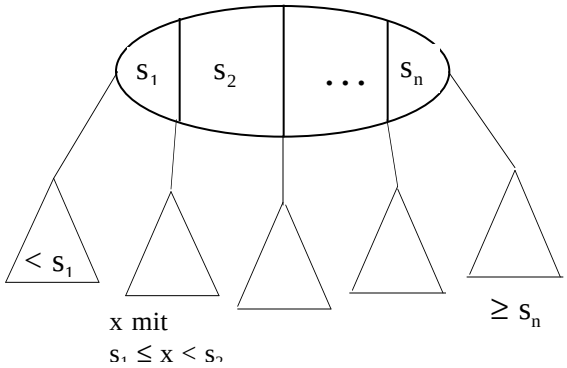
5. Suchen

Suchen = Tätigkeit, in einem vorgegebenen Datenbestand alle Objekte zu ermitteln, die eine best. Bedingung, das Suchkriterium, erfüllen und ggf. einen Verweis auf diese Objekte abzuliefern. Falls die Suche erfolglos war, hat ein Suchverfahren oftmals zugleich die Aufgabe, das neue Objekt in den Datenbestand einzufügen.
Zwei Verfahren, die auf sequentiellen Datenstrukturen arbeiten, haben wir in Abschnitt 4 besprochen: das sequentiellen Suchen auf allgemeinen Datenbeständen, das binäre Suchen auf sortierten Datenbeständen.
Sequentielle Datentypen eignen sich nur, wenn die Daten über einen längeren Zeitraum unverändert bleiben und Einfüge- oder Ausfügeoperationen kaum vorkommen. In praktischen Anwendungsfällen werden meist verkettete Speicherverfahren verwendet, bei denen Einfügeoperationen effizient möglich sind. Diese behandeln wir im folgenden

5.1. Suchbäume

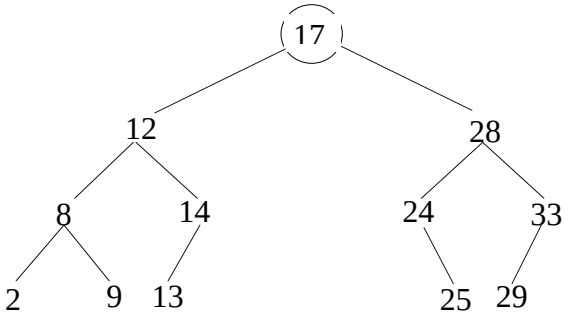
Ein Suchbaum ist ein DS, in der man Objekte effizient finden, einfügen und löschen kann.
Dazu benötigt man eine linear geordnete Menge M von Schlüsseln (=Merkmale, die Objekte eindeutig identifizieren). Der Suchbaum ist rekursiv anschaulich wie folgt definiert.

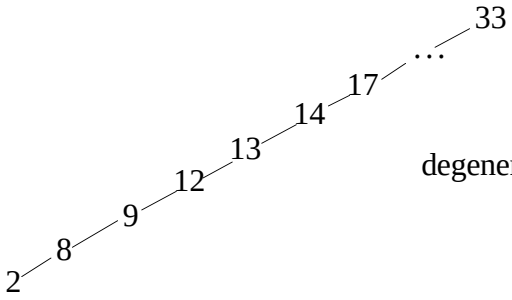


Für Internspeicher verwendet man häufig binäre Suchbäume, so daß jeder Knoten nur noch einen Schlüssel s enthält, und alle Knoten im linken Teilbaum Schlüssel < s und alle Knoten im rechten Teilbaum Schlüssel ≥s enthalten. Für Externspeicher verwendet man Suchbäume mit n = 128, 256 oder mehr Schlüsseln, sog. **B-Bäume**.

Def.: Sei M eine lineargeordnete Meng für binärer Suchbaum B = (K, A, S) für M ist ein geordneter, binärer Baum B = (K, A) mit einer Abb. S: K→M, so daß für jeden Knoten k ∈ K gilt:
(1) S(k) > S(u) für alle Knoten u im rechten Teilbaum von k
(2) S(k) ≤ S(u) für alle Knoten u im linken Teilbaum von k.

