Kapitel 3 - Suchen in Mengen

Einfache Suchverfahren

Lineare Suche

Binäre Suche

Interpolationssuche

Hashing

Suchbäume

Binärer Suchbaum

AVL-Baum

Splay-Baum

B-Baum

R-Baum

Problemstellung

- Gegeben: Eine Menge von Elementen (Records)
  Eindeutig identifiziert durch einen Schlüssel
  In der Regel werden Duplikate ausgeschlossen
- Beispiel: Menge von Personen (AusweisNr, Name, Adresse..)
  AusweisNr ist eindeutig
- Aufgabe: Finde zu einem gegebenen Schlüsselwert das entsprechende Element
- Notation: Universum – U := Menge aller möglichen Schlüssel
  Menge – $S \subseteq U$

Einführung

- Beispiele für Universen

  Symboltabelle Compiler - nur 6 Zeichen (Buchstaben und Zahlen)

  $|U| = (26 + 10)^6$

  Konten einer Bank - 6-stellige Konto-Nummer

  $|U| = 10^6$

- Typische Aufgaben

  Finde zu einem gegebenen Schlüssel das entsprechende Element (Record) und führe eine Operation aus

  Im Prinzip werden alle Mengenoperationen betrachtet (Datenbanken)

Operation Suche

- Spezifikation

  Suchalgorithmus nutzt Vergleichsmethode um gesuchtes Element zu finden

Menge	$S \subseteq U$	mit ‚n' Elementen
Element	S[i]	mit $1 \leq i \leq n$
Sortiert	$S[i] < S[i+1]$	für $1 \leq i \leq n-1$
Gesucht	a bzw. i	mit $S[i] = a$

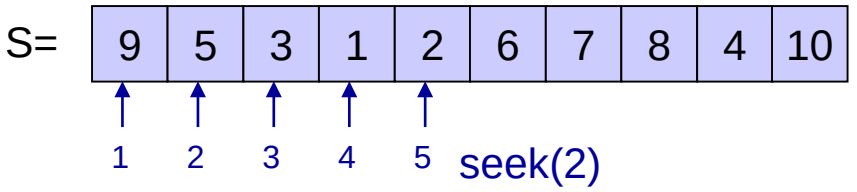
- Beobachtungen zur Effizienz
  - Im Allgemeinen dauert die Suche nach einem Element, welches nicht in ‚S' enthalten ist, länger als die Suche nach enthaltenen Elementen.
    - Erfolgreiche Suche: kann oft frühzeitig abgebrochen werden (wenn das Element gefunden wurde)
    - Nicht erfolgreiche Suche: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

  Lineare Suche (auch ohne Ordnung möglich)

  Binärsuche

  Interpolationssuche

Lineare Suche



- Lineare Suche durchläuft S sequenziell
- Wird auch als sequenzielle Suche bezeichnet
- S kann ungeordnet sein

```
public static int seek (int a, int S[]) throws Exception  {

    for ( int i=0; i<S.length; i++ )  {
        if ( S[i]==a )
            return i;
    }
    throw new Exception ("seek failed");

}
```

Komplexität: Lineare Suche

- Hier: Anzahl Schlüsselvergleiche

Anzahl Vergleiche bei linearer Suche zwischen 1 und n

– Im Mittel: n/2 Vergleiche (wenn erfolgreich)

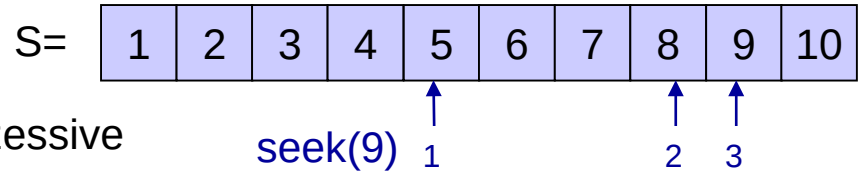
Genauer: Wenn Liste unsortiert (im Mittel)
– Erfolgreich = n/2 Vergleiche
– Erfolglos = n Vergleiche

Wenn Liste sortiert (im Mittel)
– Erfolgreich = n/2 Vergleiche
– Erfolglos = n/2 Vergleiche

=> Komplexität O(n)

Binäre Suche

- Voraussetzung: S ist sortiert



- Binäre Suche halbiert den Suchbereich sukzessive

```
public static int seek (int a, int S[]) throws Exception {
        int low=0;   int high=S.length-1;
        while ( low ≤ high )  {
              int mid = (high+low) / 2;
              if ( a==S[mid]        return mid;
             else if ( a<S[mid]  high = mid -1;
              else /* a>S[mid]    low = mid +1;
         }
         throw new Exception (“seek failed”)
 }
```

Anwendungsbeispiel binäre Suche: Suche nach Teilzeichenketten

- Suche nach Teilzeichenketten (Muster) in einem Text oder DNA-Sequenz (**S**=$s_1 s_2 \ldots s_n$)
  - Beispiele für die Zeichenkette: **S**= "aachen"
    - Beliebige Teilzeichenkette: subString(**S**,3,5) = "che"
    - Endstück einer Zeichenkette: subString(**S**,3,n) = "chen"

- Problem: Suche nach einem Muster (z.B. **P**="che")
  - **P** ist enthalten in **S** mit: **P**=subString(**S**,i,j)
    - Beispiel: **P**=subString(**S**,3,5)

- Lösungsansatz: SuffixArray Hauptspeicherstruktur für die Suche nach Teilzeichenketten

SuffixArray

S =

M	I	S	S	I	S	S	i	P	P	I	$
1	2	3	4	5	6	7	8	9	10	11	12

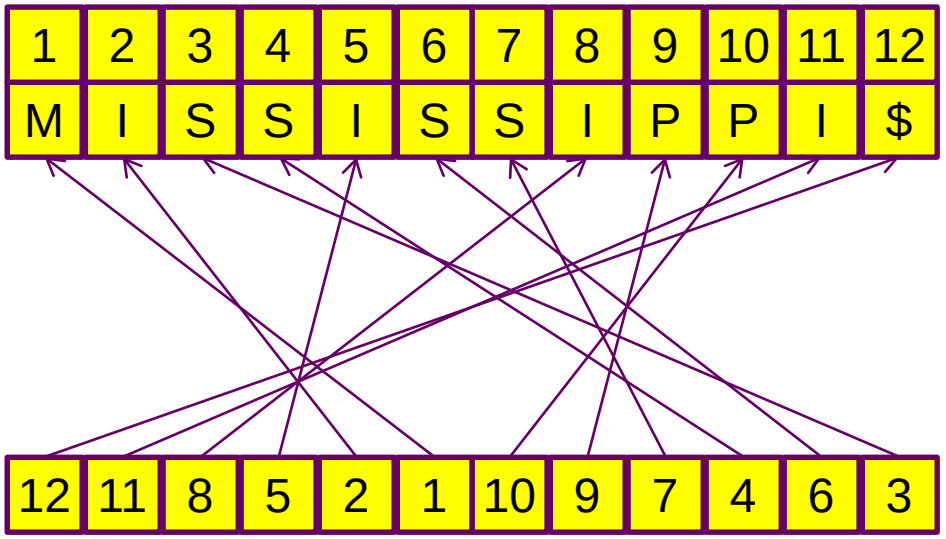
SA =

12	11	8	5	2	1	10	9	7	4	6	3
$	I $	I P P I $	I S S I P P I $	I S S I S S I P P I $	M I S S I S S I P P 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 S I P P I $

- SA Sei $S \in \sum^*$ mit n = |S| und $\$ \notin \sum$ mit der Ordnung $\$ < s_i$ für alle $s_i \in \sum^*$. SA ist ein Array der Länge n+1. Für alle SA[i] = j, $1 \le i \le n$ gilt:
  j ist ein Verweis auf subString(S$,j,n) und SA[i] ist lexikographisch kleiner als SA[i+1].
- SA ist sortiert und erfüllt die Voraussetzungen der binären Suche.

Speicherung eines SuffixArray

- In manchen Programmiersprachen (z.B. in C) lassen sich Zeiger auf das ursprüngliche Array anlegen.
  - Der Speicheraufwand ist O(n)
  - Array
  - Suffixarray



SuffixArray (Teilzeichenkettensuche)

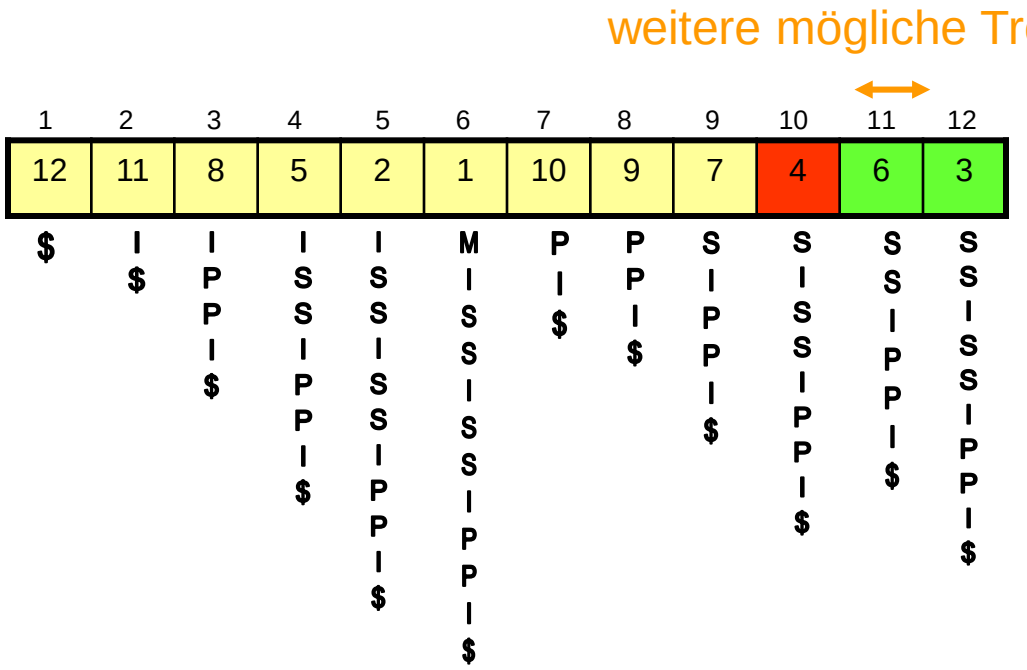
1	2	3	4	5	6	7	8	9	10	11	12
12	11	8	5	2	1	10	9	7	4	6	3
$	I$	IPPI$	ISSIPPI$	ISSISSIPPI$	MISSISSIPPI$	PI$	PPI$	SIPPI$	SISSIPPI$	SSIPPI$	SSISSIPPI$

- Fragestellung 1:
  - ist Muster **P** in **S** enthalten?
  - Beispiel:
    - **P** = "SSI"
  - binäre Suche in SA

P = SSI

SA[11] = SSIPPI$

SuffixArray (Teilzeichenkettensuche)



- Fragestellung 2:
  - an welchen Positionen kommt **P** in **S** vor?

    3 und **6**

Komplexität: Binäre Suche & SuffixArray

- Binäre Suche: Sukzessives halbieren

  Voraussetzung: Liste ist sortiert

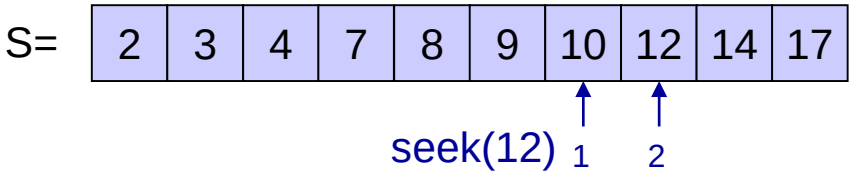
  Bei jedem Schleifendurchlauf halbiert sich der durchsuchte Bereich

  Rekursionsgleichung: $T(n) = T(n/2) + 1$

  Komplexität: $O(\log(n))$ (auch im Worst Case)

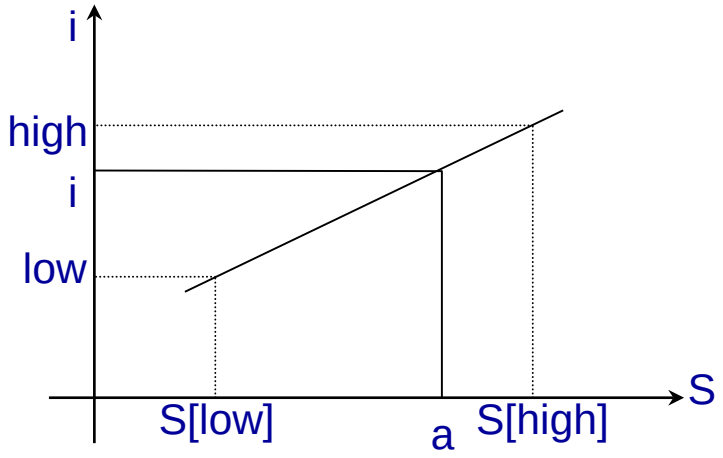
- SuffixArray: Laufzeitanalyse für die Suche „*Ist Pattern P in String S enthalten?*“
  - $O(\log(|S|))$ Schritte für die Suche (binäre Suche)
  - $O(|P|)$ Zeichenvergleiche in jedem dieser Schritt
  - ergibt $O(|P| \log(|S|))$ Zeichenvergleiche insgesamt mit $O(|S|)$ Speicher

Interpolationssuche



- S ist sortiert
- Annahme Gleichverteilung der Elemente
- Schätze gesuchte Position durch lineare Interpolation

```
public static int seek (int a, int S[]) throws Exception {
        int low=0;   int high=S.length-1;
        while ( low ≤ high )  {
            int i = low + (a-S[low])*(high-low) / (S[high]-S[low]);
            if ( a==S[i]          return i;
            else if ( a<S[i] )   high = i-1;
            else /* a>S[i]  */   low = i+1;
        }
        throw new Exception (“seek failed”);
}
```



Komplexität: Interpolationssuche

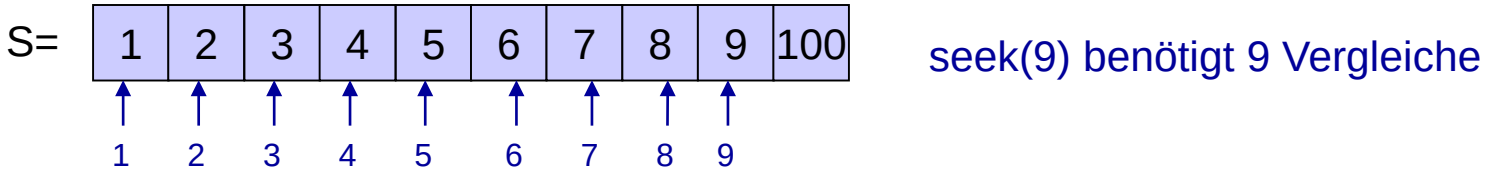
Voraussetzung: Liste ist sortiert

Komplexität im Durchschnitt: O(log(log(n)) (ohne Beweis)

Komplexität im Worst Case: O(n)

Worst-Case-Betrachtung bei Interpolations-Suche

(Abhilfe: Kombination mit Binärer-Suche)



seek(9) benötigt 9 Vergleiche

Durchschnittliche Komplexität: Vergleich der Suchverfahren

- Beispiel n = 1.000
  - Sequenziell: 500 Vergleiche
  - Binäre Suche: 10 Vergleiche ($=\lceil ld(1.000) \rceil$)
  - Interpolationssuche: 4 Vergleiche ($= \lceil ld(ld(1.000)) \rceil$)

- Beispiel n = 1.000.000
  - Sequenziell: 500.000 Vergleiche
  - Binäre Suche: 20 Vergleiche ($= \lceil ld(1.000.000) \rceil$)
  - Interpolationssuche: 5 Vergleiche ($= \lceil ld(ld(1.000.000)) \rceil$)

Zusammenfassung: Einfache Suchverfahren

Methode	Suche	Platz	Vorteil	Nachteil
Lineare Suche	n	n	Keine Initialisierung	Hohe Suchkosten
Binäre Suche	log(n)	n	Worst Case auch log(n)	Sortiertes Array
Interpolationssuche	log(log(n))	n	Schnelle Suche	Sortiertes Array Worst Case O(n)

=> Gewünscht: O(1) Suchkomplexität und O(1) Initialisierung

Kapitel 3 - Suchen in Mengen

Einfache Suchverfahren

Lineare Suche

Binäre Suche

Interpolationssuche

Hashing

Suchbäume

Binärer Suchbaum

AVL-Baum

Splay-Baum

B-Baum

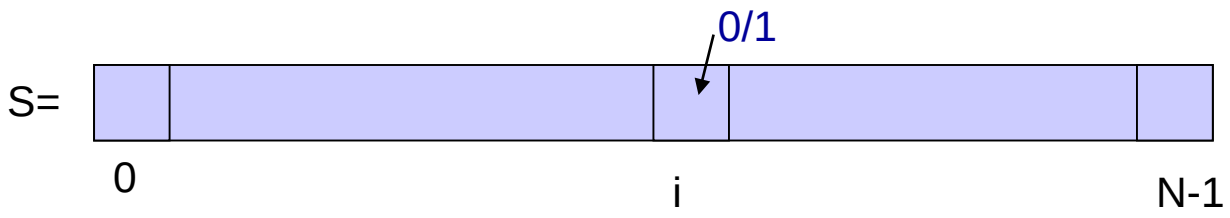
R-Baum

Bitvektor-Darstellung für Mengen

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

- Darstellung als Bitvektor
  Verwende Schlüssel ‚i‘ als Index im Bitvektor (= Array von Bits)
  - boolean isElement(Bit[], i) { return Bit[i]; }

  Bitvektor: Bit[i] = 0 wenn $i \in S$
  Bit[i] = 1 wenn $i \notin S$



Komplexität: Bitvektor-Darstellung

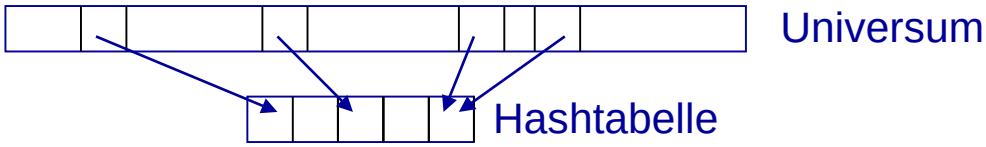
- Operationen
  - Insert, Delete — O(1) setze/lösche entsprechendes Bit
  - Search — O(1) teste entsprechendes Bit
  - Initialize — O(N) setze ALLE Bits des Arrays auf 0
- Speicherbedarf
  - Anzahl Bits — O(N) maximale Anzahl Elemente
- Problem bei Bitvektor
  - Initialisierung kostet O(N)
  - Verbesserung 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 *K*
  - Definition $h : U \rightarrow \{0,\ldots,m-1\}$ – Schlüsseltransformation
  - h(x) ist der Hash-Wert von x

- Anwendung:
  - Hashing wird angewendet wenn:
    - |U| sehr groß ist
    - |S| << |U| - Anzahl zu speichernde Elemente ist viel kleiner als die Größe des Universums

