4.3 Mehrwegbäume als Suchbäume

Motivation

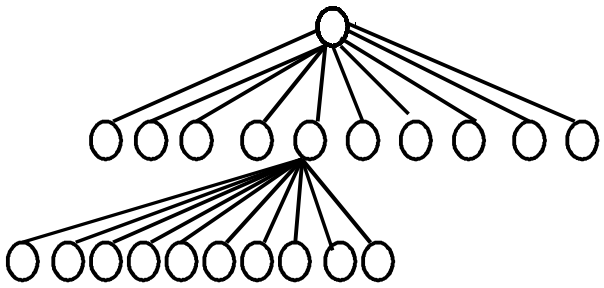
Die Länge eines Suchpfades im Baum ist proportional zur Höhe. Verwendet man statt Binärbäumen Mehrwegbäume, also Bäume von Grad $g \geq 2$, so ist die Höhe bei gleicher Knotenzahl N geringer und damit die Länge der Suchpfade kürzer. Dies ist besonders wichtig bei der Speicherung auf externen Speichern (Platten), da so die Anzahl der Plattenzugriffe minimiert werden kann.

Beispiel

Ein Binärbaum für 1000 Knoten hat 10 Ebenen, ein Baum vom Grad 10 für 1000 Knoten hat 4 Ebenen.

Baum vom Grade 10

-

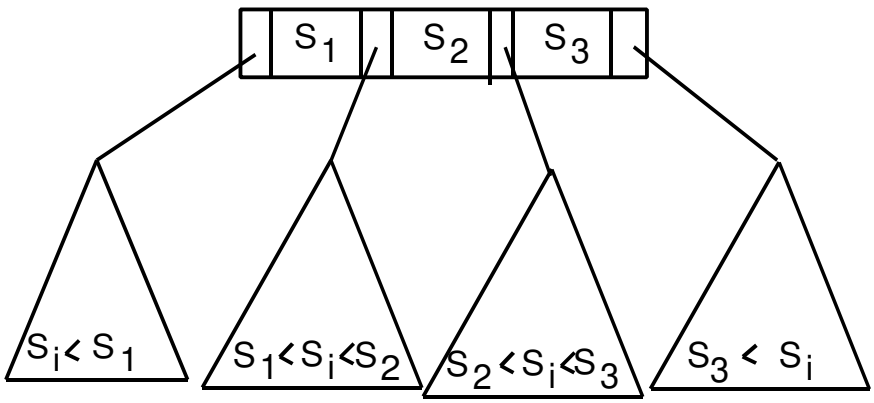


Jeder Knoten speichert 9 Schlüssel und 10 Zeiger.

Bäume vom Grade 4

Zusätzliche Forderung: Jeder Knoten muß mindestens 2 Söhne haben. Jeder Knoten hat also 2, 3 oder 4 Söhne. Deshalb heißen solche Bäume auch 2-3-4 Bäume.

Knotenstruktur



Suche in einem 2-3-4- Baum

Algorithmus

Beginne mit der Wurzel

Solange Knoten kein Blattknoten

Vergleiche Suchschlüssel mit den im aktuellen Knoten gespeicherten Schlüsselwerten.

Falls Schlüssel gefunden Ausgabe, stop

Falls (Knoten ist Blattknoten) Ausgabe "Nicht gefunden", stop

Falls Suchschlüssel < S_1 folge Zeiger 1

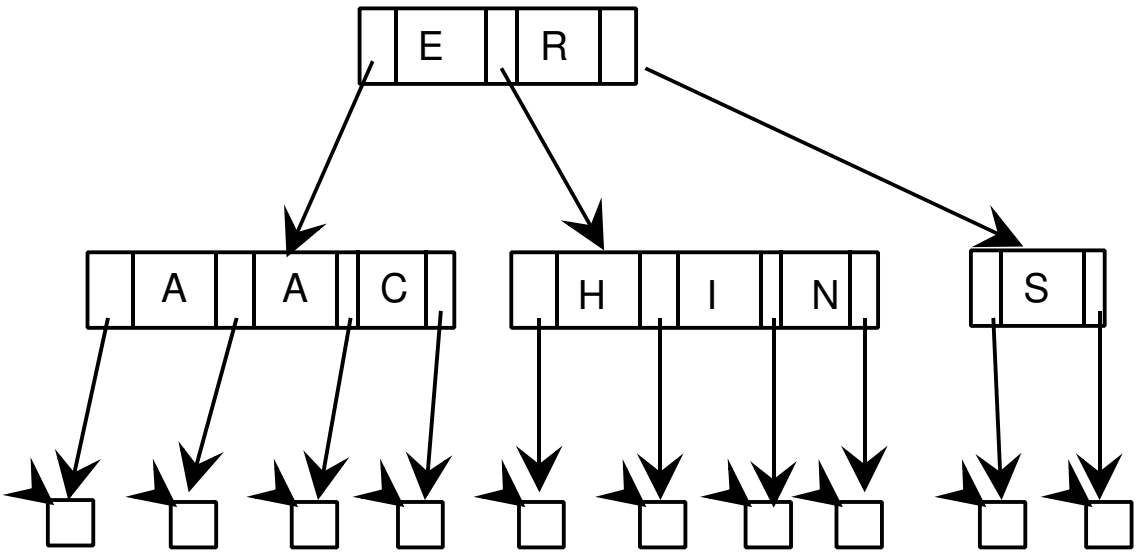
Falls Suchschlüssel < S_2 folge Zeiger 2