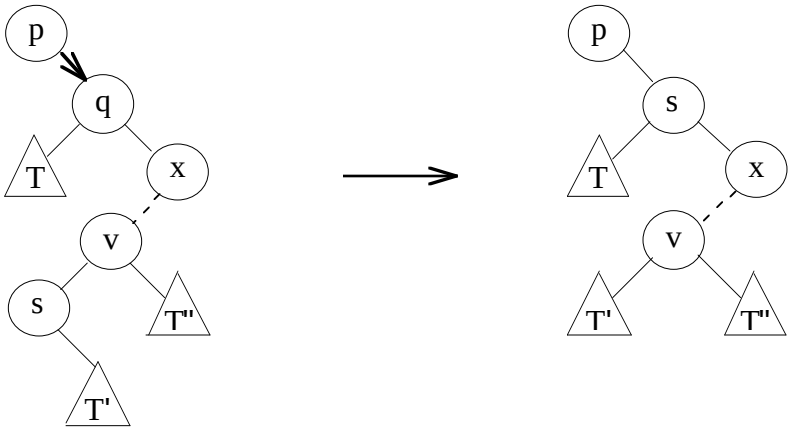
Bsp:





degenerierter Suchbaum

Operationen:
Suchen nach Schlüssel s: Beginn an der Wurzel s_0, Vergleich mit der Wurzel, Suche im linken oder rechten Teilbaum, falls $s < s_0$ bzw. $s \geq s_0$ und rekursiv weitermachen. Suche erfolglos, falls der entsprechende Sohn, zu dem man verzweigen soll, nicht existiert.
Einfügen: Neue Knoten s werden grundsätzlich als Blatt eingefügt. Vorgehensweise: Suche zunächst s; dabei gelangt man an ein Blatt, zu dem der Nachfolger nicht existiert; füge s als diesen Nachfolger ein.
Beispiel: 22
Löschen: Bis auf Sonderfälle aufwendiger: k sei zu löschender Knoten
Sonderfälle: 1) k ist Blatt. Lösche k
2) k hat keinen rechten Sohn. Lösche k
Neuer Sohn des Vaters von k wird der linke Sohn von k
3) dto., falls k keinen linken Sohn hat.
Allgemein: Suche in der inorder-Reihenfolge im rechten Teilbaum von k den ersten Knoten, der keinen linken Sohn besitzt, und füge ihn an die Stelle des zu entfernenden Kotens ein.
Beispiel: Lösche 28
Allgemein: Lösche q



s ist hier der inorder-Nachfolger von q

Laufzeit: Suchen + Einfügen + Löschen:
gut ausgeglichener Baum $\longrightarrow$ $O(\log_2 n)$
degenerierter Baum $\longrightarrow$ $O(n)$
Bem.: degenerierte Bäume sind sehr selten. Werden Schlüssel zufällig eingegeben, sind im Mittel $2 \ln n$ Vergleiche nötig
Speicherbedarf: O(1) bei nicht-rekursiver Implementierung.

Problem: Wie verhindert man degenerierte Bäume? Lösung: AVL-Bäume, siehe unten.

Algorithmus: Suche mit Argument K + Einfügen, falls nötig

```
Baum =
   record
      key: ...
      lt, rt: ↑ Baum
   end
```

```
w: Wurzel des Suchbaums
p: =w; q: =nil; B: =false;
while not B do
begin
    if p = nil then B: = true else
        begin
            if K = p↑. key the B: = true else
                begin
                    q: =p;
                    if K< p↑. key then p: = p↑. lt else p: = p↑. rt
                end
        end
end;
if p≠ nil then „p↑  ist der gesuchte Knoten“
else
    begin
        new ( r )
        r↑. key: = K;
        r↑. lt: = nil; r↑. rt: = nil;
        if K < q↑. key then q↑ = lt:= r else q↑. rt: = r
    end

Entfernen: des Knotens q↑

t: = q;
if t↑.rt = nil then q↑: = t↑. lt↑ else
    begin
        if t↑.lt = nil then q↑: = t↑.rt↑ else
        begin
            r: = t↑.rt;
            if r ↑.lt = nil then
            beginn
                r↑.lt: = t↑.lt; q: = r
            end else
            begin
                s: = r↑. lt;
                while s↑. lt ≠ nil do
                begin
                    r : = s; s : = r↑.lt
                end;
                s↑. lt: = t↑. lt; r↑.lt: = s↑. rt;
                s↑. rt: = t↑. rt; q↑: = s↑
            end
        end
    end;
dispose(t)
```

5. 2. Ausgeglichene Bäume

Offenbar hängt die Laufzeit bei Suchbäumen von der Länge des längsten Weges von der Wurzel zu einem Blatt ab. Es ist daher zweckmäßig, alle Knoten möglichst nah an der Wurzel anzuordnen. Für einen Baum mit n Knoten ist die Länge des längsten Weges im günstigsten Fall $[\log_2 (n+1)]$, was gleichzeitig eine obere Schranke für die Suchzeit liefert. Bei dynamischer Änderung des Baumes muß allerdings damit gerechnet werden, daß dies Optimalitätseigenschaft verloren geht, ein Baum also möglicherweise mehr und mehr degeneriert.