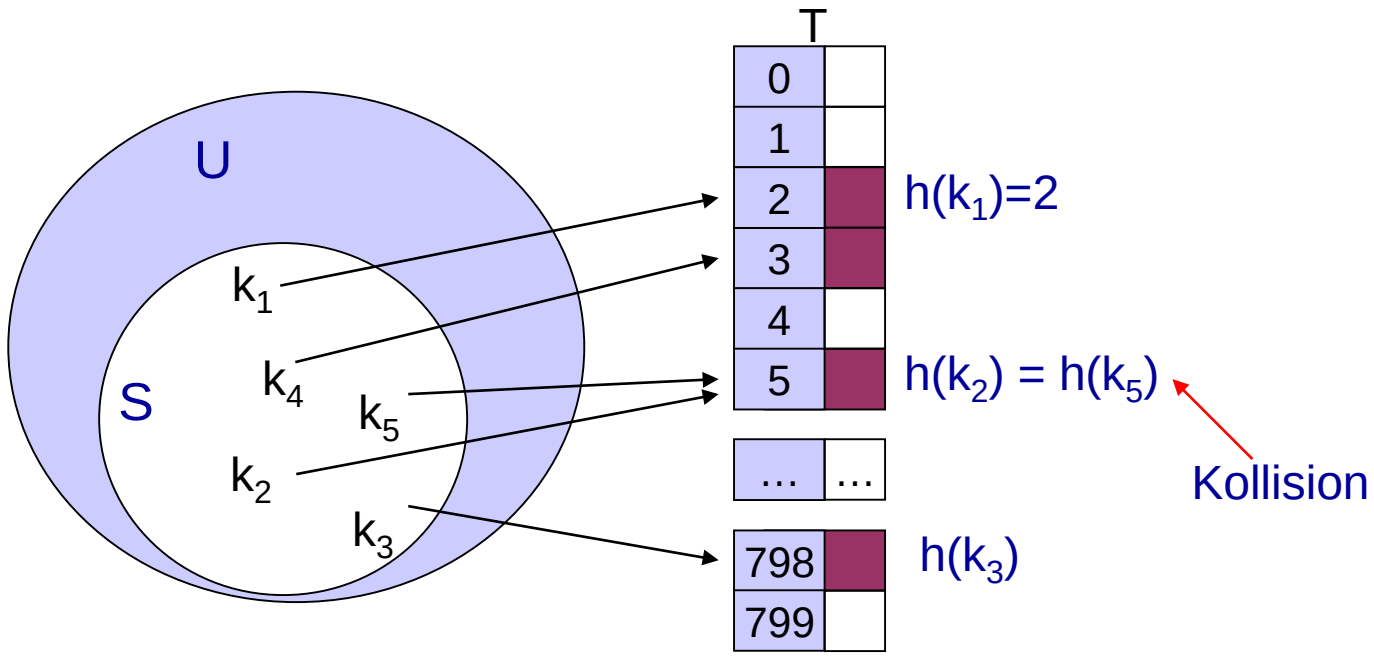
Anwendung von Hashing

- Beispiel: Symboltabelle Compiler
  - Universum U: Alle möglichen Zeichenketten
  - Eingeschränkt auf Länge 20 (nur Buchstaben und Ziffern)
  - $|U| = (26+10)^{20} = 1.3 * 10^{31}$
  - Somit keine umkehrbare Speicher-Funktion realistisch
  - Es werden nur *m*-Symbole ($m<<10^{31}$) von einem Programm verwendet (also müssen auch nur *m*-Symbole vom Compiler berücksichtigt werden)

- Beispiel: Studenten
  - Universum U: Alle möglichen Matrikelnummern (6-Stellig; d.h. $|U| = 10^{6)}$
  - Nur n Studenten besuchen eine Vorlesung (z.B. n=500) Schlüsselmenge $S=\{k_1,\ldots k_n\} \subseteq U$
  - Verwendung einer Hash-Tabelle T mit m=|T|=800

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* << |U| daher Kollisionen möglich
  - Speichern von |S| = *n* Elementen (n<m)
  - Belegungsfaktor α = n/m

- Anforderung an eine Hashfunktion
  - *h: domain(K)* $\rightarrow$ { 0, 1 , …, *m* -1 } soll surjektiv sein.
  - *h(K)* soll effizient berechenbar sein (idealerweise in O(1)).
  - *h* 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

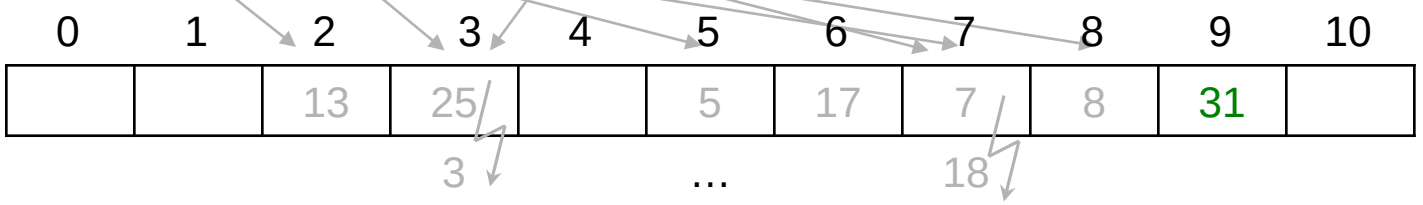
- Hashfunktion:
  - *h(k) = K mod m* für numerische Schlüssel
  - *h(k) = ord(K) mod m* für nicht-numerische Schlüssel

- Konkretes Beispiel für ganzzahlige Schlüssel:

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

- Sei m=11 :

  Schlüssel: 13,7,5,25,8,18,17,31,3,11,9,30,24,27,21,19,…



Beispiel: Divisionsmethode

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

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

  - Sei m=17:

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 $h : U \rightarrow \{0,\ldots,m-1\}$ mit $S=\{k_1,\ldots,k_n\} \subseteq U$ 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)

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 einer Hash-Tabelle mit m Plätzen ist das Produkte der einzelnen Wahrscheinlichkeiten

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

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

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

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 (Demonstration)
- Daher Strategie für Kollisionen wichtig
- Fragen:
  - Wann ist eine Hashfunktion gut?
  - Wie groß muss eine Hash-Tabelle in Abhängigkeit zu der Anzahl Elemente sein?

Kollisionen beim Hashing

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

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

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

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

Logarithmus: $\ln(1-\varepsilon) \approx -\varepsilon$

da n<<m gilt: $\ln(1-i/m) \approx -(i/m)$

Kollisionen beim Hashing

- Auflösen der Gleichung

$$P(NoCol \mid n,m) \approx \exp\left[-\sum_{i=0}^{n-1}\left(\frac{i}{m}\right)\right]$$

$$= \exp\left[-\frac{n(n-1)}{2m}\right]$$

$$\approx \exp\left[-\frac{n^2}{2m}\right]$$

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

Hashing: Allgemeine Typen

- Entscheidend beim Hashing ist das Problem der **Kollisionen**
  - Eine Kollision tritt auf, wenn zwei Schlüssel den selben Hashwert erhalten und demnach an der selben Stelle in der Hashtabelle gespeichert werden müssten

- Treten Kollisionen beim Hashing auf, so existieren zwei Konzepte:

  - **Offenes Hashing**:

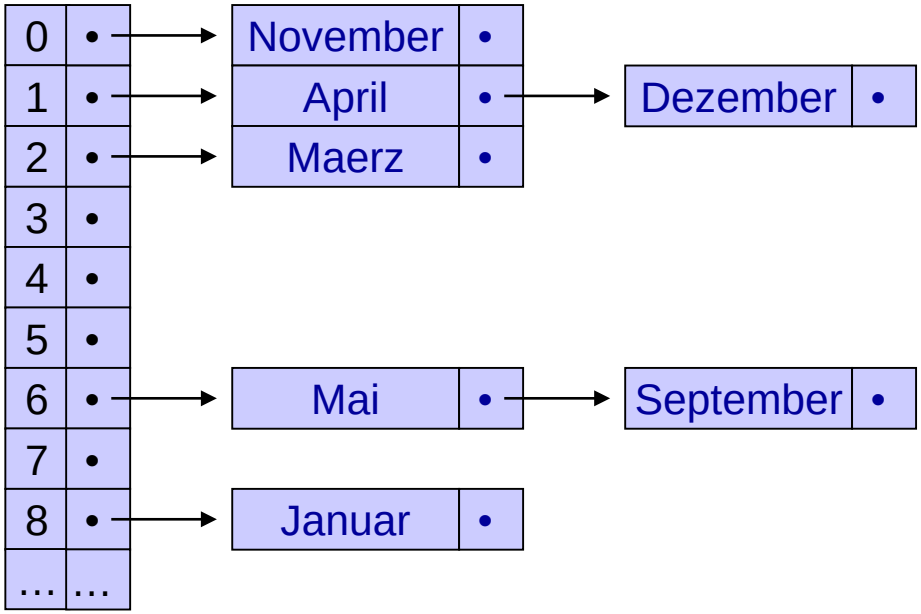
    Bei Kollisionen werden Elemente unter der selben Adresse abgelegt, z.B. als verkettete Liste außerhalb der Tabelle

  - **Geschlossenes Hashing**:

    Speicherung innerhalb der Tabelle, dabei müssen freie Adressen gesucht werden.

Offenes Hashing

- Speicherung der Schlüssel außerhalb der Tabelle, z.B. als verkettete Liste
- Bei Kollisionen werden Elemente unter der selben Adresse abgelegt
- Problem: Wie kann die externe Speicherung effektiv und effizient gelöst werden?



Geschlossenes Hashing

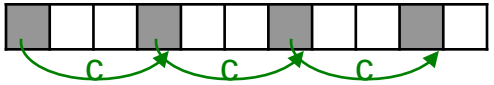
- Bei Kollision wird mittels bestimmter Sondierungsverfahren eine freie Adresse gesucht
- Jede Adresse der Hashtabelle nimmt höchstens einen Schlüssel auf
- Problem: Finden geschickter bzw. effizienter Sondierungsverfahren, so dass nur wenige Sondierungsschritte nötig sind

- Einhashen eines Schlüssels x:

  Bestimme *h(x)* mit der gegebenen Hashfunktion *h* (das entspricht *h(x,0)* mit der Sondierungs-Hashfunktion, s. u.), bei Kollision wird durch eine Folge *h(x,j)* mit j=1, 2, … solange sondiert, bis eine freie Adresse gefunden wird.

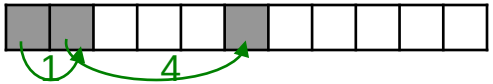
Geschlossenes Hashing: Sondierungsverfahren

– **Lineares Sondieren**:



- Prinzip: Verändern des Hashwertes um eine Konstante *c*
- Sei $h(x,j) = (h(x) + c \cdot j) \bmod m$
- Problem der **Cluster**-Bildung tritt auf:
  - Schlüssel mit ähnlichen Hashwerten liegen unter aufeinanderfolgenden Adressen, deshalb sind viele Sondierungschritte nötig um einen freien Platz zu finden.

– **Quadratisches Sondieren**:



- Sei $h(x,j) = (h(x) + j^2) \bmod m$
- Wie viele Adressen werden von der Hashfunktion wirklich getroffen?
  Ist z.B. *m* eine Primzahl, so ist $(j^2 \bmod m)$ für $j=0,\ldots,\lfloor m/2 \rfloor$ immer verschieden
- Problem: Immer noch Bildung von Clustern:
  - Schlüssel mit gleichem Hashwert werden auf die selben Felder sondiert.

Geschlossenes Hashing: Komplexität

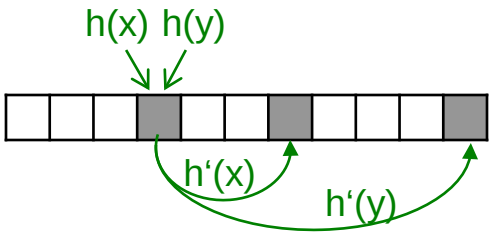
- Hier: Anzahl Sondierungsschritte
  - Einfügen: $C_{Ins}(n,m)$
  - Erfolglose Suche: $C^{-}_{search}(n,m)$
  - Erfolgreiche Suche: $C^{+}_{search}(n,m)$
  - Löschen: $C_{Del}(n,m)$
  - m: Größe der Hash-Tabelle
  - n: Anzahl der Einträge
  - α=n/m: Belegungsfaktor der Hash-Tabelle

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 $P(h(x) = h(y)) = 1 / m$
  - $P(h'(x)=h'(y)) = 1 / m$
  - $P(h(x)=h(y) \wedge h'(x)=h'(y)) = 1 / m^2$
- Sondierung mit $h(x,j) = (h(x) + h'(x) \cdot j^2) \bmod m$
- 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

Zusammenfassung: Hashing

- 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

Kapitel 3 - Suchen in Mengen

Einfache Suchverfahren

Lineare Suche

Binäre Suche

Interpolationssuche

Hashing

Suchbäume

Binärer Suchbaum

AVL-Baum

Splay-Baum

B-Baum

R-Baum

Suchbäume

Bisher betrachtete Algorithmen für Suche in Mengen

- Sortierte Arrays
  - Nur sinnvoll für **statische** Mengen, da Einfügen und Entfernen O(n) Zeit benötigt
  - Zeitbedarf für Suche ist O(log(n)) (Binäre Suche)
  - Bereichsanfragen möglich

- Hashing
  - Stark abhängig von gewählter Hashfunktion
  - Kollisionsstrategie nötig
  - Anzahl der Objekte muss im Groben vorher bekannt sein
  - Keine Bereichs- oder Ähnlichkeitsanfragen

Suchbäume

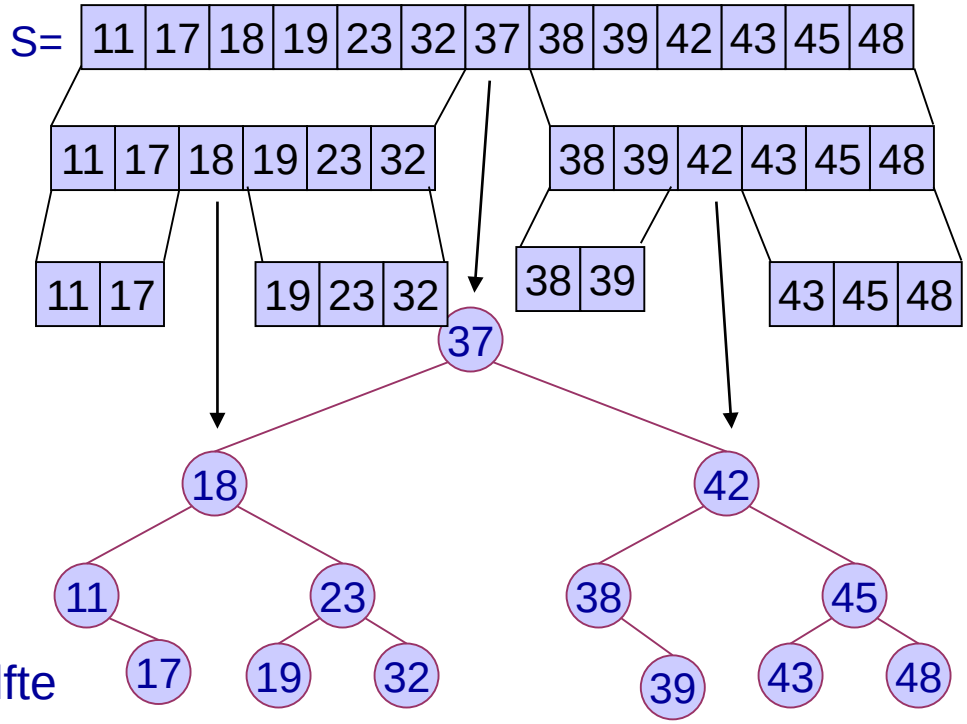
- Suchbäume
  - Beliebig dynamisch erweiterbar
  - Operationen Einfügen, Entfernen und Suchen sind in O(log(n)) realisierbar
  - Effiziente Lösungen für die Verwendung des Sekundärspeichers

- Wir betrachten im weiteren folgende Arten von Bäumen
  - Binäre Suchbäume
  - Balancierte Bäume (binär und nicht binär)
    - AVL-Bäume, B-Bäume, R-Bäume

Binäre Suchbäume

Ausgangspunkt: Binäre Suche

- Start bei der Mitte -> Wurzel
- Aufteilen in:
  - linken Teil (ohne Mitte)
  - rechten Teil (ohne Mitte)
- Rekursiv weiter:
  - linker Teilbaum mit linker Hälfte
  - rechter Teilbaum mit rechter Hälfte

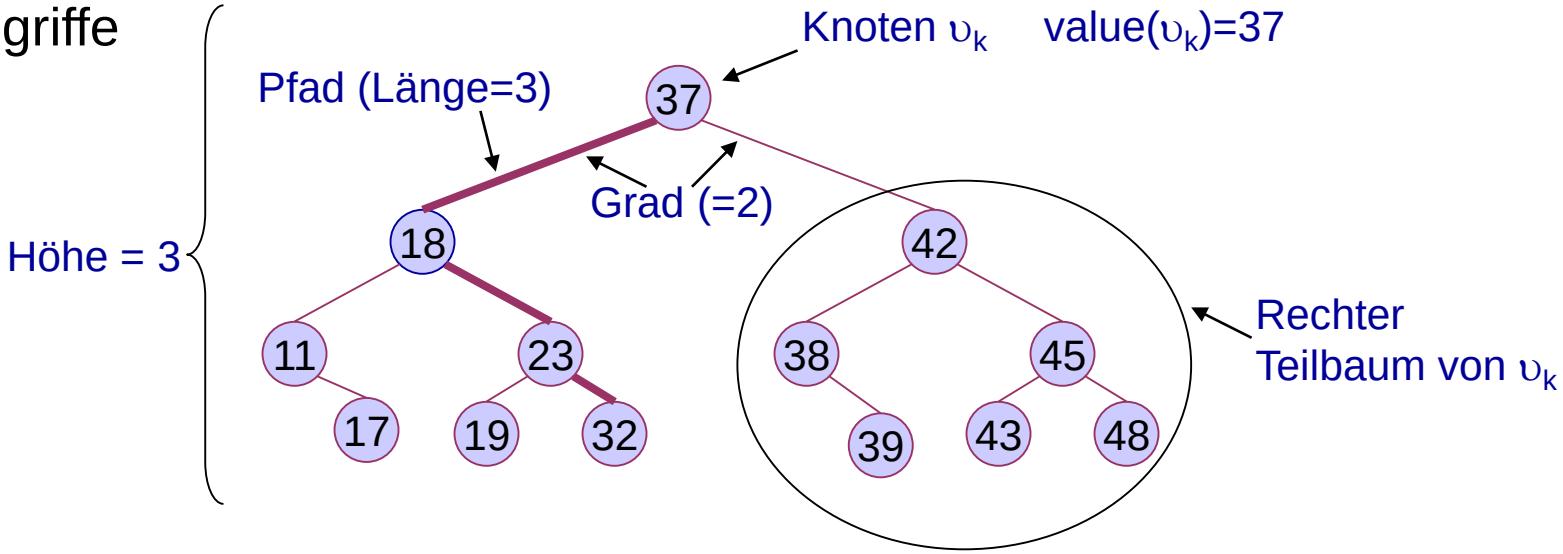


Definition: Binärer Suchbaum

- Definition
  - Ein binärer Suchbaum für eine Menge von Schlüsseln $S=\{x_1, x_2, \ldots, x_n\}$ besteht aus einer Menge beschrifteter Knoten $\upsilon = \{\upsilon_1, \upsilon_2, \ldots, \upsilon_n\}$ mit Beschriftungsfunktion *value* : $\upsilon \rightarrow S$
  - Die Beschriftungsfunktion bewahrt die Ordnung in der Form wenn υ_i im linken Teilbaum von υ_k liegt und υ_j im rechten Teilbaum dann

$$value(\upsilon_i) \leq value(\upsilon_k) \leq value(\upsilon_j)$$

- Begriffe



Binärer Suchbaum vs. Heap

- Ein binärer Suchbaum und ein Heap unterscheiden sich durch ihre strukturellen Invarianten:
  - Wenn υ_i im linken Teilbaum von υ_k liegt und υ_j im rechten Teilbaum dann gilt:

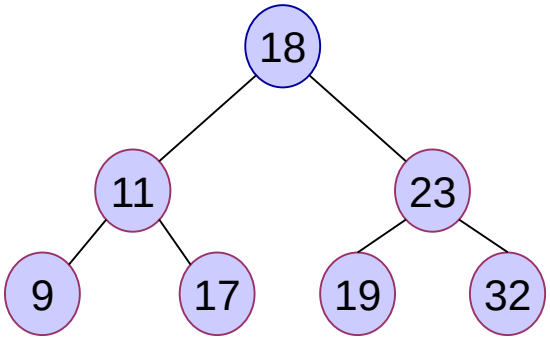
Binärer Suchbaum:

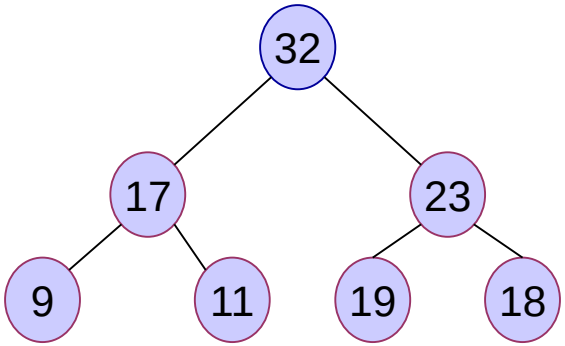
$value(\upsilon_i) \leq value(\upsilon_k) \leq value(\upsilon_j)$

Heap:

$value(\upsilon_i) \leq value(\upsilon_k)$

$value(\upsilon_j) \leq value(\upsilon_k)$





Implementierung: Binärer Suchbaum

- Datenstruktur

```
class Node {
    public int value;              // Schlüssel
    public Node leftChild;
    prublic Node rightChild;

};

class Tree {
    public Node root;              // Wurzel des Baumes
    public void   insert(int value);
    public void   delete(int value);
    public Node search(int value);
};
```

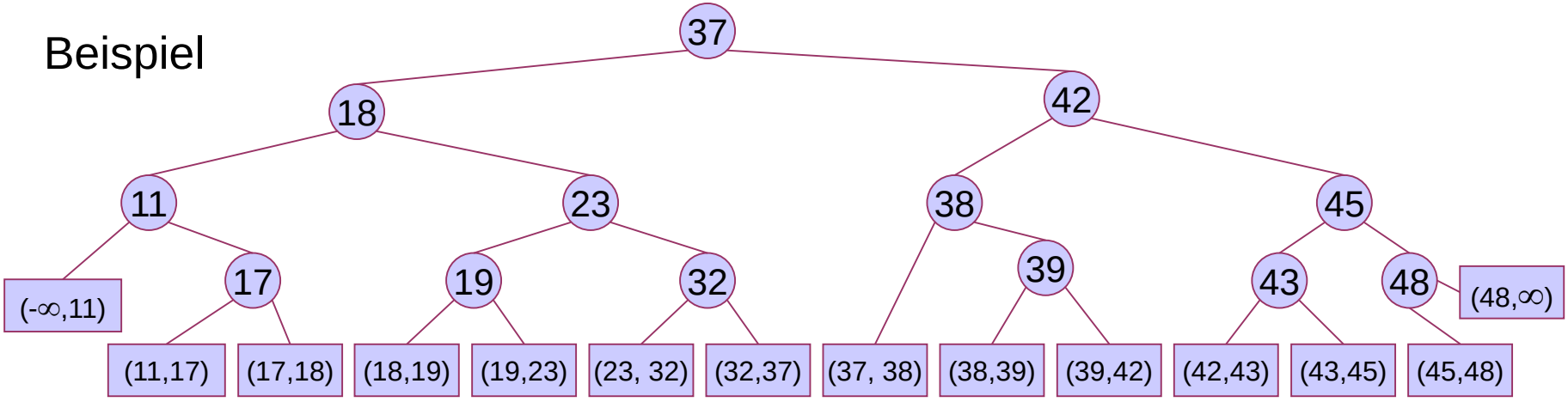
In der Regel werden Knoten durch Ihren Inhalt identifiziert (υ_i = value(υ_i) = x_j)

Erweiterte graphische Darstellung

Graphische Darstellung mit virtuellen Bereichsblättern

- Leere Teilbäume werden als Bereichsblätter dargestellt, welche die Intervalle zwischen den gespeicherten Schlüsseln beinhalten
- Ein binärer Baum mit *n* Knoten besitzt *n+1* Bereichsblätter
- Erfolglose Suche endet immer in einem Bereichsblatt
- Bereichsblätter werden typischerweise nicht gespeichert

Beispiel



Suche in binärem Baum

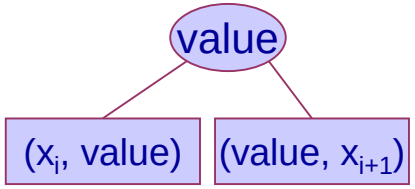
- Implementierung: Analog zur binären Suche
- public Node search (int value) {

```
Node v = root;
while (   v != null &&   v.value!=value   )
{
        if ( value < v.value )          v = v.leftChild;
        else                            v = v.rightChild;
}
return v;
}
```

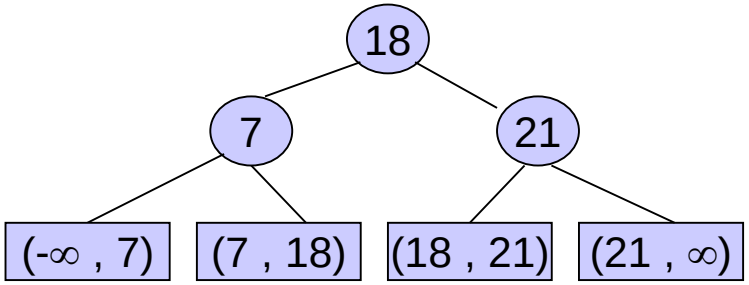
- Die Methode search endet
  1. In einem inneren Knoten, wenn *value* gefunden wurde
  2. In einem leeren Teilbaum (Bereichsblatt), wenn *value* nicht gefunden wurde

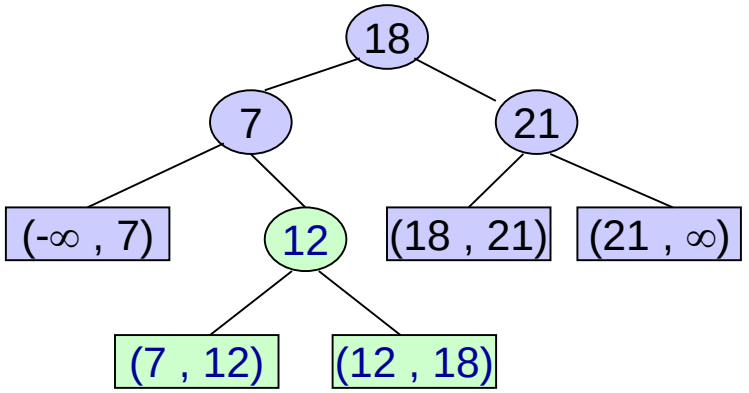
Binärer Suchbaum: Einfügen

- Operation t.insert(value) für Tree t
  - Sucht Element value in t, Ergebnis sei das Bereichsblatt (x_i, x_{i+1})
  - Ersetze Bereichsblatt durch (x_i, x_{i+1}) durch



- Beispiel: Einfügen des Knotens 12
  1. Suche Konten
     - Suche endet in einem Bereichsblatt (=leerer Teilbaum)
  2. Blatt wird durch neuen Knoten ersetzt

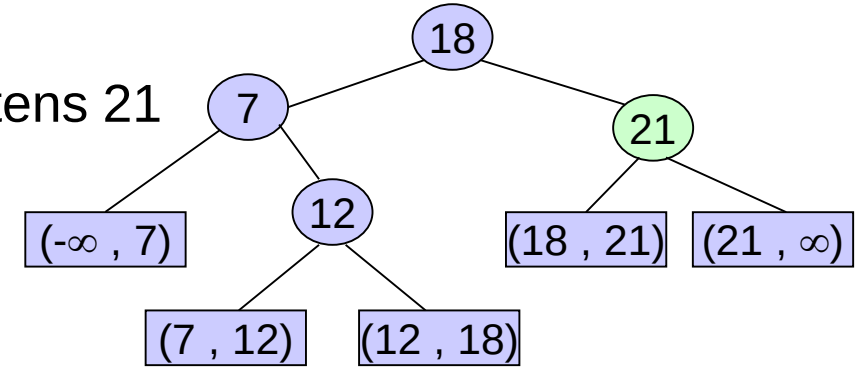




Binärer Suchbaum: Löschen

- Operation t.delete(value)
  - Suche zu löschenden Knoten *v*
  - Es werden 3 Fälle unterschieden:
    1. Falls *v* nur leere Teilbäume hat → *v* kann gelöscht werden
    2. Falls *v* nur einen Teilbaum v_s hat → ersetze *v* durch v_s
    3. Falls *v* zwei Innere Knoten als Kinder hat
       → Suche *w,* den rechtesten (größten) Unterknoten im linken Teilbaum von *v*
       → Ersetze *v* durch *w*
       → Lösche *w*

- Beispiel Fall 1: Löschen des Knotens 21
  -> Kann direkt gelöscht werden

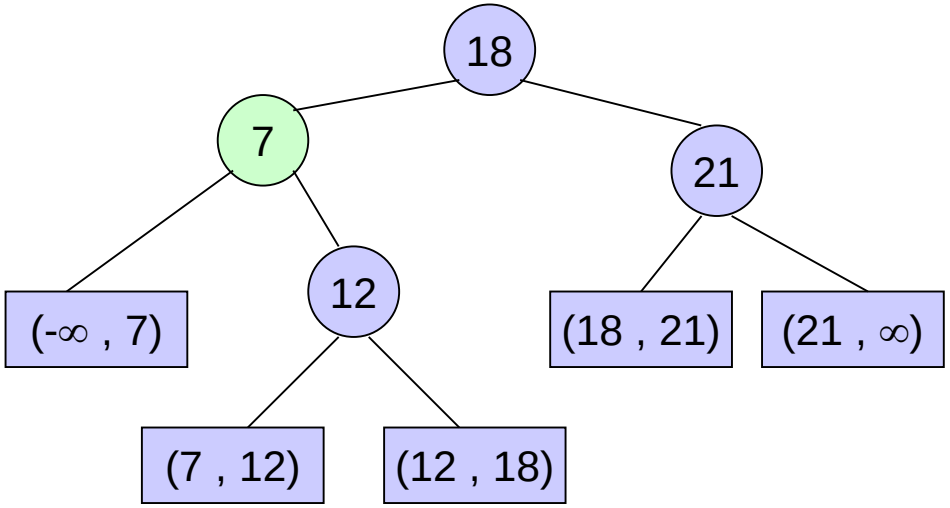


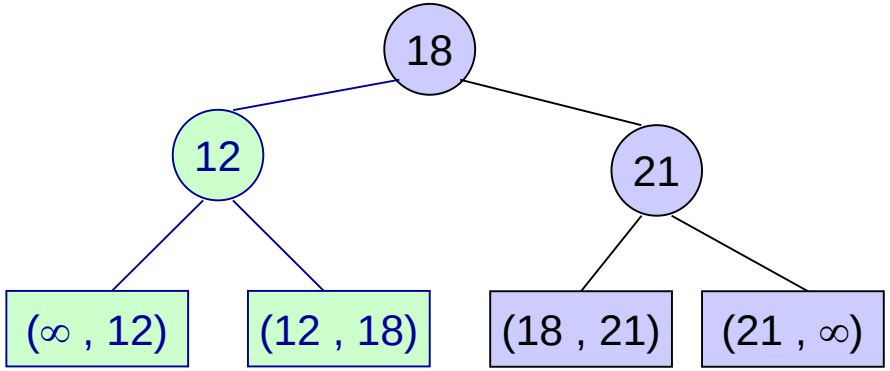
Binärer Suchbaum: Löschen

Beispiel Fall 2:
Löschen des Knotens 7

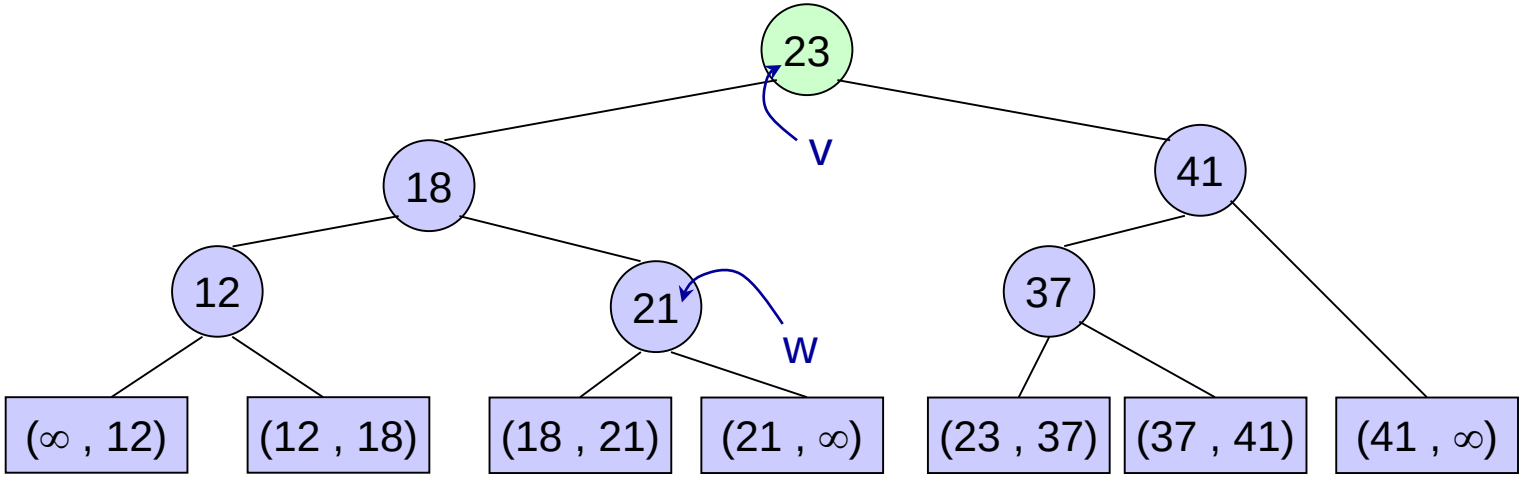
1. Suche Knoten
   - Suche endet in innerem Knoten mit **einem** Nachfolger

2. Nachfolger ersetzt den Knoten





Binärer Suchbaum: Löschen



- Beispiel Fall 3: Löschen von Knoten 23
  Suche Knoten 23 -> Ergebnis: Innerer Knoten mit zwei Nachfolgern

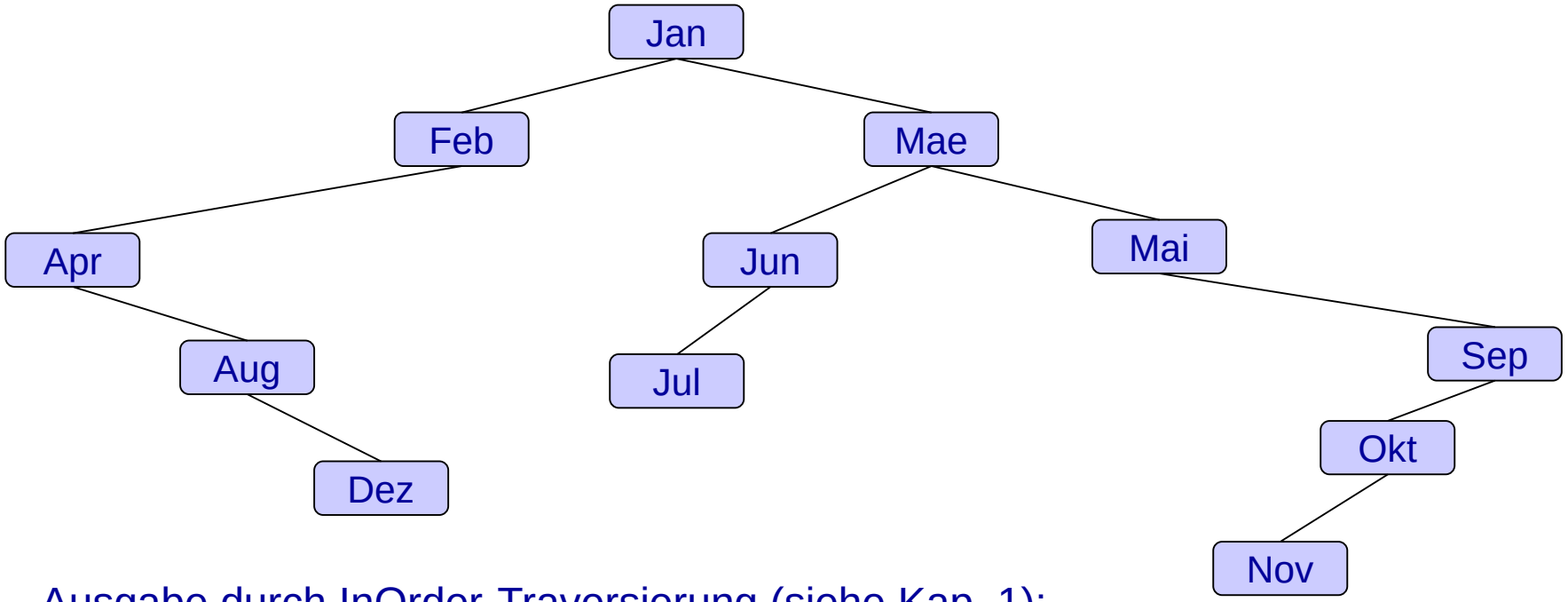
  Suche *w,* den rechtesten Unterknoten im linken Teilbaum von *v*
  - *v = 23*
  - *w = 21*

  Ersetze *v* durch *w*
  - der rechteste Unterknoten im linken Teilbaum von v ist größer als alle Knoten im linken Teilbaum und kleiner als alle Knoten im rechten Teilbaum

  Lösche *w*

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

Komplexitätsanalyse: Binärer Suchbaum

- Analyse der Laufzeit
  - Die Operationen *Insert* und *Delete* bestehen immer aus:
    - Suchen der entsprechenden Position im Baum
    - Lokale Änderungen im Baum in O(1)

- Analyse des Suchverfahrens
  - Anzahl Vergleiche entspricht Tiefe des Baumes, da immer ein Pfad betroffen ist
  - Sei h(t) die Höhe des Suchbaumes t, dann ist die Komplexität der Suche O(h(t))

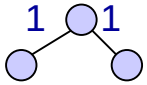
  - Wir benötigen die Komplexität in Abhängigkeit von der Anzahl Knoten.