Falls Suchschlüssel < S_3 folge Zeiger 3

Folge Zeiger 4

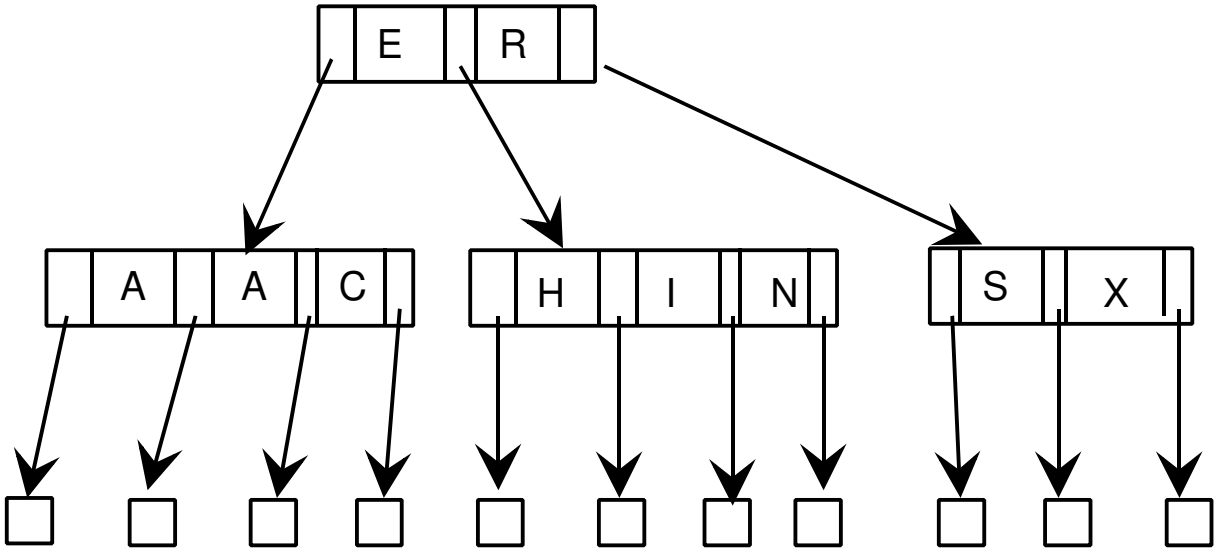
2-3-4 Bäume, Beispiel

Schlüsselwerte: ASEARCHIN



Einfügen in einem 2-er oder 3-er-Knoten einfach! Das Einfügen erfolgt ja immer in einen Blattknoten.