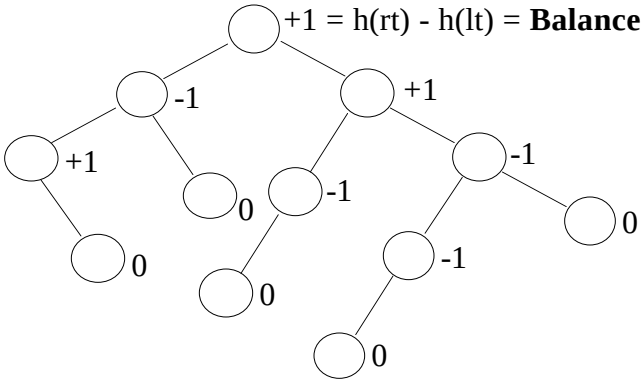
Naiver Zugang:
Man gleicht den Suchbaum nach jeder Veränderung vollständig aus. Konsequenz: erheblicher Effizienzverlust.

Offenbar kommt es nicht so sehr auf die vollständige, sondern auf eine „gewisse“ Ausgeglichenheit an.

Verbesserung:
Man geht über zu „fast“ ausgeglichenen Bäumen:

Def.: Ein binärer Baum heißt **ausgeglichener Baum** oder **AVL-Baum** (Adelson-Velskij und Landis), falls sich für jeden Knoten R die Höhen der beiden Teilbäume um höchstens 1 unterscheiden.

Bsp.:



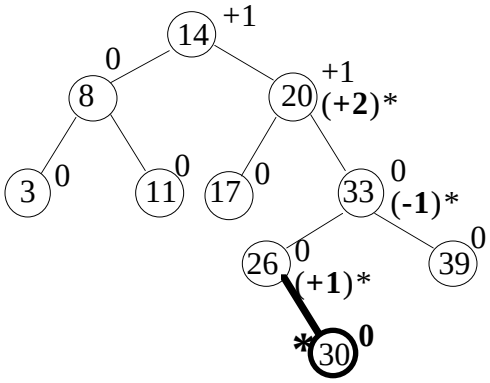
Operationen:

Suchen: wie bisher

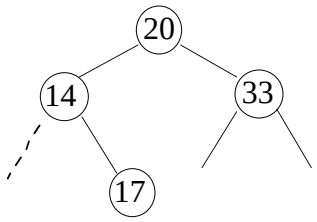
Einfügen: wie bisher, anschließend ausgleichen

Bsp. :

Einfüge 30



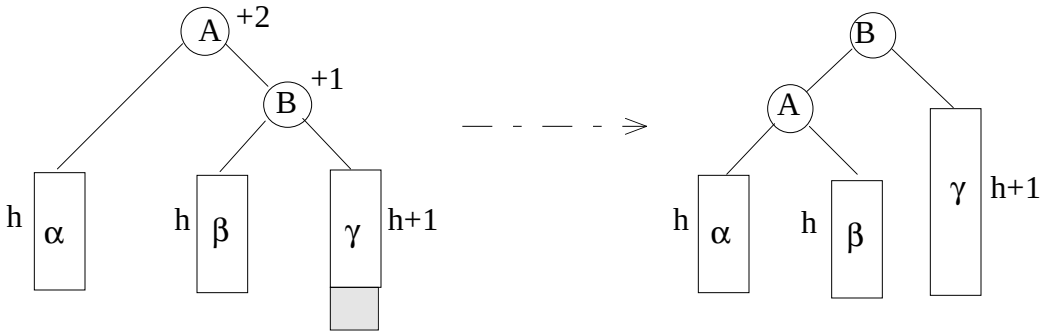
Rotation an Knoten (20)



Systematisch:

2 Fälle (bis auf Symmetrie):

1)