    ->Frage: Wie hoch ist ein binärer Suchbaum, welcher ‚*n*‘ Knoten beinhaltet?
    Oder: Wie viele Knoten enthält ein Suchbaum der Höhe ‚*h*‘ maximal bzw. minimal ?

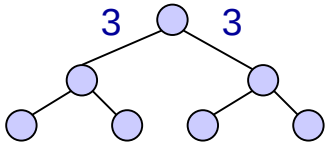
Komplexitätsanalyse: Binärer Suchbaum

- Best Case
  - Alle Knoten bis auf die Blätter haben zwei Nachfolger
  - Höhe h=0 — Anzahl Knoten n = 1

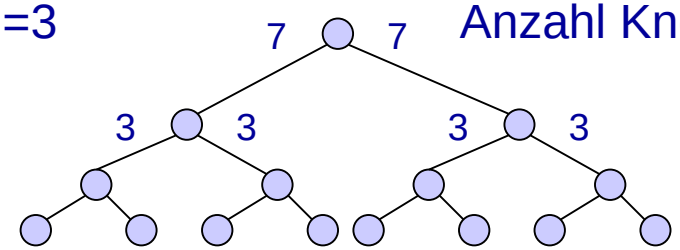
  - Höhe h=1 — Anzahl Knoten $n = 2*1 +1 = 2^{h+1}-1$



  - Höhe h=2 — Anzahl Knoten $n = 2 * (2+1) +1$
    $= 2^2+2 +1=2^3-1$
    $= 2^{h+1} -1$



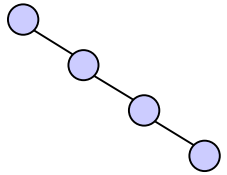
  - Höhe h=3 — Anzahl Knoten $n = 2 * (2^2+2+1) +1$
    $= 2^3+4+2+1=2^4-1$
    $= 2^{h+1} -1$ → Allgemein:
    $n=2 *(2^h -1) +1$
    $=2^{h+1} -2 +1$
    $=2^{h+1} -1$



Ein binärer Suchbaum der Höhe *h* enthält $n = 2^{h+1} -1$ Knoten

Komplexitätsanalyse: Binärer Suchbaum

- Anzahl Vergleiche in Abhängigkeit von *n* (Anzahl Knoten)
- Worst Case
  - Alle Knoten bis auf ein Blatt haben nur einen Nachfolger
  - Baum degeneriert zu einer linearen Liste
  - Baum der Höhe *h* beinhaltet genau *n=h+1* Knoten
  - Komplexität: O(n) → Vergleiche lineare Suche



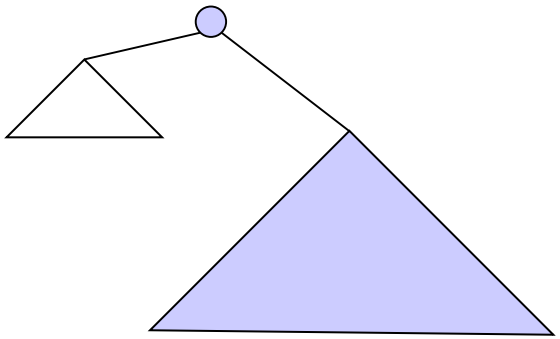
- Best Case
  - Die Anzahl Vergleiche entspricht der Höhe *h* des Baums *t*
  - Voll gefüllter Baum besitzt $n \leq 2^{h+1}-1$ Knoten
    Umformung: $h \geq ld(n+1)-1$ bzw. $h = \lceil ld(n+1)-1 \rceil$
  - Komplexität: O(log n) → Vergleiche binäre Suche

Komplexitätsanalyse: Binärer Suchbaum

- Problemanalyse
  - Der binäre Suchbaum hat im optimalen Fall eine gute Komplexität für die Operationen *Insert*, *Delete*, *Search*
    (Hauptbestandteil ist jeweils Search mit O(log n)
  - Durch die Operationen *Insert*, *Delete* kann ein Binärbaum zu einer linearen Liste entarten

- Ziel
  - Die Operationen *Insert* und *Delete* müssen so verändert werden, dass ein Baum immer ausbalanciert bleibt

Balancieren = Die Höhe der beiden Teilbäume sollte etwa gleich sein (dann auch die Anzahl Knoten)



unbalancierter Baum

Balancierte Bäume

- Ziel:
  - Verhindern der Worst-Case-Komplexität für *Search* O(n)
  - Entartete Bäume verhindern –> Bäume ausbalancieren

- Balancieren (zwei Arten)
  - Gleichgewichtsbalancierung (BB(*a*)-*Bounded Balance* mit Grenze *a*): Die Anzahl der Blätter in den Unterbäumen wird ausbalanciert.

    Dabei beschreibt *a* den maximalen relativen Unterschied zwischen den Teilbäumen.

  - Höhenbalancierung:
    Die Höhe der beiden Teilbäume wird ausbalanciert (Höhe ± 1)

AVL-Baum

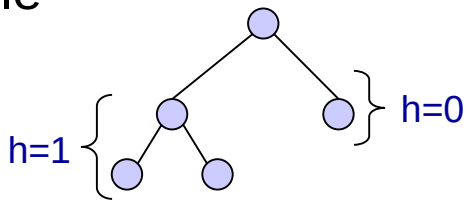
- Historisch erste Variante eines balancierten Baums
- Name basiert auf den Erfindern: Adelson-Velsky & Landis

- Definition:
  - Ein AVL-Baum ist ein binärer Suchbaum mit einer Strukturbedingung (Invariante):
    - Für alle Knoten gilt: Die Höhen der beiden Teilbäume unterscheiden sich höchstens um eins

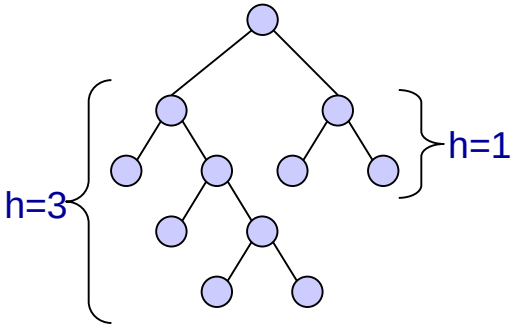
- Operationen:
  - Suchen exakt wie bei binären Suchbäumen
  - Damit die AVL-Bedingung auch nach einer Update-Operation (Insert/Delete) noch gilt, muss der Baum ggf. rebalanciert werden

AVL-Baum

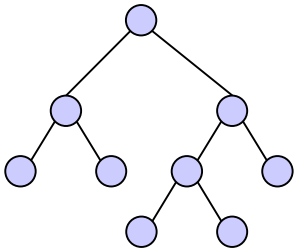
- Beispiele



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-Baum:

Anzahl der Knoten in Abhängigkeit von der Höhe

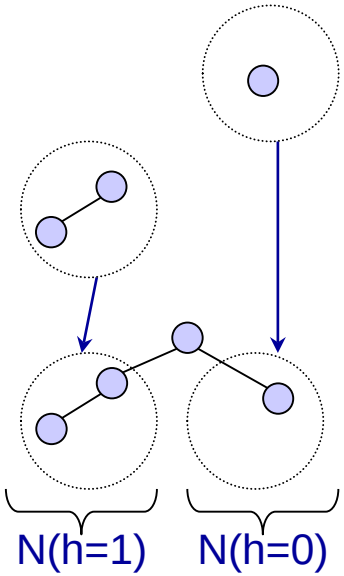
- Gesucht ist die minimale Knotenanzahl
  Also betrachtet man minimal gefüllte Bäume

- Dabei sei N(h) die minimale Anzahl Knoten eines AVL-Baums der Höhe *h*

- Höhe h=0 N(h) = 1 (nur Wurzel)

- Höhe h=1 N(h) = 2 (nur ein Zweig gefüllt)

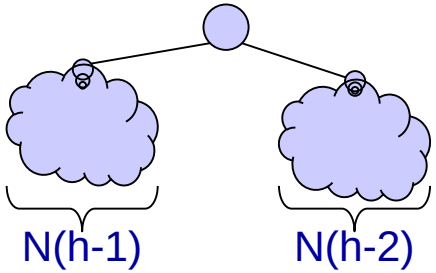
- Höhe h=2 N(h) = 3 (Wurzel mit:
  einem min. Baum h=1
  und einem min. Baum h=2)



AVL-Baum:

Anzahl der Knoten in Abhängigkeit von der Höhe

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



- Ähnlichkeit zu Fibonacci-Reihe

AVL-Baum:

Anzahl der Knoten in Abhängigkeit von der Höhe

- Ein minimal gefüllter AVL-Baum heißt auch Fibonacci-Baum
  - Der Baum besitzt dabei $$N(h) = \begin{cases} 1 & , h = 0 \\ 2 & , h = 1 \\ N(h-1) + N(h-2) + 1, & h > 1 \end{cases} \quad Knoten$$
  - Fibonacci-Reihe $$f(h) = \begin{cases} 0 & , h = 0 \\ 1 & , h = 1 \\ f(h-1) + f(h-2) & , h > 1 \end{cases}$$

AVL-Baum:

Anzahl der Knoten in Abhängigkeit von der Höhe

- AVL-Baum (Fibonacci-Baum)
  - Vergleich Fibonacci-Reihe f(h) mit Höhe AVL-Baum N(h)

– h	=	0	1	2	3	4	5	6	7	8	9
– f(h)	=	0	1	1	2	3	5	8	13	21	34
– N(h)	=	1	2	4	7	12	20	33	54	…	

  - Beweis mittels Induktion N(h) = f(h+3)-1

- Jetzt: Wie hoch ist ein Baum, der aus *n* Knoten besteht?
  - Exakte Berechnung der Fibonacci-Zahlen mit der geschlossenen Formel von Moivre-Binet:

$$f(h) = \frac{\phi_0^h - \phi_1^h}{\sqrt{5}} \quad mit \quad \phi_0 = \frac{1+\sqrt{5}}{2}, \phi_1 = \frac{1-\sqrt{5}}{2}$$

AVL-Baum:

Höhe in Abhängigkeit von der Knotenanzahl

- Für große h gilt: $f(h) \approx \frac{1}{\sqrt{5}} \phi_0^h$
- Also gilt für einen AVL-Baum mit *n* Knoten für die Höhe *h*:

$$N(h) \leq n \quad und \ mit \ N(h) = f(h+3) - 1 \ ergibt \ sich:$$

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

$$\frac{1}{\sqrt{5}} \phi_0^{h+3} \leq n+1$$

$$\log_{\phi_0}\left(\frac{1}{\sqrt{5}}\right) + h + 3 \leq \log_{\phi_0}(n+1)$$

$$h \leq \log_{\phi_0}(n+1) + const$$

AVL-Baum:

Höhe in Abhängigkeit von der Knotenanzahl

$$h \leq \log_{\phi_0}(n+1) + const$$

$$= \frac{ld(n+1)}{ld(\phi_0)} + const = \frac{1}{ld(\phi_0)} ld(n+1) + const$$

$$= 1.4404\ ld(n+1) + const$$

- Ergebnis:
  - Ein AVL-Baum ist maximal 44 % höher als ein maximal ausgeglichener binärer Suchbaum
  - Zur Erinnerung: Die Komplexität der Operation Search hängt nur von der Höhe eines Baums ab

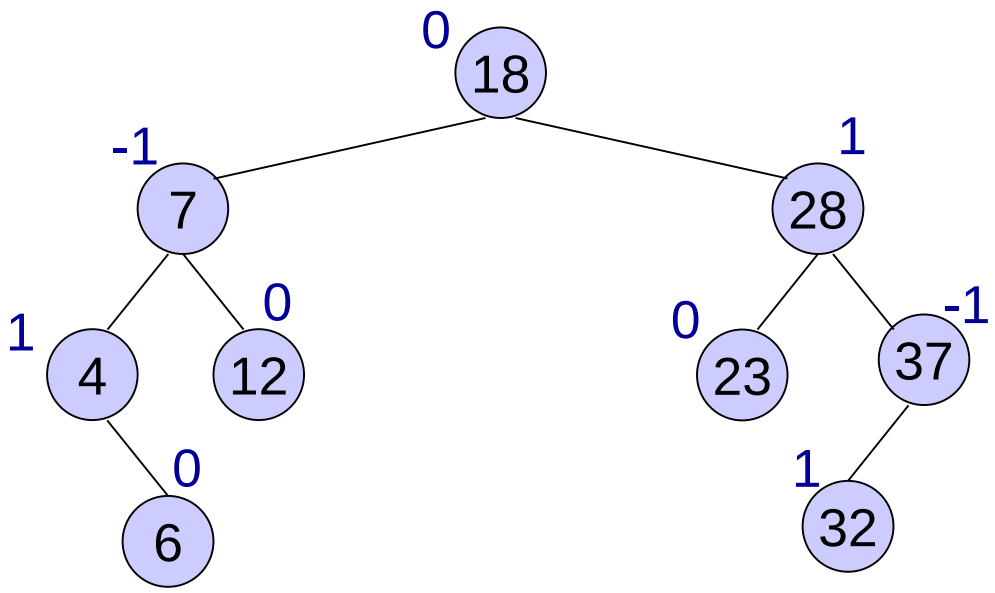
- Operationen
  - Frage: Wie müssen die Operationen Insert/Delete verändert werden, damit die Balance eines AVL-Baums gewährleistet wird?

Balance bei AVL-Bäumen

- Vorgehensweise
  - Bei jedem Knoten wird die Höhendifferenz (Balance *b*) der beiden Teilbäume mit abgespeichert:

b = Höhe(Rechter Teilbaum) - Höhe(Linker Teilbaum)

- Beispiel



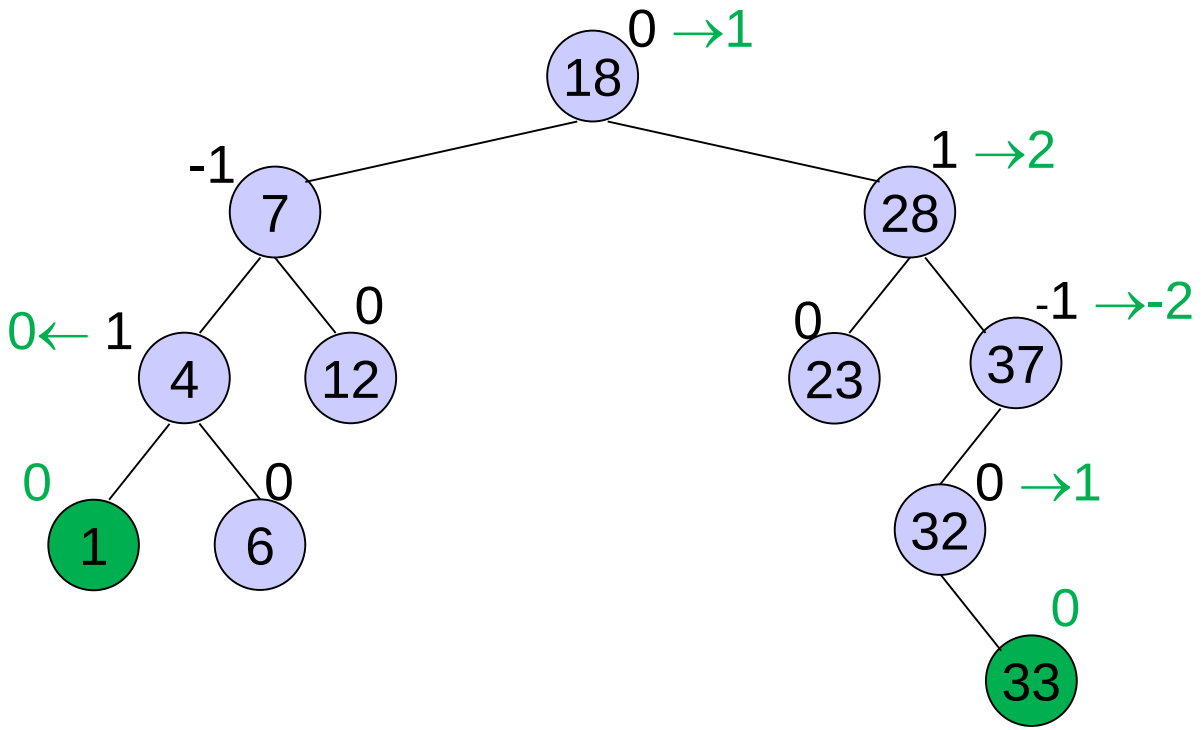
Einfügen bei AVL-Bäumen

- Zuerst normales Einfügen wie bei binären Bäumen
- Beim Einfügen kann sich nur die Balance *b* von Knoten verändern, welche auf dem Suchpfad liegen

$\rightarrow$ Dabei kann das AVL-Kriterium verletzt werden

- Gehe nach dem Einfügen eines neuen Knotens den Suchpfad wieder zurück und aktualisiere die Balance

Einfügen bei AVL-Bäumen



Beispiel: Einfügen von 1 und 33

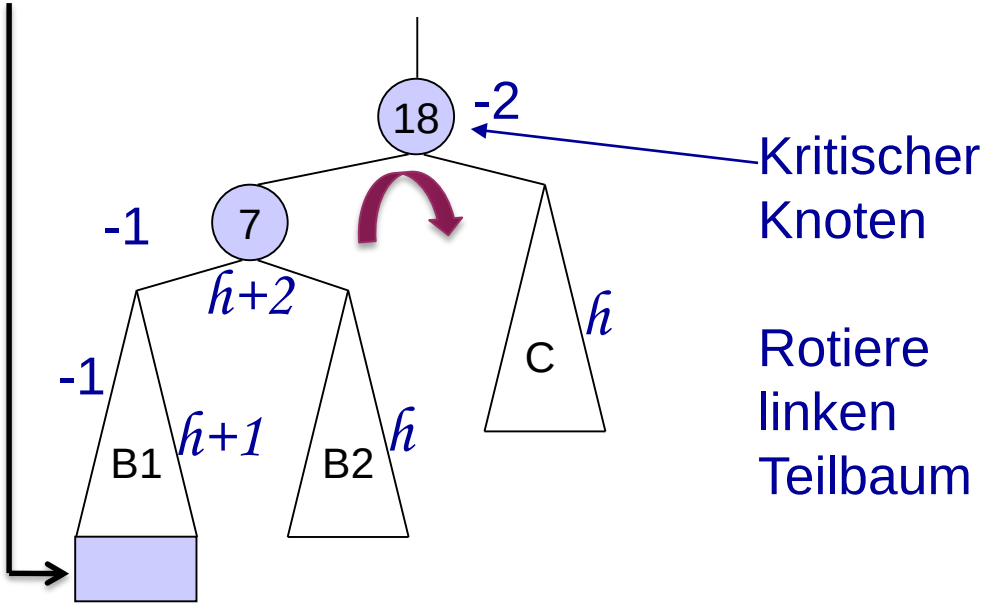
Ablauf:

- Nach dem „normalen“ Einfügen den kritischen Knoten bestimmen (nächstgelegener Vorgänger zum neuen Knoten mit Balance $b = \pm 2$): Dieser ist Ausgangspunkt der Reorganisation („Rotation“)
- Der Pfad vom kritischen zum neuen Knoten legt den Rotationstyp fest.

