hashtabelle nimmt höchstens einen Schlüssel auf.
- Das Sondierungsverfahren bestimmt die Effizienz, so dass nur wenige Sondierungsschritte nötig sind.

0	November
1	April
2	Maerz
3	Dezember
4	
5	
6	Mai
7	September
8	Januar
...	...

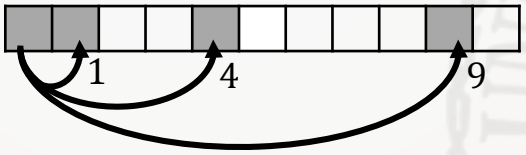
Polynomielles Sondieren

- Für $j = 0 \ldots m$ teste Hashadresse
$$h(x, j) = (h(x) + c_1 j + c_2 j^2 + \cdots) \; mod \; m$$
bis eine freie Adresse gefunden wird.

- Für $h(x, j) = (h(x) + c_1 j) \; mod \; m$ sprechen wir von linearem Sondieren.



- Für $h(x, j) = (h(x) + c_2 j^2) \; mod \; m$ sprechen wir von quadratischem Sondieren.



- Problem: Clusterbildung, für viele gleiche Schlüssel werden die gleichen Positionen sondiert.

Geschlossenes Hashing: Komplexität

- Hier: Anzahl Sondierungsschritte
  - Einfügen: $C_{Ins}(n,m)$
  - Erfolglose Suche: $C^{-}_{search}(n,m)$
  - Erfolgreiche Suche: $C^{+}_{search}(n,m)$
  - Löschen: $C_{Del}(n,m)$
  - m: Größe der Hash-Tabelle
  - n: Anzahl der Einträge
  - $\alpha = \frac{n}{m}$: Belegungsfaktor der Hash-Tabelle

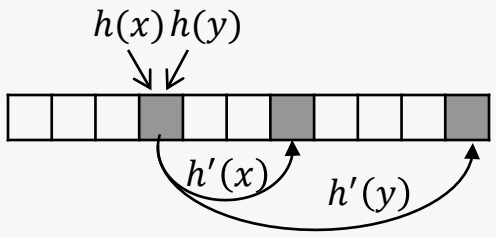
Belegung α	$C^{-}_{search}(n,m) = C_{Ins}(n,m) \approx \frac{1}{1-\alpha}$	$C^{+}_{search}(n,m) = C_{Del}(n,m) \approx \frac{1}{\alpha}\ln\frac{1}{1-\alpha}$
0,5	≈ 2	≈ 1,38
0,7	≈ 3,3	≈ 1,72
0,9	≈ 10	≈ 2,55
0,95	≈ 20	≈ 3,15

min. n=19 und m=20 damit α=0,95 (bei ganzen Zahlen)

Doppelhashing

- Doppelhashing soll Clusterbildung verhindern, dafür werden zwei unabhängige Hashfunktionen verwendet.

- Dabei heißen zwei Hashfunktionen h und h' unabhängig, wenn gilt
  - Kollisionswahrscheinlichkeit $P\big(h(x) = h(y)\big) = \frac{1}{m}$
  - $P\big(h'(x) = h'(y)\big) = \frac{1}{m}$
  - $P\big(h(x) = h(y) \wedge h'(x) = h'(y)\big) = \frac{1}{m^2}$
- Sondierung mit $h(x, j) = (h(x) + h'(x) \cdot j^2)\ mod\ m$
- Nahezu ideales Verhalten aufgrund der unabhängigen Hashfunktionen



Hashing: Suchen nach Löschen

- Offenes Hashing: Behälter suchen und Element aus Liste entfernen → kein Problem bei nachfolgender Suche
- Geschlossenes Hashing:
  - Entsprechenden Behälter suchen
  - Element entfernen und Zelle als gelöscht markieren
    - Notwendig da evtl. bereits *hinter* dem gelöschten Element andere Elemente durch Sondieren eingefügt wurden
      (In diesem Fall muss beim Suchen über den freien Behälter hinweg sondiert werden)
  - Gelöschte Elemente dürfen wieder überschrieben werden

Hashing: Zusammenfassung

- Anwendung:
  - Postleitzahlen (Statische Dictionaries)
  - IP-Adresse zu MAC-Adresse (i.d.R. im Hauptspeicher)
  - Datenbanken (Hash-Join)

- Vorteil
  - Im *Average Case* sehr effizient ($O(1)$)

- Nachteil
  - Skalierung: Größe der Hash-Tabelle muss vorher bekannt sein
    - Abhilfe: Spiral Hashing, lineares Hashing
  - Keine Bereichs- oder Ähnlichkeitsanfragen
    - Lösung: Suchbäume

Suchen: Zusammenfassung

Hashing

- Extrem schneller Zugriff für Spezialanwendungen
  - Bestimmung einer Hashfunktion für die Anwendung
  - Beispiel: Symboltabelle im Compilerbau, Hash-Join in Datenbanken

Binärer Bäum (AVL-Baum, Splay-Baum)

- Allgemeines effizientes Verfahren für Indexverwaltung im Hauptspeicher
  - Bereichsanfragen möglich, da explizit ordnungserhaltend
  - Bei Updates effizienter als sortierte Arrays

B-Baum, B^+-Baum, R-Baum, etc.

- Effiziente Implementierung für die Verwendung von blockorientierten Sekundärspeichern
- B^+-Bäume werden in nahezu allen Datenbanksystemen eingesetzt