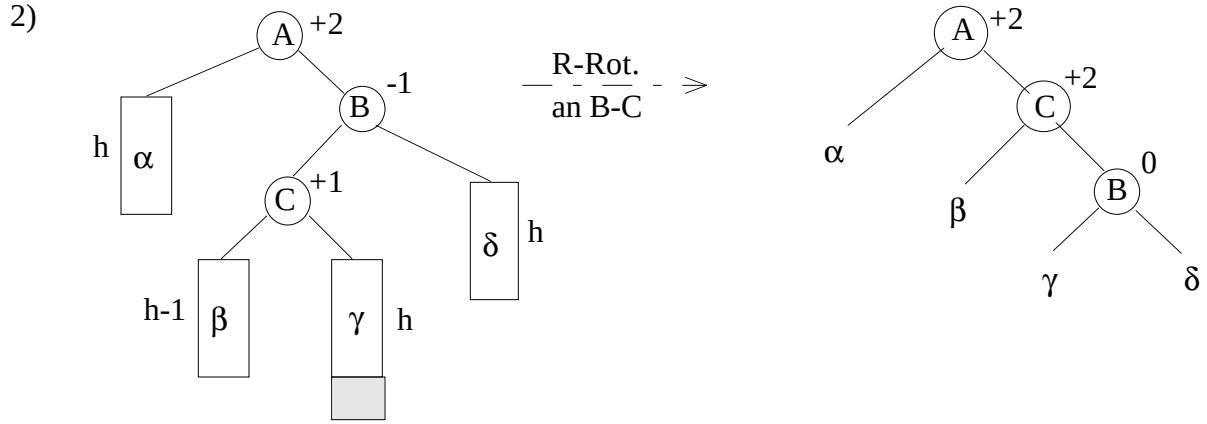


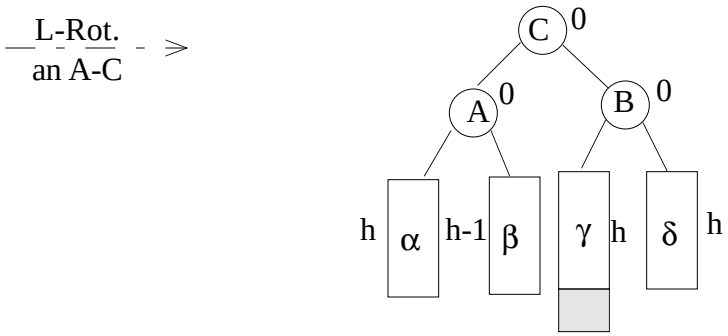
L - Rotation an Kante A-B

dgl. R - Rotation

Schreibweise: $\alpha\ (\beta\ \gamma) \rightarrow (\alpha\ \beta)\ \gamma$

2)





$\alpha\ ((\beta\ \gamma)\ \delta) \rightarrow$

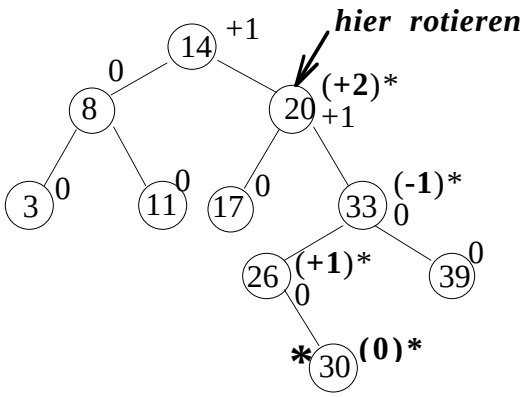
$\alpha\ (\beta\ (\gamma\ \delta))\ ^{R}\rightarrow$

$(\alpha\ \beta)\ (\gamma\ \delta)$

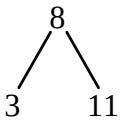
dgl. LR, LL, RR-Rotation

Implementierung:
Trage Schlüssel in Baum ein; Verfolge Weg zurück und verändere Balance bis Balance von ± 1 auf 0 abgeändert (=fertig) oder Balance auf ± 2 abgeändert wird (= rotieren).