AVL-Baum: Einfachrotation

- Rechtsrotation

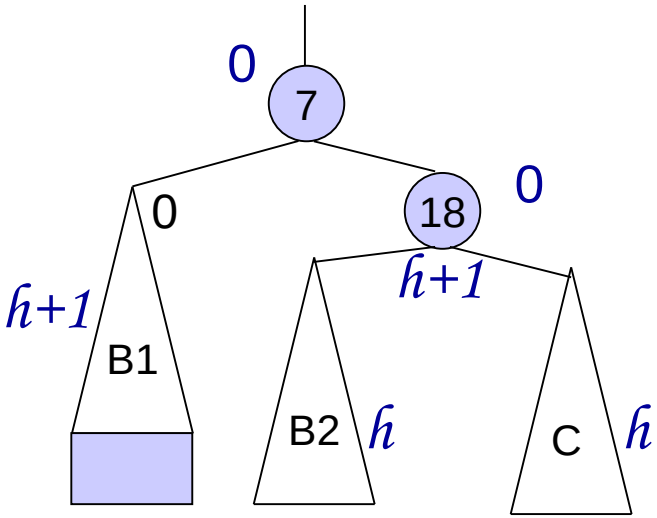
Beispiel: Einfügung war in Teilbaum „links links“ (Balance=-2)



Kritischer Knoten

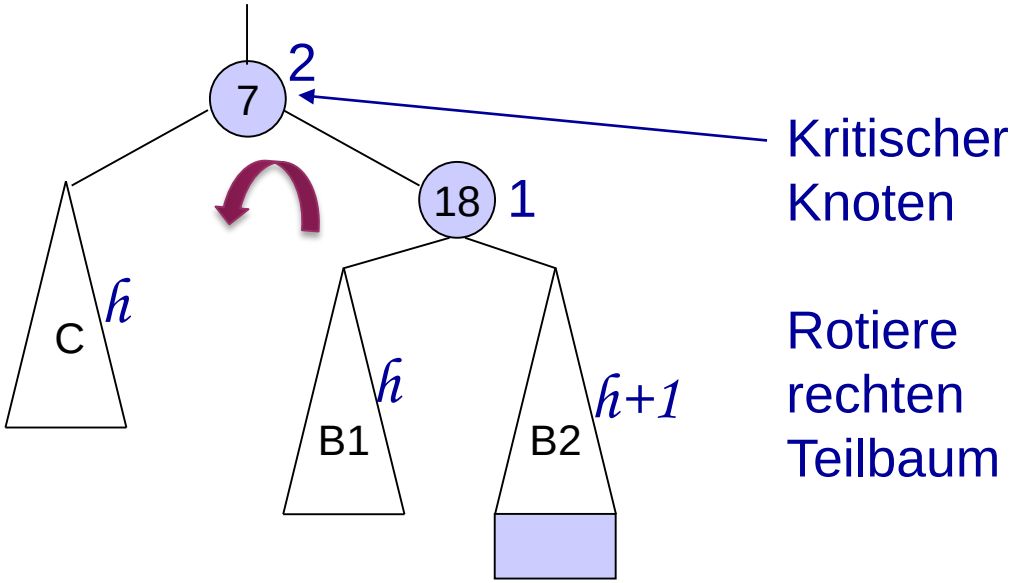
Rotiere linken Teilbaum

Baum ist nach der Rotation wieder balanciert



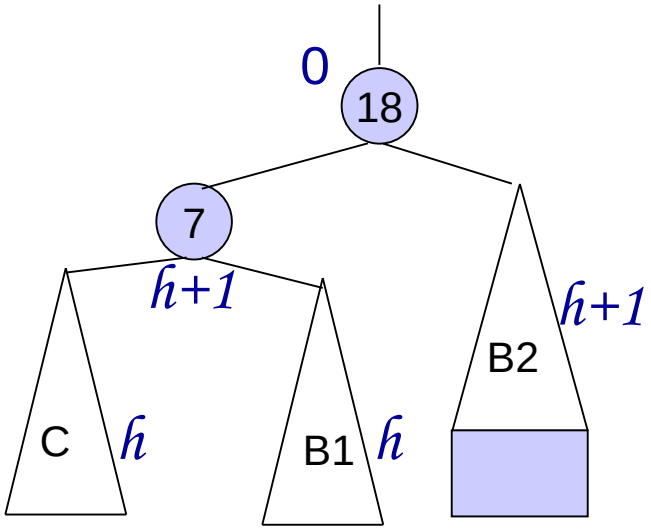
AVL-Baum: Einfachrotation

- Linksrotation: Einfügung war in Teilbaum „rechts rechts“ (Balance=2)



Kritischer Knoten

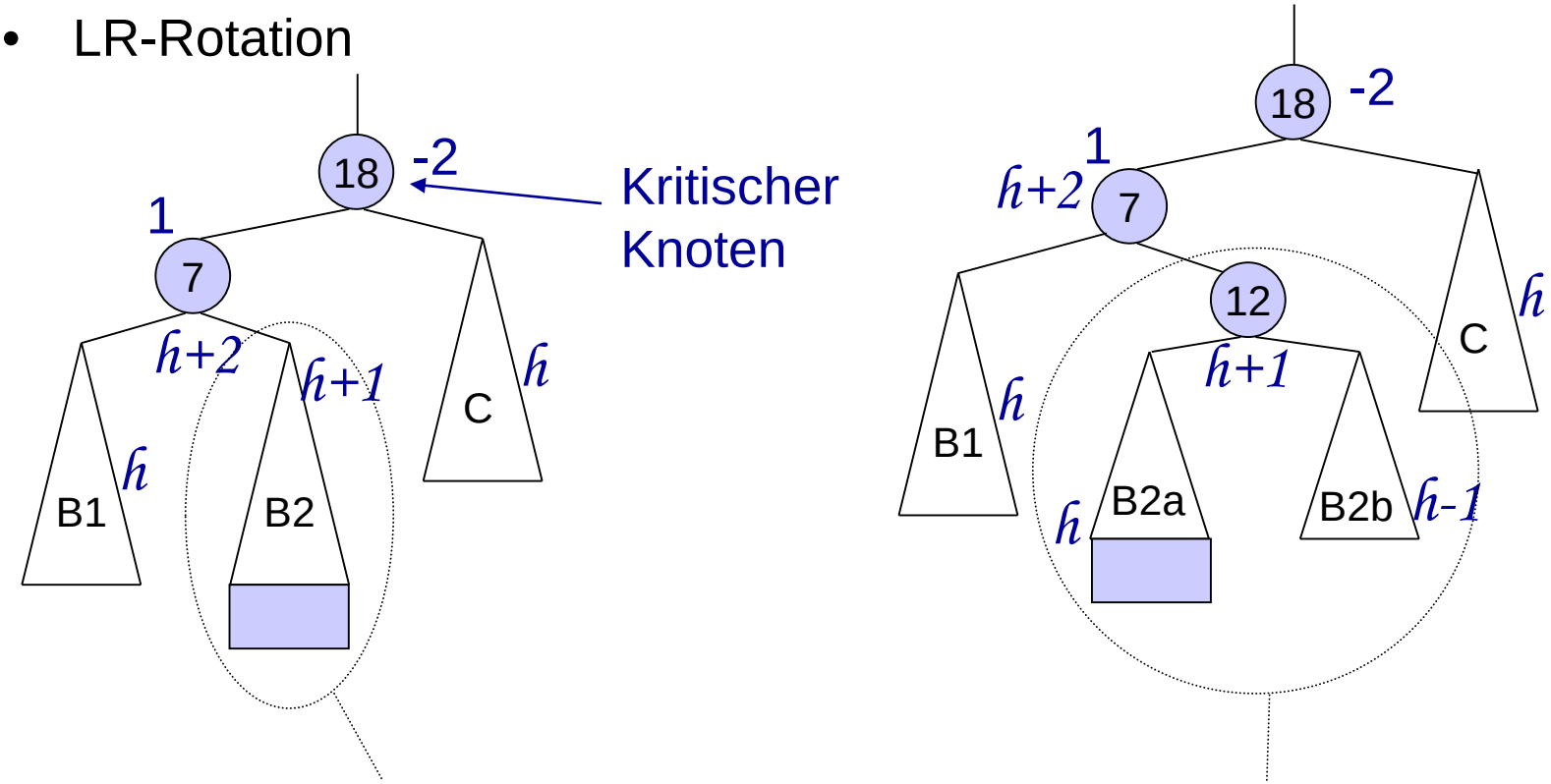
Rotiere rechten Teilbaum



Symmetrisch zur Rechtsrotation

AVL-Baum: Doppelrotation

- LR-Rotation

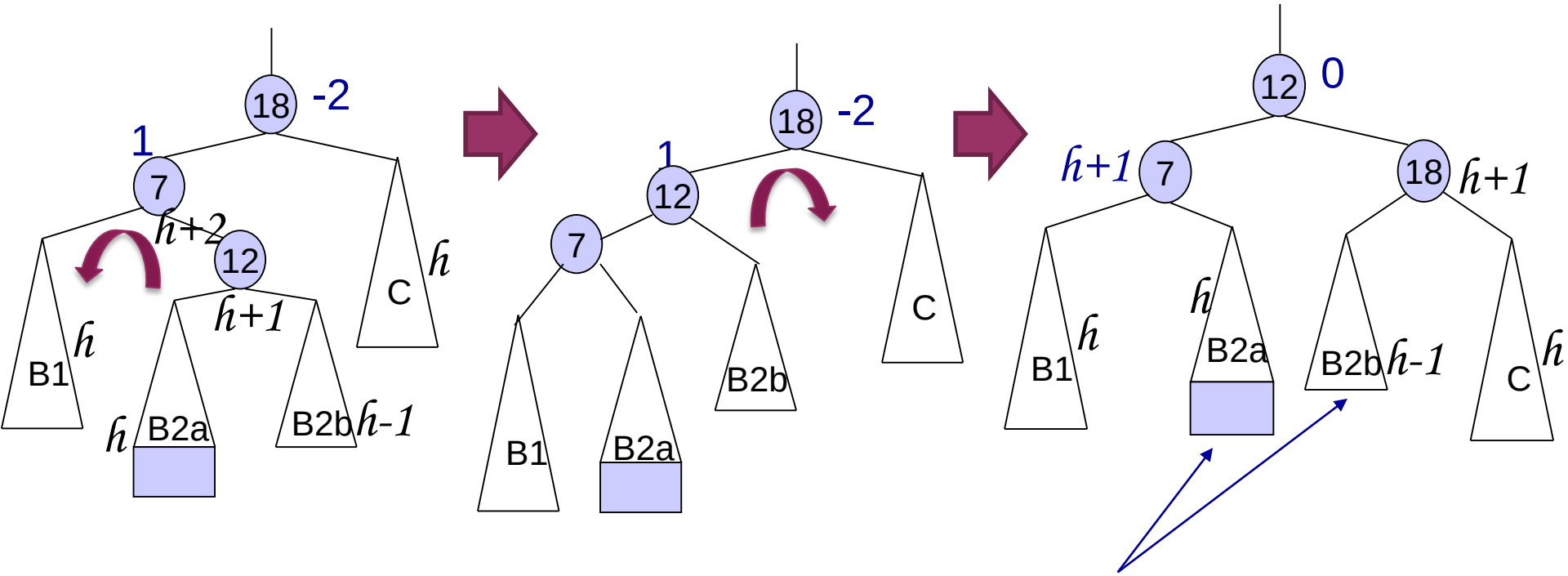


Eine einfache Rotation ist nicht mehr ausreichend, da der problematische Baum innen liegt

→ der Baum B2 muss näher betrachtet werden

AVL-Baum: Doppelrotation

- LR-Rotation



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-Baum: Komplexität beim Einfügen

Komplexität

- 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)	+	const	=	O(log(n))

Löschen bei AVL-Bäumen

Vorgehensweise

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

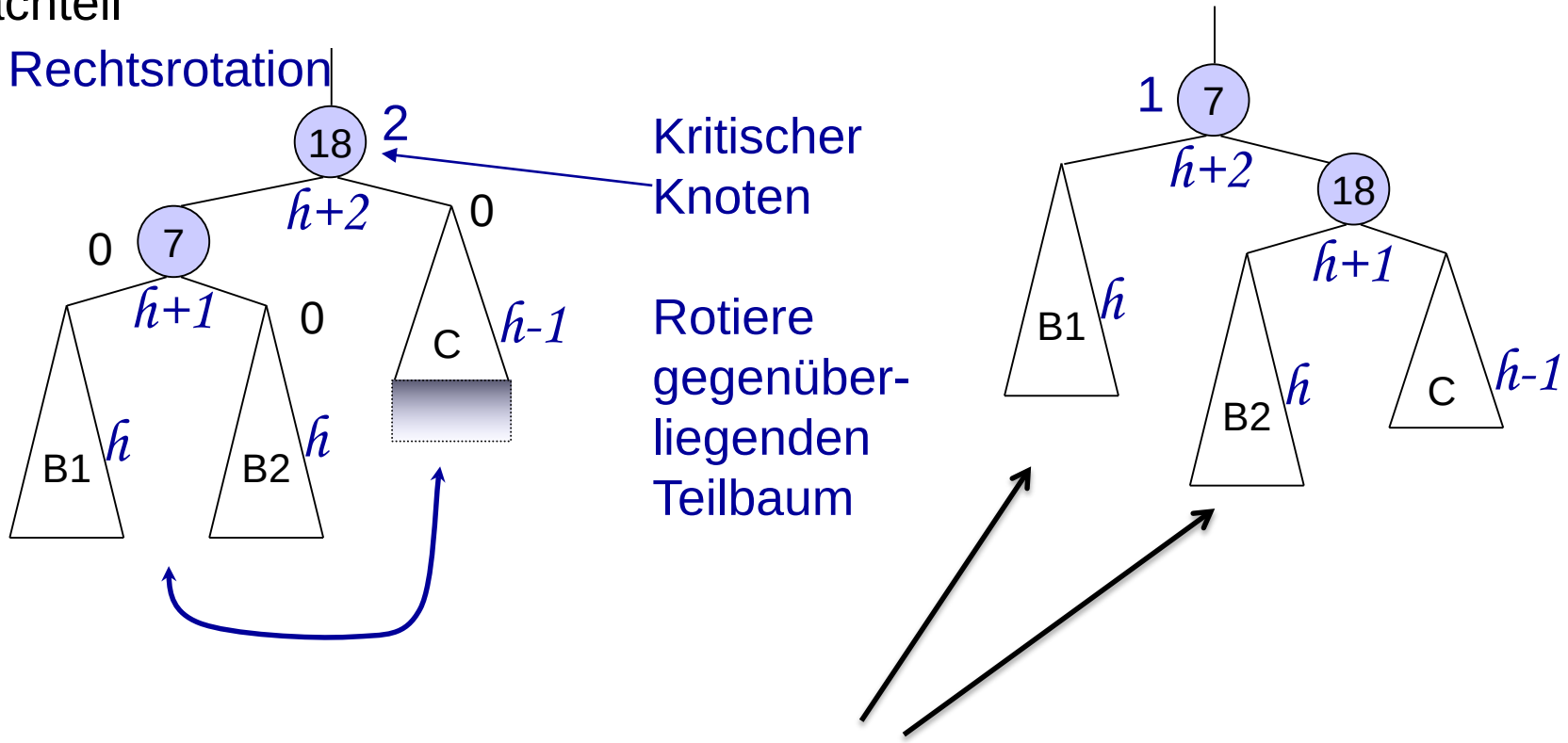
Ablauf:

- 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

Löschen bei AVL-Bäumen

Nachteil

- Rechtsrotation



- 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.

Löschen bei AVL-Bäumen

Komplexität

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

  → auf dem weiteren Pfad zur Wurzel kann es zu weiteren Rebalancierungen (des obigen Typs, also immer im anderen Unterbaum) kommen

  → 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, d.h. einige Anfragen treten häufiger auf, ist es wünschenswert, wenn sich der Baum an diese anpasst.
  -> 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 welche mit einer einzelnen Grundoperation gekoppelt sind, dem sogenannten Splay.
  - Der Splay platziert das gegebene Element als Wurzel des Baums.
  - Splay-Bäume haben keine strukturelle Invariante wie AVL-Bäume, welche für deren Effizienz verantwortlich ist.

    Einzig der Splay führt zu einer heuristischen Restrukturierung.