Beispiel: Einfügen von X



2-3-4-Bäume, Splitting

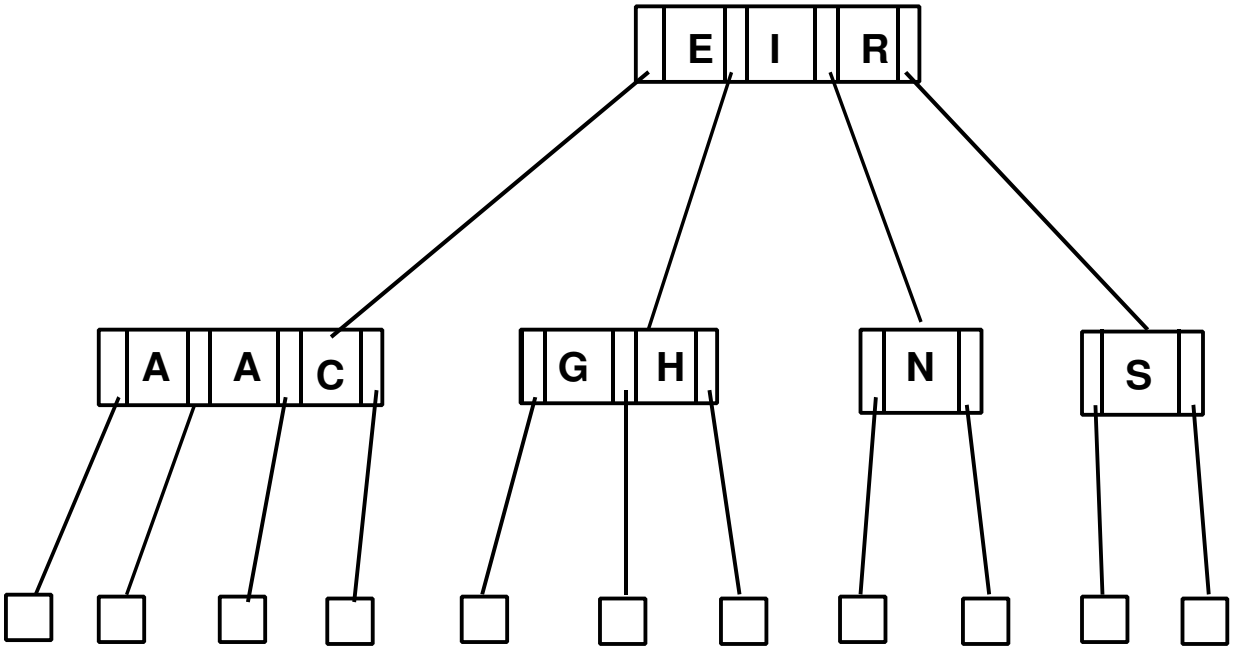
Wenn ein neuer Schlüssel in einen 4-er-Knoten eingefügt werden müßte, kann man

a) einen neuen Blattknoten auf einer tieferen Ebene einrichten, oder (besser)

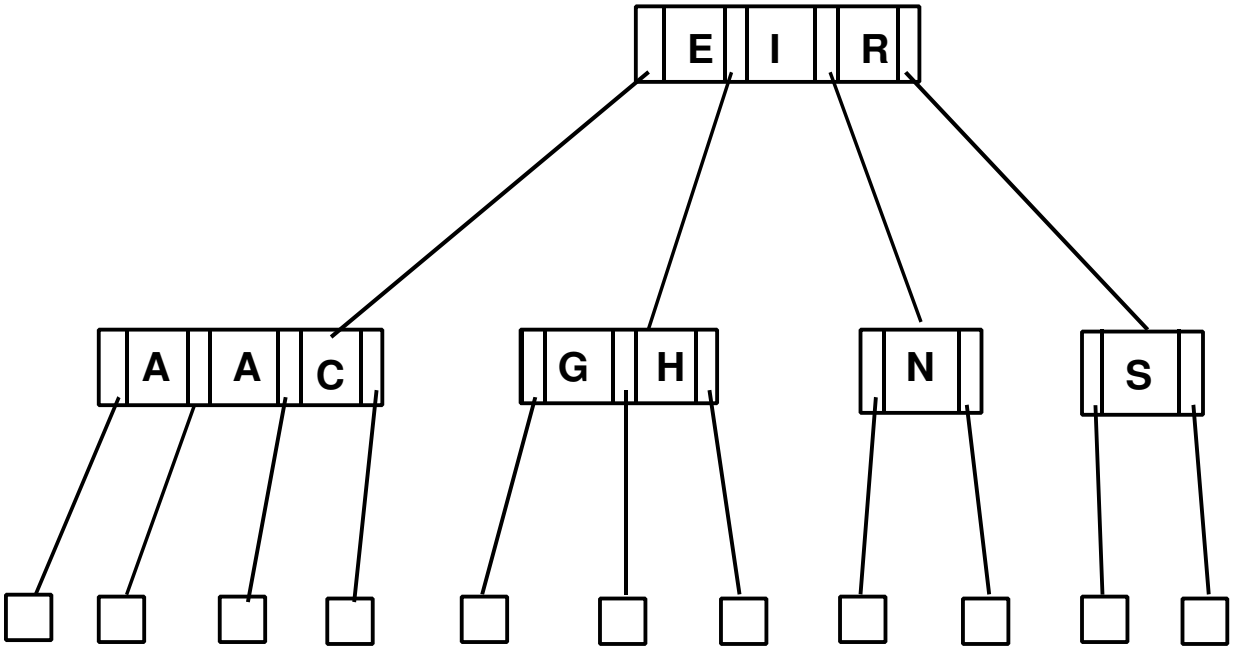
b) den Knoten splitten, einen seiner Schlüssel als Teiler in den Vaterknoten "hochziehen" und dort einen neuen Zeiger einrichten. Die beiden entstehenden Blattknoten haben dann zwei und einen Schlüssel.