Bsp. :



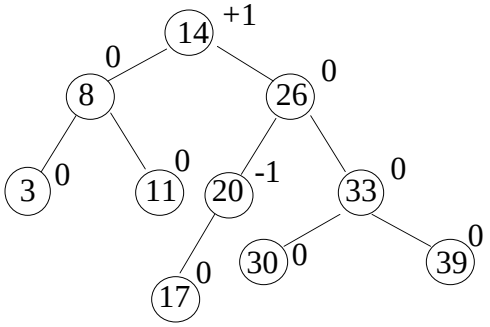
Fall 2 liegt vor: mit α =



β = 

$\gamma = 39$ $\delta = 17$, A = 20 , B = 33, C = 26

RL-Rot. ⇒



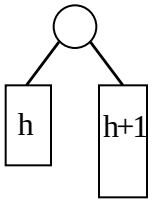
<u>Löschen:</u>
wie bei allg. Suchbäumen;
anschließend Korrektur der Balancen und Ausgleichsoperation vom Blatt aus;
Mehrere Ausgleichsoperationen sind möglich, maximal für jeden Knoten auf dem Weg von der Wurzel zur Ausfügestelle also O(log n). Da jede Rotation nur konstant viele Operationen erfordert, folgt

<u>Satz</u>
Suchen, Einfügen, Löschen in AVL-Bäumen erfordern im schlimmsten Fall $O(\log_2 n)$ Zeit.

Satz:
AVL-Bäume haben Höhe von O(log N)

Beweis:
Höhe 1: 2Blätter
Höhe 2: 3 Blätter

h + 2:



$\geq F(h+2) = F(h) + F(h+1)$. Fibonacci-Funktion

AVL-Baum der Höhe h hat wenigstens F(h+1) Blätter.
$F(h) \approx 0.72 \cdot 1.6^h$
[F (h) ist die $\frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{h+1}$ nächstgel. Zahl]
Anzahl der Blätter wächst oxponentiell mit h.
Anz. Blätter $N \geq F(h+1) \approx 1{,}17 \cdot 1.618^h$
$\Rightarrow h \leq 1.44 \cdot \log_2 N$

Ergebnis: 44 % Verlust an Höhe.

5.3 B-Bäume

Bisher nur Suchalgorithmen für Hauptspeicher. Wie sieht die Suche auf externen Speichern (Platten) aus? Übertrage Baumstrukturen auf Plattenspeicher (Hauptspeicheradressen→Plattenspeicheradressen)

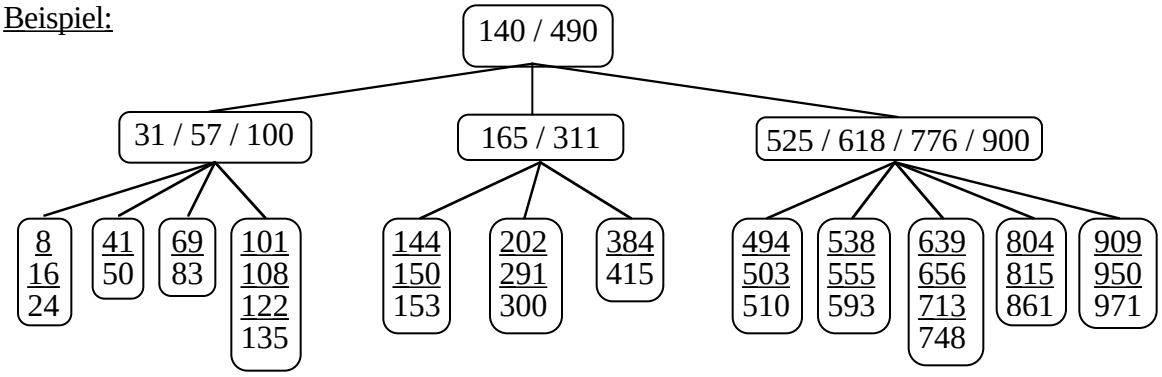
Problem: Die Zugriffszeit auf einen Knoten über einen Zeiger ist nicht mehr vernachlässigbar (nicht mehr nur eine Zeiteinheit). Konsequenz: Reduktion der Plattenzugriffe auf ein Mindestmaß und Nutzung der geblockten Übertragung und Abspeicherung. Konsequenz: Übertragung eines vollst. Teilbaums mit jedem Plattenzugriff.

Idee:
Zusammenfassung von Teilbäumen zu einem Super-Knoten und Realisierung einer Balanciertheit.