Splay-Bäume: Operationen

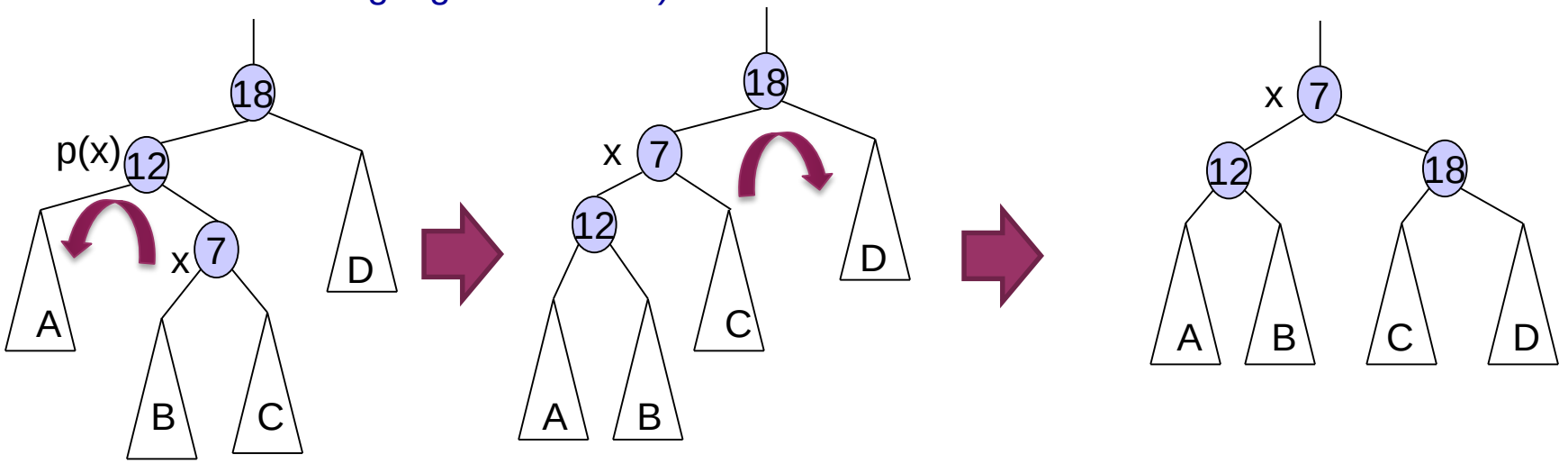
- Suchen
  - Normale Binärsuche im Suchbaum
  - Endet in Knoten *x* mit Schlüssel *k*
  - Wende Operation Splay auf Knoten *x* an
- Einfügen
  - Normale Binärsuche im Suchbaum
  - Einfügen eines Knotens als Blatt
  - Wende Splay auf diesen Knoten an
- Löschen
  - Normale Binärsuche im Suchbaum
  - Entferne den gefundenen Knoten wie im Binärbaum

Splay-Bäume: Splay

- Der Splay repositioniert einen gegebenen Baumknoten als Wurzel.
- Umsetzung: Sukzessives Rotieren, bis der Knoten die Wurzel ist.
- Die Art der Rotation ist abhängig vom Kontext des Knotens x, wobei 3 Fälle zu unterscheiden sind:
  - Der Knoten x hat die Wurzel als Vorgänger:
    - Hier reicht eine einzelne Rechts- bzw. Linksrotation (zig bzw. zag) wie bei AVL-Bäumen

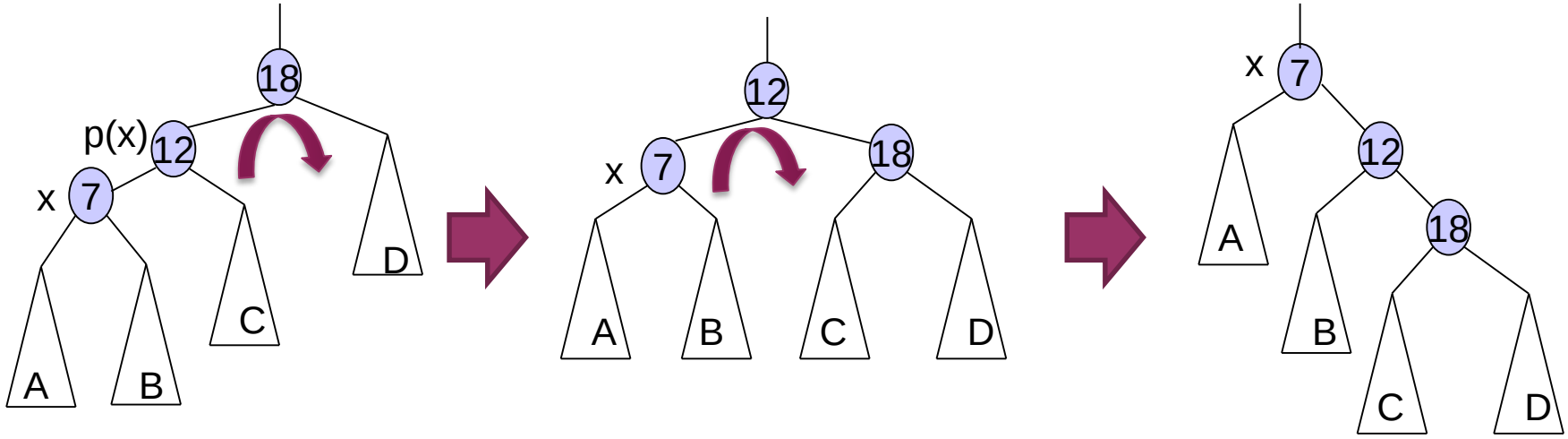
Splay-Bäume: Splay

- Der Knoten x ist ein linkes Kind und der Vorgänger p(x) ist ein rechtes Kind bzw. umgekehrt:
  - Hier ist eine Doppelrotation vonnöten, wie sie als RL- bzw. LR-Rotation bei AVL-Bäumen bekannt ist. Beim Splay-Baum werden diese Operationen als zig-zag bzw. zag-zig bezeichnet.)



Splay-Bäume: Splay

- Der Knoten x sowie sein Vorgänger p(x) sind linke bzw. rechte Kinder:
  - In diesem Fall werden zwei Einzelrotationen durchgeführt, jedoch in Top-Down-Reihenfolge (zig-zig bzw. zag-zag), d.h. anders als bei AVL-Bäumen.



Splay-Bäume: Zusammenfassung

- Gut geeignet zur Umsetzung von Caches oder Garbage Collection

  Aber: Bei Gleichverteilung der Anfragen ist die Suchkomplexität schlechter als bei einfachen binären Suchbäumen.
- Vorteil gegenüber AVL-Bäumen:
  - 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
- 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.

Zusammenfassung: Binäre Bäume

- Vergleich mit Hashing
  - Hashing hat im optimalen Fall konstante Komplexität O(1), im schlechtesten Fall O(n)
  - Damit Hashing gut funktioniert müssen viele Parameter bekannt sein
    - Wertebereich
    - Größe der Tabelle
    - Gute Hashfunktion
  - Bäume haben in allen Operationen konstant logarithmische Komplexität O(log n)
  - Bei Bäumen ist vorab kein Wissen über die Daten notwendig
- Java-Applet, welches verschiedene Binärbäume demonstriert:
  http://webpages.ull.es/users/jriera/Docencia/AVL/AVL%20tree%20applet.htm

Verwendung von Sekundärspeicher

Motivation

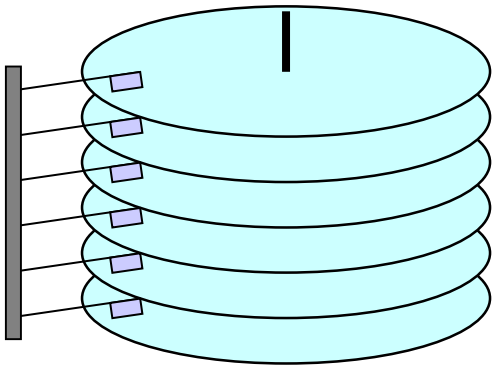
- Falls Daten persistent gespeichert werden müssen
- Falls Datenmenge zu groß für Hauptspeicher

Sekundärspeicher/Festplatte

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

Zugriffszeit Festplatten

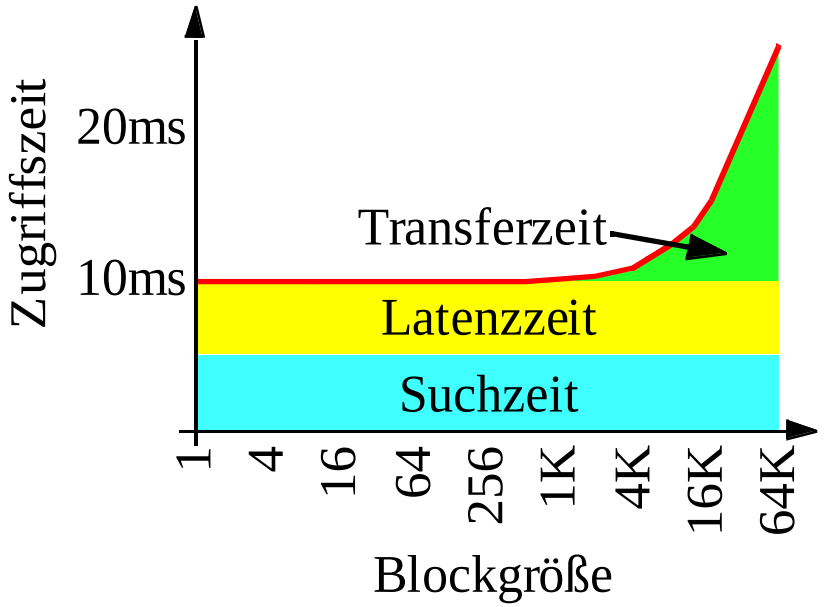
- **Suchzeit** [ms]: Armpositionierung (Translation)
- **Latenzzeit** [ms]: Rotation bis Blockanfang
- **Transferzeit** [ms/MB]: Übertragung der Daten



Verwendung von Sekundärspeicher

Blockgrößen

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



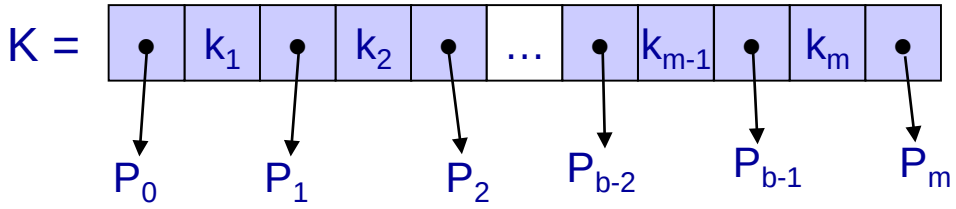
Problem

- Seitenzugriffe sind teurer als Vergleichsoperationen

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

Definition: Mehrwegbäume

- Knoten haben n≥2 Nachfolger



Knoten $K=(P_0, k_1, P_1, k_1, P_2, \ldots P_{m-1}, k_m, P_m)$ 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: Motivation

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

- Im Unterschied zum Hashing eine dynamische Datenstruktur, die
  - Bereichsanfragen ermöglicht (Ordnungserhaltung)
  - effizient wachsen und schrumpfen kann

- Bedeutung des „B“:
  - **B**alanced Tree (technische Beschreibung)
  - **B**ushy Tree, **B**road Tree (Hinweis auf hohen Verzweigungsgrad)
  - Prof. Rudolf **B**ayer, Ph.D. (mit Ed McCreight Erfinder der B-Bäume)
  - 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, da Mehrwegebäume; richtig, da binäre Suche in den Knoten)

B-Baum: Definition

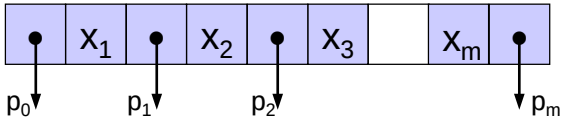
- Ein **B-Baum** der Ordnung k und Höhe h ist ein balancierter Mehrwegbaum

- T ist leer ($h = 0$) oder hat die folgenden Eigenschaften:

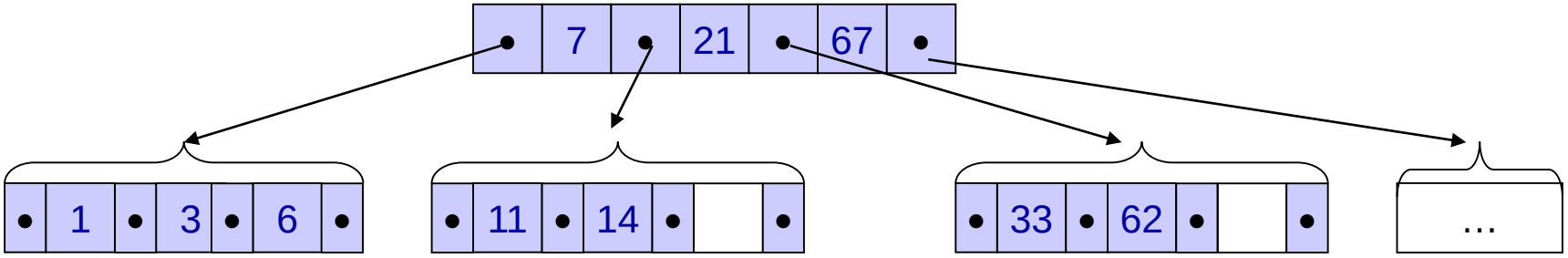
  - jede Seite außer der Wurzel enthält zwischen k und $2k$ Schlüssel; die Wurzel enthält zwischen 1 und $2k$ Schlüsseln.
  - eine Seite mit m Schlüsseln, die kein Blatt ist, besitzt $m + 1$ Kinder.
    - jeder Knoten außer der Wurzel und den Blättern hat mindestens $k + 1$ Kinder
    - die Wurzel ist ein Blatt oder hat mindestens zwei Kinder
    - jeder Knoten hat höchstens $2k + 1$ Kinder
  - jeder Pfad von der Wurzel zu einem Blatt hat die selbe Länge $l \geq 1$

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:
  - $\forall y \in K(p_0)$: $y < x_1$
  - $\forall y \in K(p_i)$, $1 < i < m - 1$: $x_i < y < x_{i+1}$
  - $\forall y \in K(p_m)$: $x_m < y$

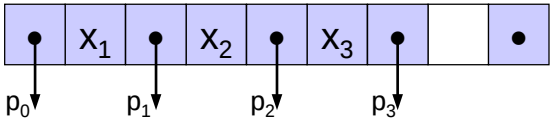


- *Beispielbaum:*

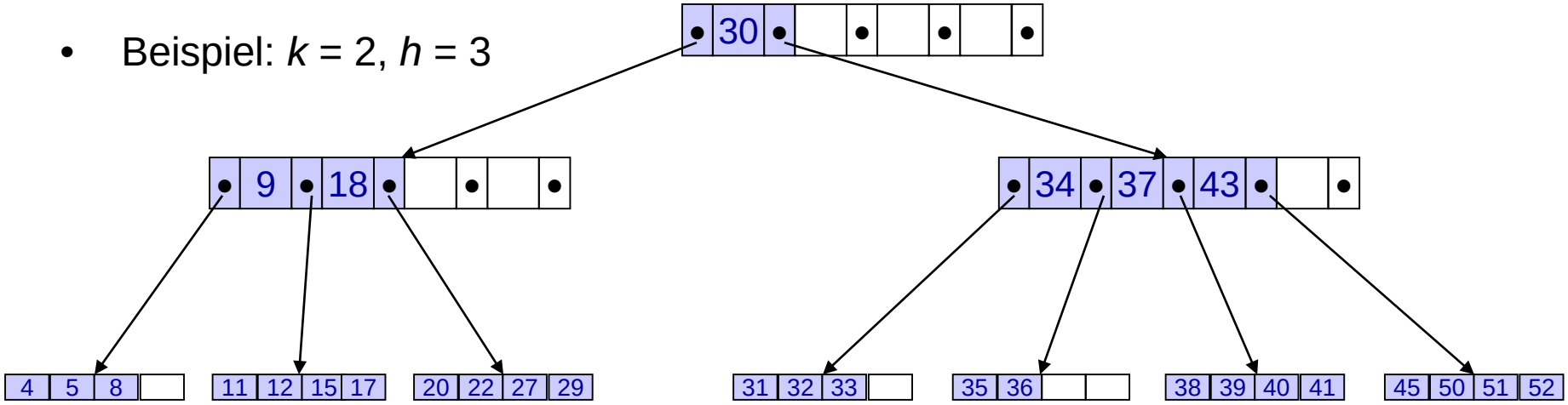


B-Baum: Beispiel

- Die Ordnung $k \in IN$ 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 $k = 40$.

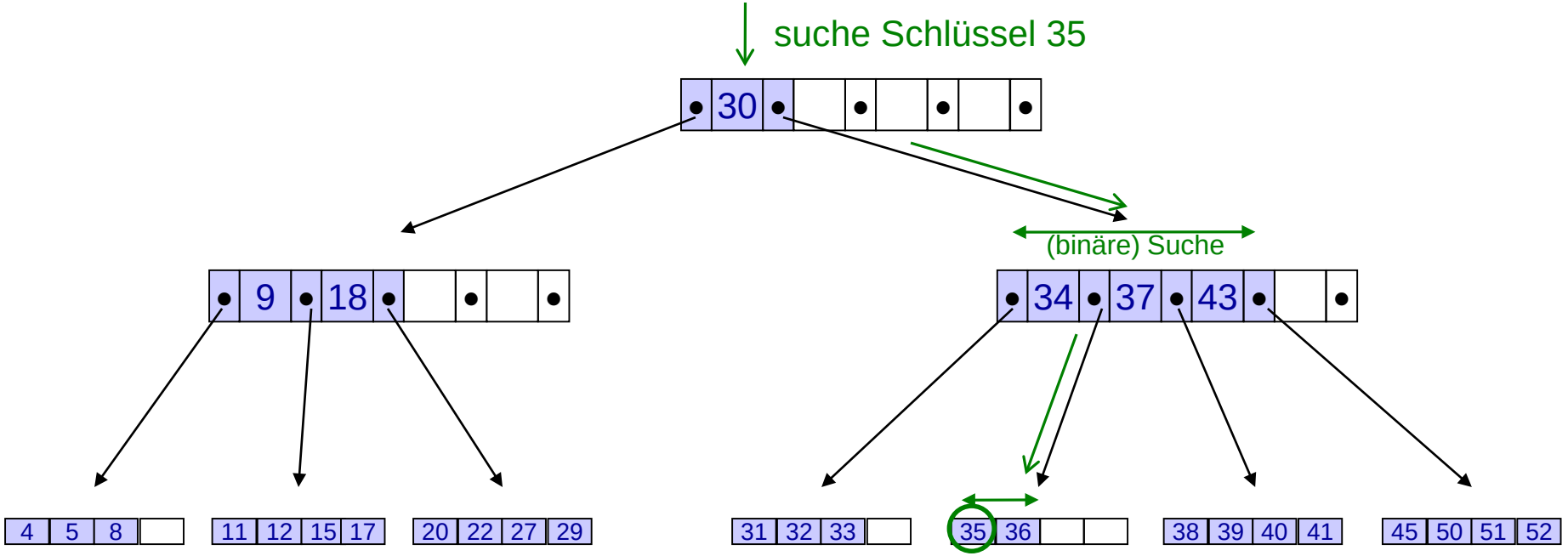


- Beispiel: $k = 2$, $h = 3$



Suchen im B-Bäumen

- 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

- Wie hoch wird ein B-Baum mit 1.000, 100.000, 1.000.000,10.000.000, 1.000.000.000 Schlüsseln für die Ordnung k=100?

  Ein Baum der Höhe h enthält mindestens die folgende Anzahl an Schlüsseln auf einer Ebene:

	Schlüssel pro Ebene	Nachfolger	Summe
h=0	1	2	1
h=1	2 * k	2*(k+1)	201
h=2	2*(k+1)*k	2*(k+1)*(k+1)	20401
h=3	2*(k+1)*(k+1)*k	2*(k+1)*(k+1)*(k+1)	1.6241.201
h=4	2*(k+1)*(k+1)*(k+1)*k	2*(k+1)*(k+1)*(k+1)*(k+1)	3.264.481.601

falsch

D.h.: Bäume von 201 bis 80800 haben höchstens Höhe 1

  - Ein Baum mit 1.000 Schlüssel hat höchstens die Höhe 1
  - Ein Baum mit 100.000 sowie 1.000.000 Schlüssel hat höchstens die Höhe 2
  - 10.000.000 und 1.000.000.000 Schlüssel passen in einen B-Baum der Höhe 3

B-Baum: Höhenabschätzung

- Maximale Schlüsselzahl eines Baums der Höhe $h \geq 0$:
  - Wurzel (h=0): max. 2k Schlüssel, also 2k+1 Nachfahren → 2k Schlüssel
  - erste Ebene (h=1): jeder Knoten max. $2k$ Schlüssel → $2k \cdot (2k+1)$
  - zweite Ebene (h=2): wieder max. $2k$ Schlüssel → $2k \cdot (2k+1)^2$
  - …
  - h-te Ebene: jeweils $2k$ Schlüssel in $(2k+1)^h$ Knoten → $2k \cdot (2k+1)^h$
  - Man erhält:

$$N_{\max} = 2k + 2k(2k+1) + 2k(2k+1)^2 + \ldots = 2k\left(\sum_{i=0}^{h}(2k+1)^i\right) \overset{geom.\,Reihe}{=} (2k+1)^{h+1} - 1$$

- Minimale Schlüsselzahl eines Baums der Höhe $h \geq 0$:
  - In analoger Vorgehensweise erhält man

$$N_{\min} = 1 + 2k + 2k(k+1) + 2k(k+1)^2 + \ldots = 1 + 2k\left(\sum_{i=0}^{h-1}(k+1)^i\right) \overset{geom.\,Reihe}{=} 2(k+1)^h - 1$$

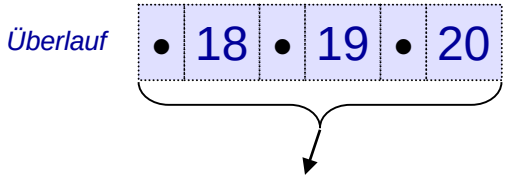
→ $h \approx \log_{k+1}(N)$ max. Höhe
→ $h \approx \log_{2k+1}(N)$ min. Höhe

Somit ist die Höhe logarithmisch durch die minimale bzw. maximale Anzahl Nachfolger eines Knoten beschränkt

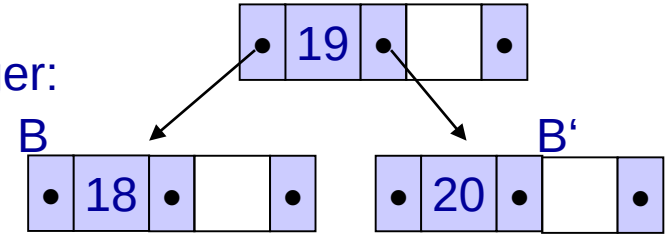
Einfügen in B-Bäumen

Algorithmus

- Durchlaufe den Baum und suche das Blatt B, in welches der neue Schlüssel gehört
- Füge x sortiert dem Blatt hinzu
  - Wenn hierdurch das Blatt $B = (x_1,..,x_{2k+1})$ überläuft: B= • 19 • 20 • hier k=1
    ->Split



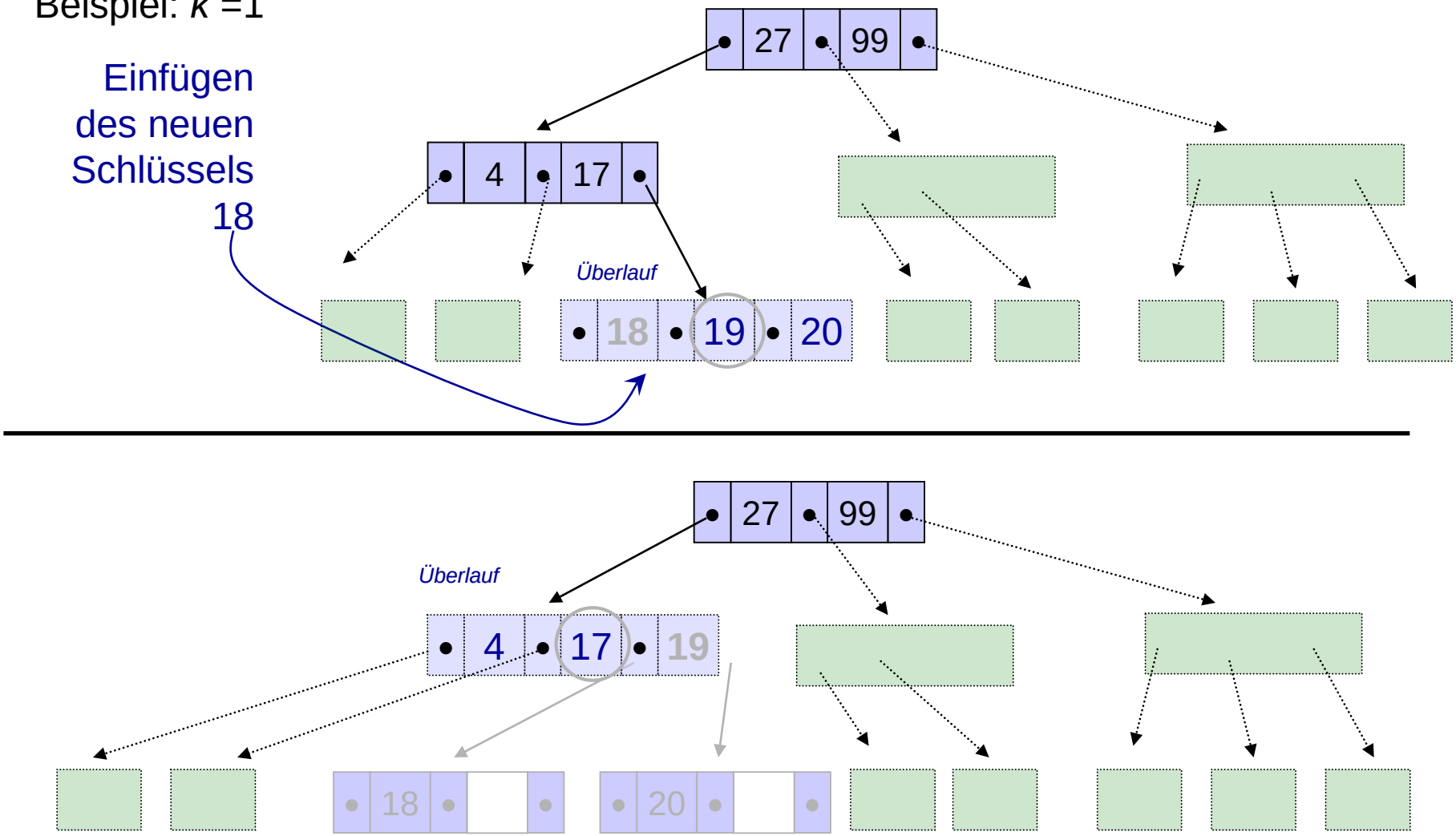
1. Erzeuge ein neues Blatt B‘
2. Verteile Schlüssel auf altes und neues Blatt
   $B = (x_1,..,x_k)$ und $B‘ = (x_{k+2},..,x_{2k+1})$
3. Füge den Schlüssel x_{k+1} dem
   Vorgänger hinzu ggf. erzeuge neuen Vorgänger:
   x_{k+1} dient als Trennschlüssel für B und $B‘$



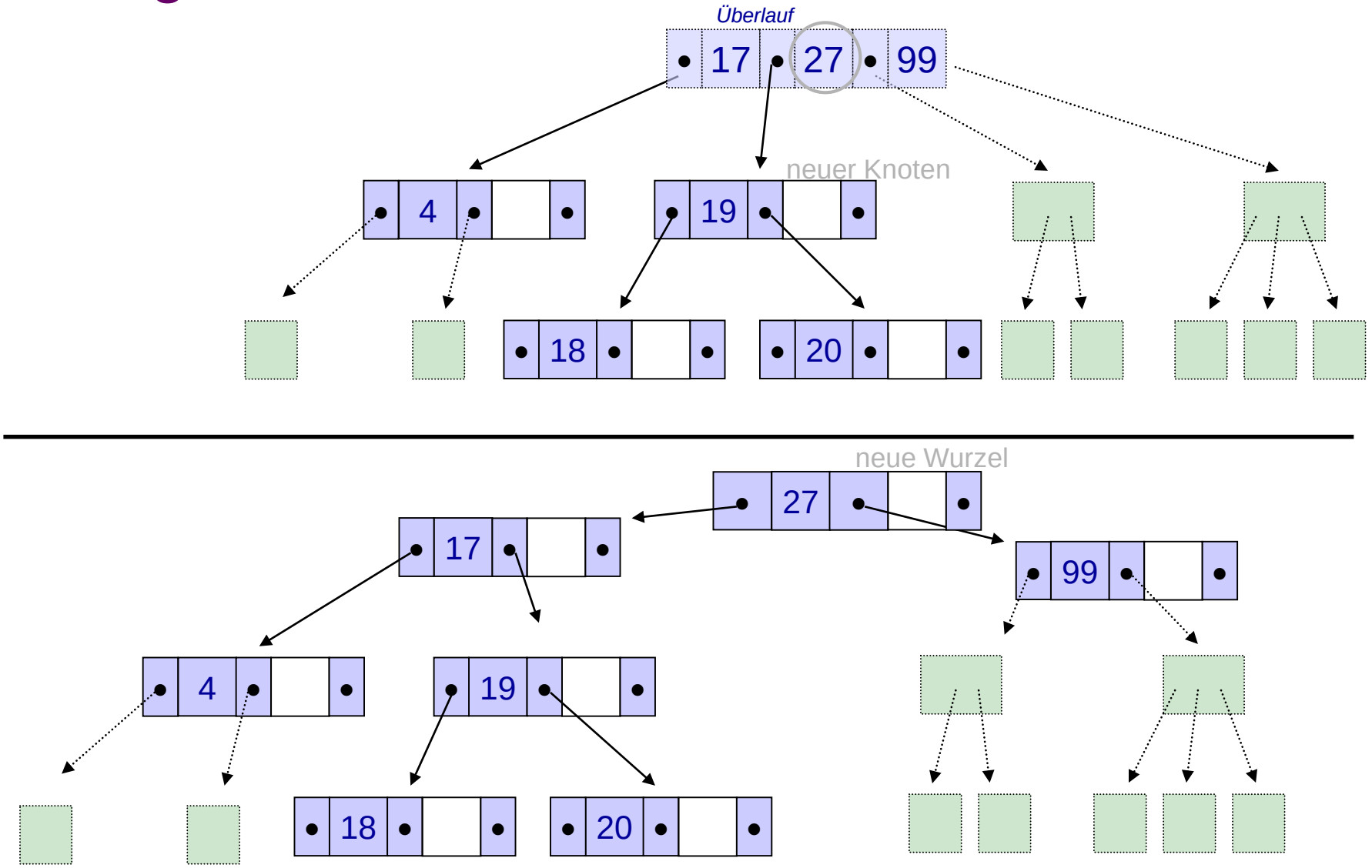
- Vorgänger kann auch überlaufen, ggf. rekursiv bis zur Wurzel weiter splitten
- Wenn die Wurzel überläuft
  - Wurzel teilen
  - *Mittlerer* Schlüssel x_{k+1} wird neue Wurzel mit zwei Nachfolgern

Einfügen in B-Bäumen

- Beispiel: k =1



Einfügen in B-Bäumen



Löschen in B-Bäumen

Algorithmus

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

Löschen in B-Bäumen: Unterlauf

- Unterlauf in einem Knoten / Blatt (nicht Wurzel)
  - Der Knoten N hat einen Nachbarn M, mit mehr als *k* Schlüssel beinhaltet
    - → Dann **Ausgleich** des N durch die Schlüssel x_i aus dem Nachbarknoten M unter Einbeziehung des Vorgängers
  - Der Knoten N hat einen Nachbarn M, der 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
- Sonderfall: N ist die Wurzel
  - Wurzel wird gelöscht, wenn diese keine Schlüssel mehr beinhaltet → Baum ist leer

Löschen in B-Bäumen: Verschmelzen

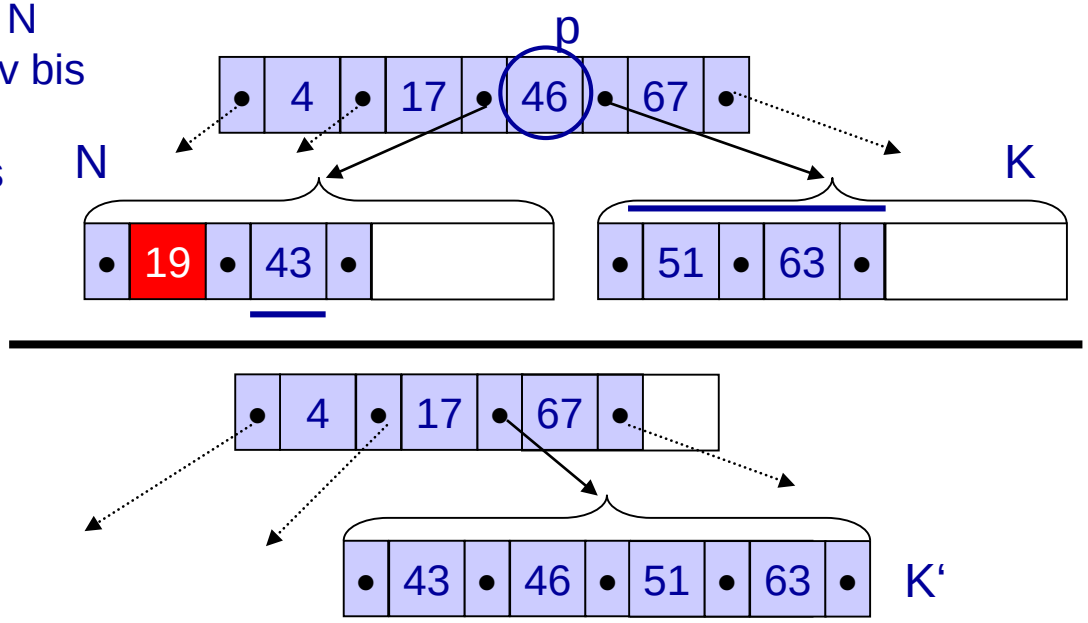
Aus dem Knoten N = (x_1 … x_k) soll der Schlüssel x_i entfernt werden

Verschmelzen für B-Baum der Ordnung *k*

- Sei K = (x'_1 … x'_k) ein Nachbarknoten mit **genau** k Schlüsseln
- O.B.d.A sei K rechts von N und *p* der Trennschlüssel im Vorgänger V

Bsp: k=2

- Verschmelze die Knoten N und K zu K‘, füge p 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

Löschen in B-Bäumen: Ausgleich

Aus dem Knoten N = (x_1 … x_k) soll der Schlüssel x_i entfernt werden

Ausgleich für B-Baum der Ordnung *k*

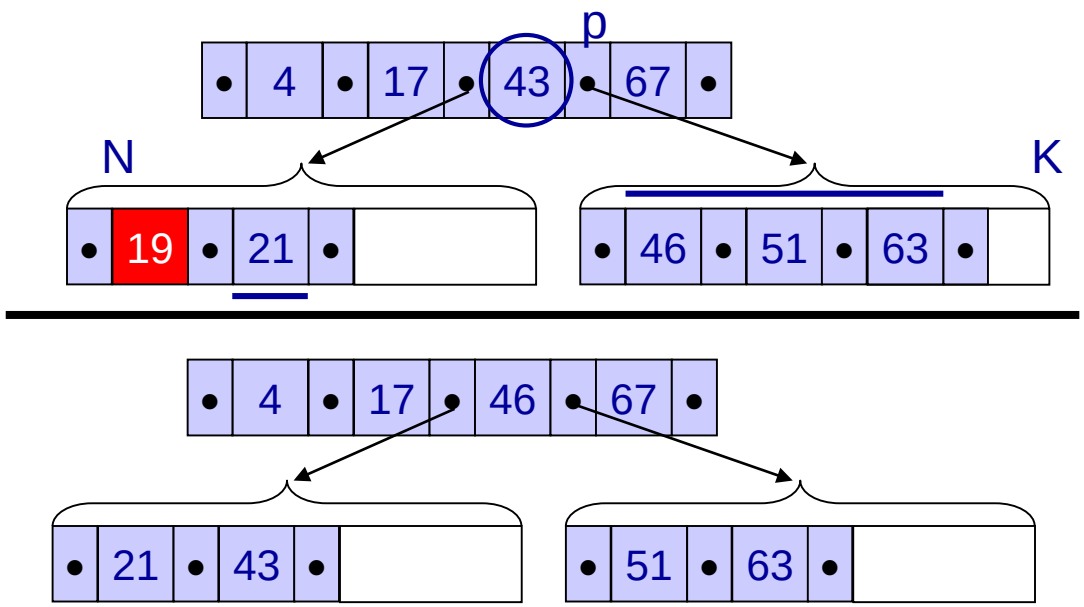
- Sei K = (x'_1 … x'_n) ein Nachbarknoten mit **mehr** als *k* Schlüsseln (*n*>*k*)
- O.B.d.A sei K rechts von N und *p* der Trennschlüssel im Vorgänger
- Verteile die Schlüssel x_1 … x_a, *p*, x'_1 … x'_n auf die Knoten K und N ersetze den Schlüssel *p* im Vorgänger durch den mittleren Schlüssel
- K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

Beispiel:

B-Baum mit *k* = 2

Lösche Schlüssel 19

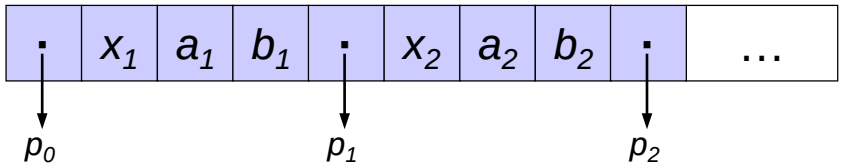
Ausgleich(21 , 43 , 46 , 51 , 63)



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.:

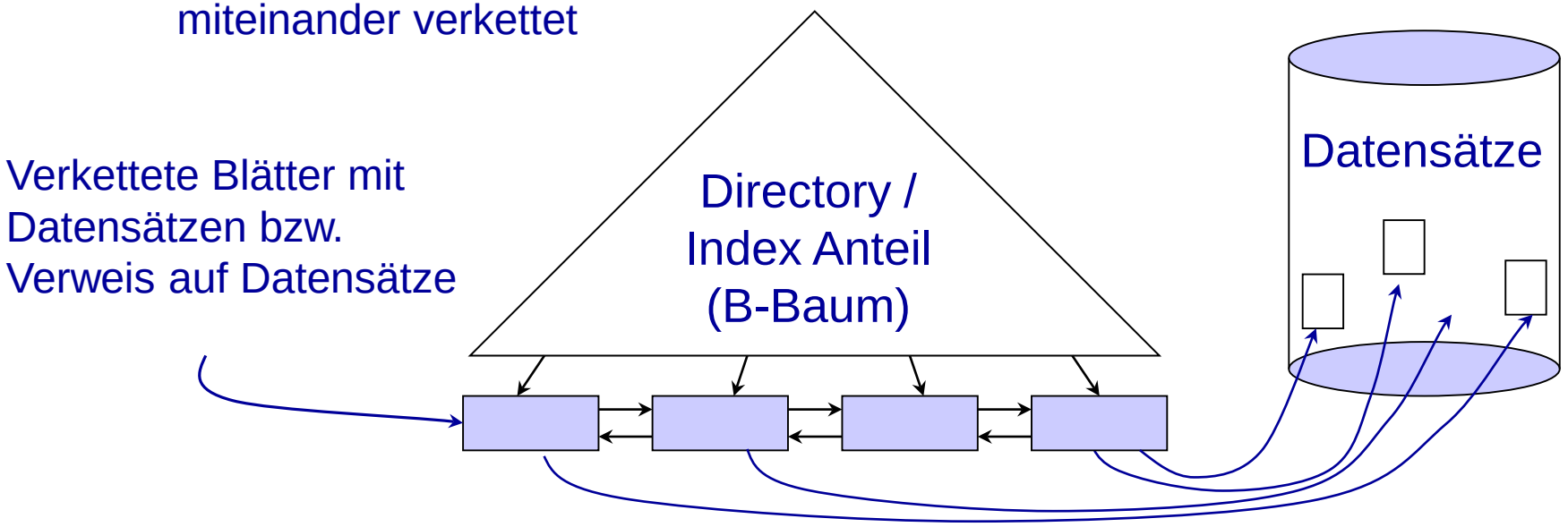


- Problem: Durch mehr Daten in den inneren Knoten (Seiten) sinkt der Verzweigungsgrad, dadurch steigt die Baumhöhe → kontraproduktiv