Def.:
Ein Baum heißt **B-Baum der Ordnung** m, falls die folgenden Eigenschaften gelten:

a) Jeder Knoten enthält ≤ 2m Schlüssel.
b) Jeder Knoten (mit Ausnahme der Wurzel) enthält mind. m Schlüssel.
c) Ein Knoten mit k Schlüsseln hat genau k + 1 Söhne oder keinen Sohn.
d) Alle Knoten, die keine Söhne haben, befinden sich auf gleichem Niveau.
e) Suchbaumeigenschaft: Sind $s_1, ..., s_k$ mit $m \leq k \leq 2m$ die Schlüssel eines Knotens x, dann sind alle Schlüssel des ersten Sohnes von x kleiner als s_1, alle Schlüssel des (k+1)-ten Sohnes größer als s_k und alle Schlüssel des i-ten Sohnes, $1<i<k+1$, größer als s_{i-1} und kleiner als s_i.

Beispiel:



Operationen:

Suchen: klar

Einfügen: grundsätzlich an den Blättern. Falls Überlauf, Knoten teilen, mittleren Schlüssel zum Vater reichen usw.

Bsp.: Einfügen 45 (einfach)
Einfügen 680 (mehrmals Aufspalten und Weiterreichen)

Löschen: aufwendiger

Prinzip:
- Anreichern von Knoten um Schlüssel aus dem Vater
- Verschmelzen von Knoten

Algorithmen für B-Bäume der Ordnung m = 1 (sog. 2-3-Bäume): s. Vorlesung.

Laufzeitanalyse:
B-Baum der Ordnung m mit n Knoten

a) Suchen: Auf wieviele Plattenblöcke muß im worst-case zugegriffen werden?
Niveau 1: ≥ 2 Knoten (Blöcke)
Niveau 2: $\geq 2\ (m+1)$
Niveau 3: $\geq 2\ (m+1)^2$
Niveau k: $\geq 2\ (m+1)^{k-1}$ Knoten
Andererseits $n \geq 2\ (m+1)^{k-1}$

$$\Rightarrow k \leq 1 + \log_{m+1}\left(\frac{n}{2}\right) = 0\ (\log_{m+1} n)$$
$$= [0\ (\log_2 m)]$$

Praxis: m = 200 - 2000.

b) Zeitraubende Operation ist Aufspalten von Knoten.
s: durchschnittl. Anzahl der Aufspaltungen pro Einfügung.
t: Gesamtzahl der Aufspaltungen bei Einfügung der Schlüssel $k_1, ..., k_n$ in leeren B-Baum

$$s = \frac{t}{n}$$

p: Anzahl der Blöcke in einem B-Baum
t = p-1

n_p: Mindesanzahl von Schlüsseln in einem B-Baum mit p Seiten

$n_p = 1 + (p-1) \cdot m \leq n$

$$\Rightarrow p \leq 1 + \frac{n-1}{m}$$

$$\Rightarrow s = \frac{t}{n} = \frac{p-1}{n} = \frac{1}{n}\left(\frac{n-1}{m}\right) < \frac{1}{m.}$$

$\Rightarrow$ Aufspaltungen „selten“ ($200 \leq m \leq 2000$).

Ergebnis:
Satz:
Suchen, Einfügen, Löschen in B-Bäumen der Ordnung m implementierbar mit 0 ($\log_m n$) Plattenzugriffen.

Optimierungen:
Statt Aufspalten eines Knotens beim Einfügen mit geringer Speicherausnutzung besser Umverteilung der Schlüssel auf Bruderknoten (wie beim Löschen). Erst wenn Bruderknoten voll ist, wird aufgespalten (sog.B*-Bäume).
Garantie: Ausnutzung von mind. 2/3 der Kapazität jedes Knotens (Ausnahme: Wurzel).

B*-Bäume

5.4 Hashing

Bisher: Gegeben ein Schlüssel s mit systematischer Suche nach s im Datenbestand

Idee: Errechne aus s die Position, an der sich das Objekt im Speicher befindet.

Genauer: Sei K eine Menge von Schlüsseln und A eine Menge von Adressen, unter denen die Objekte mit Schlüsseln aus K abgespeichert sind. Eine Funktion

$$h: K \rightarrow A$$

heißt **Hash-Funktion**. Die Suche nach dem Objekt mit Schlüssel k beschränkt sich auf die Berechnung von h (k).

Ziel: Beschreibe h mit möglichst einfachen arithm. Operationen.

Beispiel: K_1 = {Januar, ..., Dezember}
A = {0, ..., 16}
f: {A, ..., Z} $\rightarrow$ {1, ..., 26} ordne jedem Buchstaben seine Position im Alphabet zu.