- 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

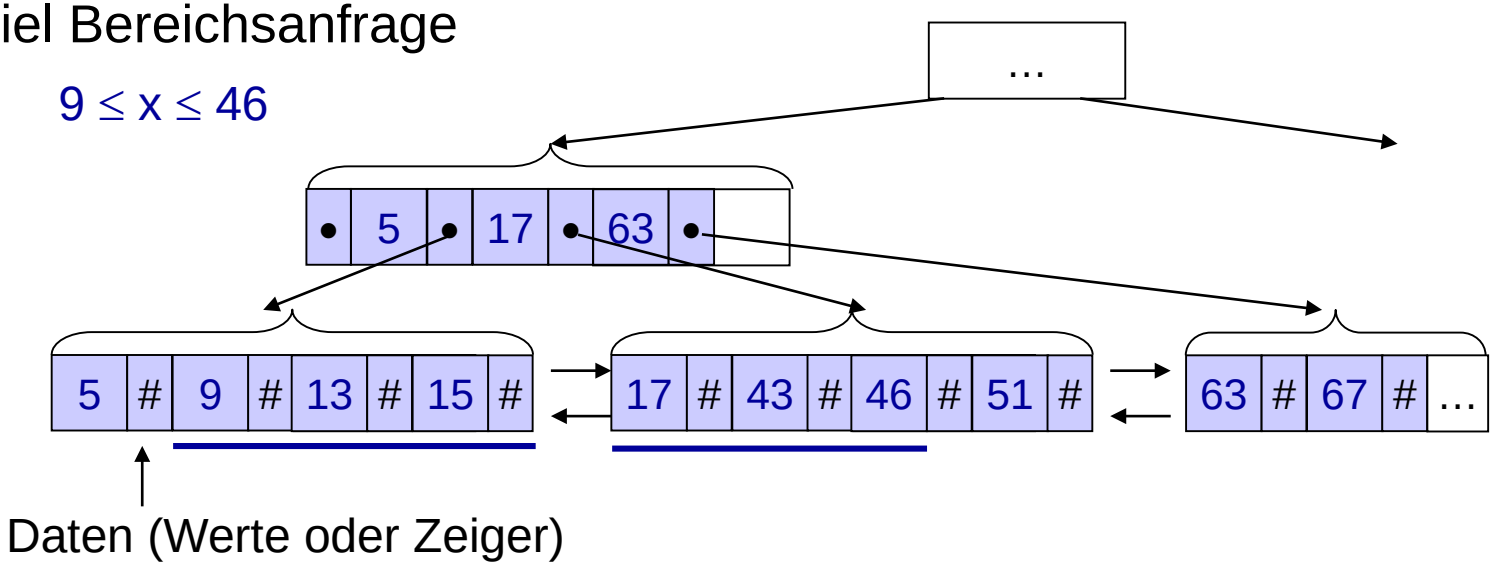


B^+-Bäume: Bereichsanfrage

- 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

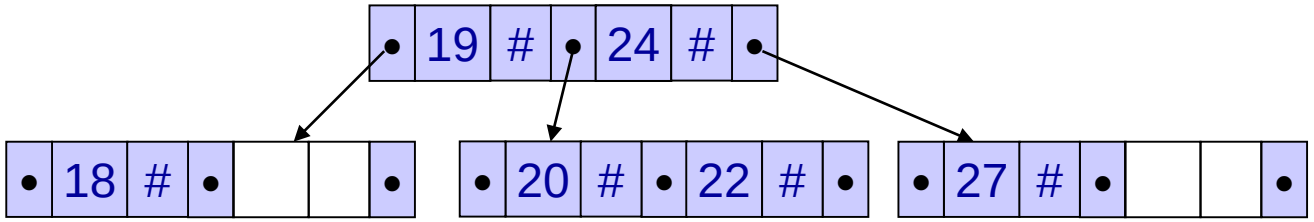
Beispiel Bereichsanfrage

$9 \leq x \leq 46$



Vergleich Verzweigungsgrad: B-Baum

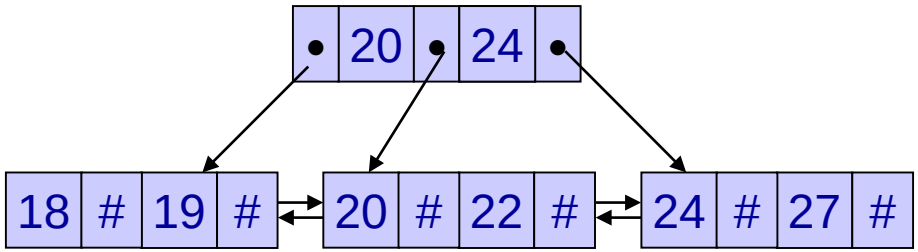
- Datensätze (#) stehen entweder direkt in den Knoten (ggf. bei Primärindex) oder sind ausgelagert und über Verweise referenziert (z.B. bei Sekundärindex)
- Bei der Größenabschätzung für B-Bäume sind diese Daten noch nicht berücksichtigt
- Beispiel: Seite 4096 B, Datenpointer 8 B, Knotenpointer 4 B,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



Vergleich Verzweigungsgrad: B$^+$-Baum

- Wie beim B-Baum: Seite 4096 B, Datenpointer 8 B, Knotenpointer 4 B, Schlüssel 16 B; 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: $(2k+1)*4 + 2k*16 \leq 4096 \Rightarrow 40k \leq 4092 \Rightarrow k \leq 102{,}3$
      - Also k maximal 102, der Verzweigungsgrad maximal 2k+1=205
    - Blätter: $2k*8 + 2k*16 + 2*4 \leq 4096 \Rightarrow$

      $48k \leq 4088 \Rightarrow k \leq 85.1$
      - k maximal 85
      - Füllgrad 170 maximal

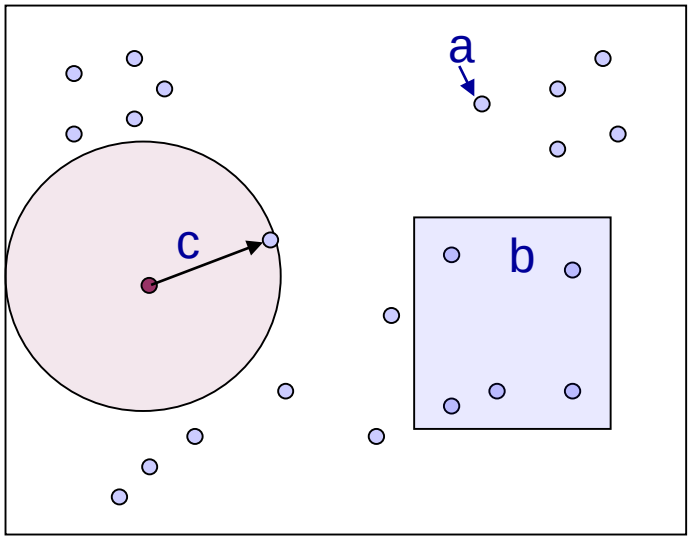


Räumliche Daten

Beispiel Punktdaten in 2D

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

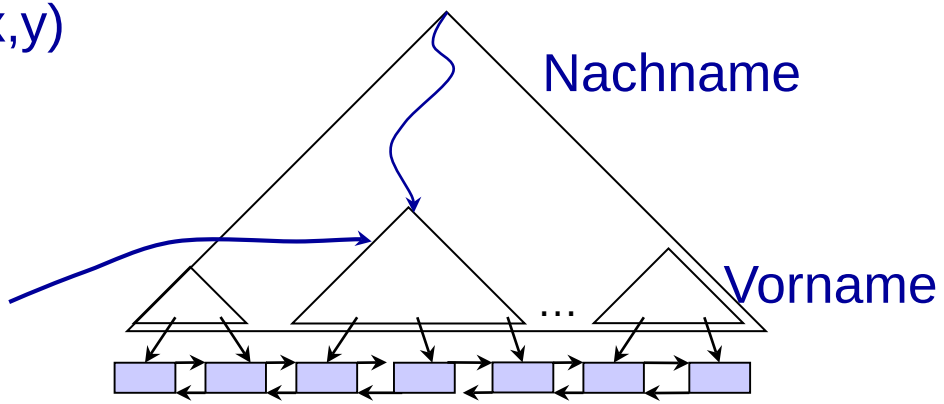
a) Punktanfragen
b) Alle Objekte in einem bestimmten Bereich
c) Nächster Nachbar



Zusammengesetzte Schlüssel (Composite)

Einsatzgebiet

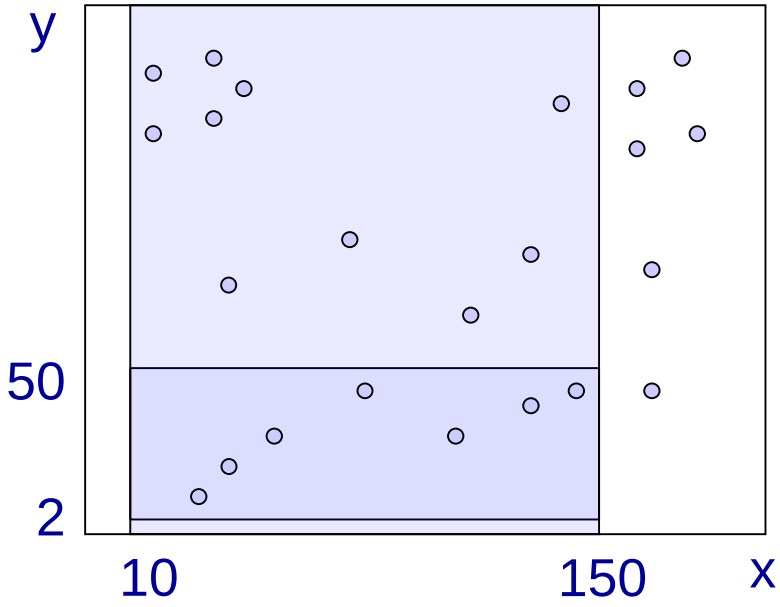
- Bäume können auf zusammengesetzte Schlüssel erweitert werden
  - Dabei wird zuerst nach der ersten und dann nach der zweiten Komponente sortiert (lexikografische Ordnung)
  - Beispiel: Nachname, Vorname
    Punktdaten (x,y)



Anwendung

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

Räumliche Daten

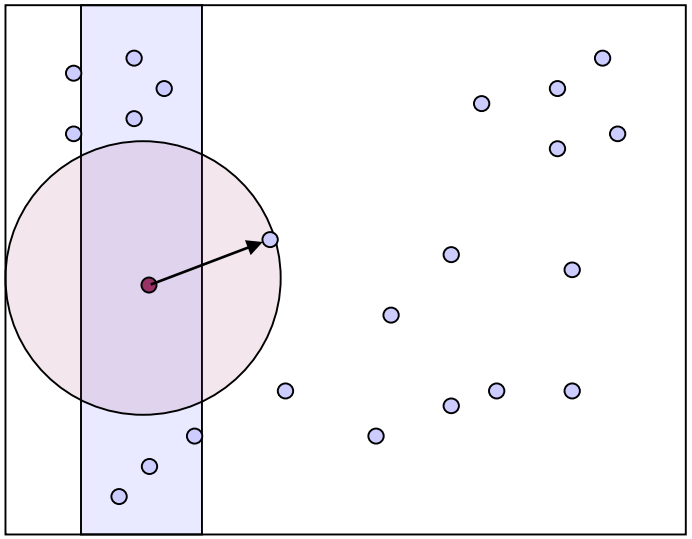


Beispiel: Bereichsanfrage

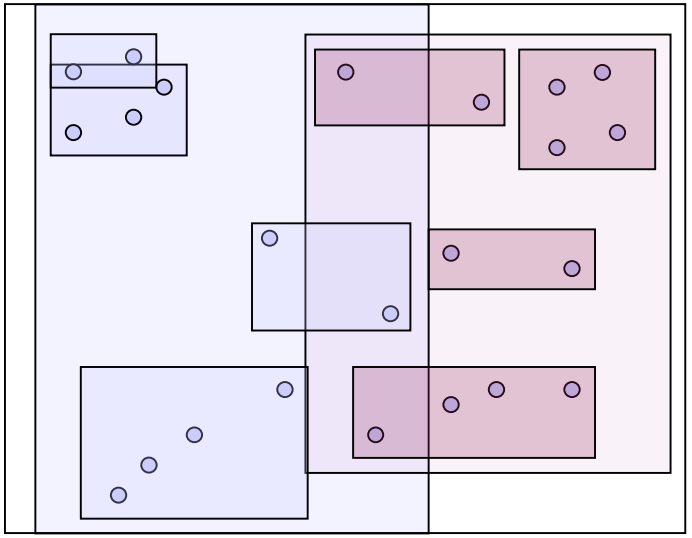
- Annahme: B-Baum auf (x,y)
  - x>10 AND x<150
  - y>2 AND y<50
  - Die Selektion nach x schränkt den Suchbereich nicht gut ein

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

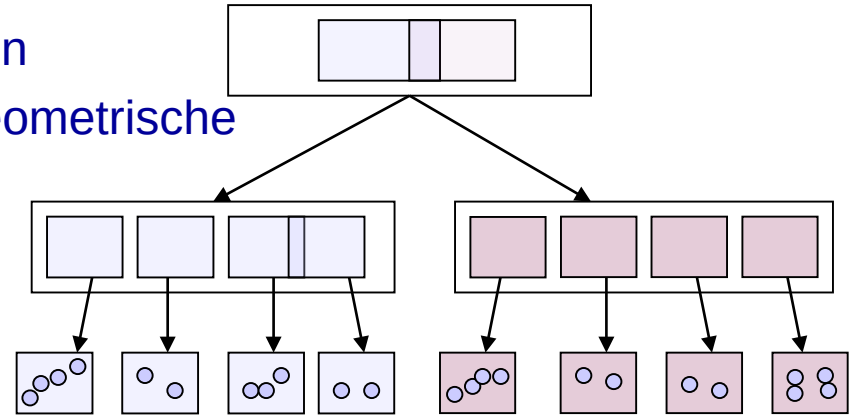


R-Baum



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



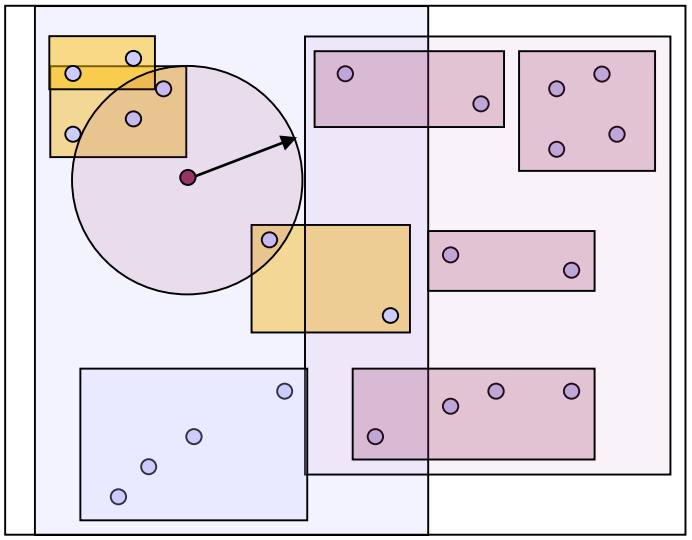
Aufbau

- Balance analog zum B-Baum
- Regionen können überlappen

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

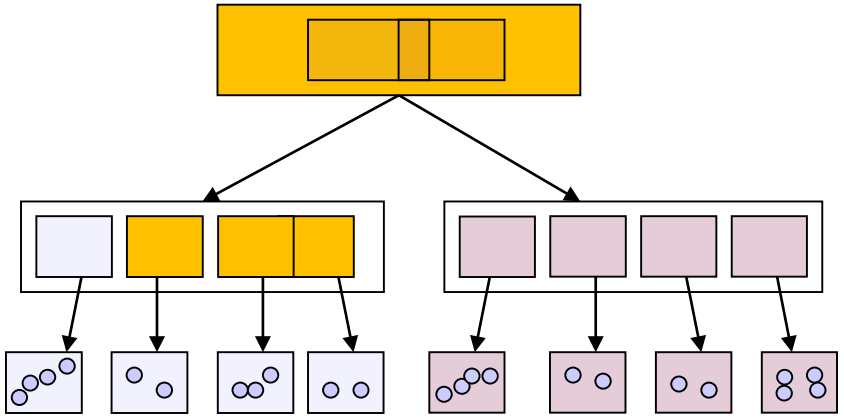
R-Baum: Bereichsanfragen

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

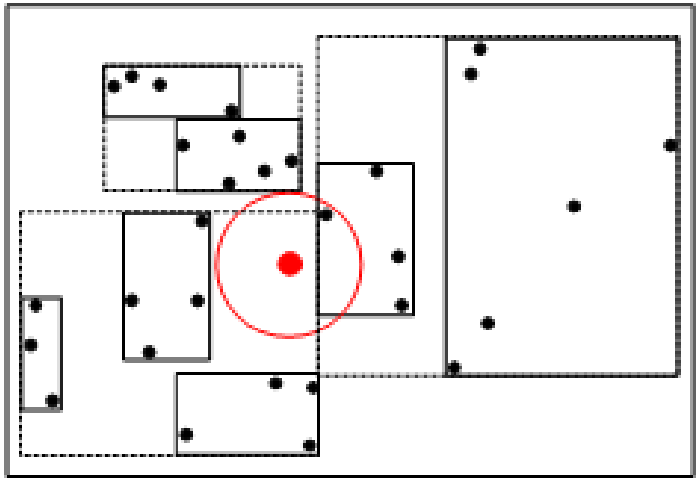


Eingabe: *Punkt p, Radius r*

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



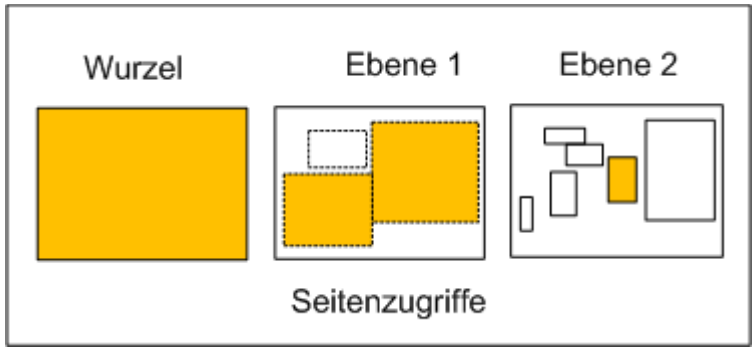
R-Baum: Nächste-Nachbarn-Anfragen



- 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^+-Baume)
  - 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

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