Def.: $h_1: K_1 \rightarrow A$ durch
h_1(k) = (f (1. Buchst. vor k) + f (2. Buchst. vor k)) mod 17

Ergebnis:

Monat	J	F	M	A	M	J	J	A	S	O	N	D
h_1	11	11	14	0	14	14	14	5	7	9	12	9

Kollisionen

h_2 (k) = (f (2. Buchst.) + f (3. Buchst.)) mod 17 (1 Kollision)
h_3 (k) = (2 · f(1. Buchst.) + 2 · f (2. Buchst.) + f (3. Buchst.)) mod 17 (kollisionsfrei)

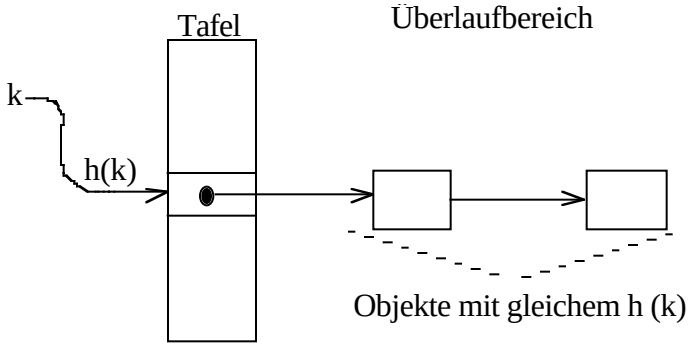
Vorgehensweise:

1) Bestimme Hash-Fkt. h, die einfach zu berechnen ist, und die K möglichst gleichmäßig auf A abbildet.
2) Festlegung einer Strategie für Kollisionen.

zu 1) • Schon bei kleinem K und großem A scheitert die Wahl einer zufälligen Funktion h: K → A.
Zur Begründung vgl. Geburtstagsparadoxon:
|K| = 23, |A| = 365 und f: K → A zufällig.
Dann ist die Wahrscheinlichkeit, daß
$\forall k \neq k', k, k' \in K$ gilt: f (k) ≠ f(k')
kleiner als 1/2. Die Wahrscheinlichkeit, daß von 23 Personen zwei am selben Tag Geburtstag haben, ist also recht groß.

- Tatsächlich ist die Zahl der möglichen Schlüssel meist um ein Vielfaches größer als die Zahl der Adressen:
z.B. K = {mögliche Nachnamen mit ≤ 10 Buchstaben} in einer Personaldatei
A = 1...1000 Personalnummern für höchstens 1000 Mitarbeiter
Folglich ist f üblicherweise nicht umkehrbar eindeutig (→ viele Kollisionen)
- Einfacher Ansatz
h (k) = k mod p (p Primzahl)
k nicht numerisch → k vorher numerisch machen

zu 2)
a) Version mit Überlaufbereich: direkte Verkettung



Implementierung:
type Tafel = array [A] of ↑Objekt;
var T: Tafel; je Objekt ein next-Zeiger

Suchen: h: = h (k); p:= T [h]; „sequentielle Suche nach k“;

Effizienz: gut bei kurzen Listen Gewinn von $\frac{|A|}{|K|}$ an Zeit, weil jede Liste im Durchschnitt $\frac{|A|}{|K|}$ Einträge hat.

Speicher: |K| Objekte und |A| + |K| Zeiger

b) Version ohne Überlaufbereich: offene Adressierung
b1) lineare Verschiebung:

Falls h (k) schon belegt ist, suche k nacheinander an den Stellen
h (k) + c, h (k) + 2c, ...
c Konstante teilerfremd zu p

b2) quadratische Verschiebung:

Falls h (k) schon belegt ist, suche k nacheinander an den Stellen
h (k) +1, h (k) +4, h (k) +9, ...

Beispiel: h für die Monatsnamen nachvollziehen.

Analyse:

Annahme: Einfügungen erfolgen an zufälligen Positionen. Jede der $\left(\frac{|A|}{|K|}\right)$ möglichen Konfigurationen von |A|-|K| freien und |K| besetzten Positionen ist gleichwahrscheinlich.

Aussage: (o. Beweis) Ist die Hash-Tabelle zu 90 % besetzt, werden im Mittel nur 2,56 Sondierungen nötig, um einen Schlüssel zu finden.