Das Splitting muß eventuell bis zur Wurzel fortgesetzt werden.

2-3-4-Bäume, Beispiel für Splitting



Einfügen des Schlüssels G:



B-Bäume

Definition

Ein *B-Baum der Ordnung m* ist ein Baum mit folgenden Eigenschaften:

(1) Alle Blätter haben die gleiche Tiefe.

(2) Jeder Knoten mit Ausnahme der Wurzel und der Blätter hat wenigstens [*m*/2] Söhne.

(3) Die Wurzel hat wenigstens 2 Söhne.

(4) Jeder Knoten hat höchstens *m* Söhne.

(5) Jeder Knoten mit i Söhnen hat *i*-1 Schlüssel.

2-3-4-Bäume sind Binärbäume der Ordnung 4.

Anzahl der Schlüssel in einem Binärbaum

Um die Anzahl der in einem B-Baum mit gegebener Höhe *h* gespeicherten Schlüssel abzuschätzen, genügt es also, die Anzahl seiner Blätter abzuschätzen. Ein B-Baum der Ordnung *m* mit gegebener Höhe *h* hat die minimale Blattzahl, wenn seine Wurzel nur 2 und jeder andere innere Knoten nur [*m*/2] Söhne hat. Daher ist die minimale Blattzahl

$$N_{min} = 2 \cdot \left\lceil \frac{m}{2} \right\rceil^{h-1}.$$

Die Blattzahl wird maximal, wenn jeder innere Knoten die maximal mögliche Anzahl *m* von Söhnen hat. Daher ist die maximale Blattzahl

$$N_{max} = m^h.$$

Ist umgekehrt ein B-Baum mit N Schlüsseln gegeben, so hat er (N+1) Blätter. Hat der Baum die Höhe *h*, so muß gelten:

$$N_{min} = 2 \cdot \left\lceil \frac{m}{2} \right\rceil^{h-1} \leq (N+1) \leq m^h = N_{max}$$

Also

$$h \leq 1 + \log_{\lceil \frac{m}{2} \rceil} \left(\frac{N+1}{2} \right) \text{ und } h \geq \log_m (N+1).$$

Wir haben also wieder die für eine Klasse balancierter Bäume typische Eigenschaft, daß die Höhe eines B-Baumes logarithmisch in der Anzahl der gespeicherten Schlüssel beschränkt ist. Da die Ordnung m eines B-Baumes üblicherweise etwa b ei 100 bis 200 liegt, sind B-Bäume besonders niedrig. Ist

etwa $m = 199$, so haben B-Bäume mit bis zu 1999999 Schlüsseln höchstens die Höhe 4.

Struktur eines Knotens mit l Schlüsselwerten:

p_0	s_1	p_1	s_2	p_2	...	s_l	p_l

Für die Anordnung der Schlüsselwerte im B-Baum gilt:

Für jeden Knoten p mit l Schlüsseln $s_1,...,s_l$ und $(l+1)$ Söhnen $p_0,...,p_l$ ($\lceil m/2 \rceil \leq l+1 \leq m$) gilt: Für jedes i, $1 \leq i \leq l$, sind alle Schlüssel in $T_{p_{i-1}}$ kleiner als s_i, und s_i wiederum ist kleiner als alle Schlüssel in T_{p_i}.

Im folgenden werden wir stets annehmen, daß alle gespeicherten Schlüssel voneinander verschieden sind (es gibt keine Duplikate).

B-Bäume und externe Speicher

Besonders häufig werden B-Bäume zur Organisation von großen Datenmengen auf *externen Speichern* eingesetzt, typischerweise auf Magnetplatten. Dann entspricht ein Knoten im Baum einer Seite auf der Platte (= Block fester Größe; Einheit des Transfers im Dateisystem; typischerweise 512 Byte bis 4 Mbyte). Jeder Zugriff auf einen Knoten entspricht einem I/O-Vorgang.

Suche in einem B-Baum

Der Algorithmus zur Suche eines Schlüssels in einem B-Baum ergibt sich durch sinngemäße Erweiterung des Suchalgorithmus für 2-3-4-Bäume.

Die Suche *innerhalb* eines Knotens im B-Baum kann sequentiell erfolgen, aber auch durch binäre Suche, da die Schlüsselfelder sortiert sind und in der Regel feste Länge haben. Im Vergleich zu den Kosten eines Plattenzugriffs sind die internen Suchkosten vernachlässigbar.

Einfügen in einen B-Baum

1.) Suche im Baum nach dem neu einzufügenden Schlüssel. Die Suche endet erfolglos bei einem Blattknoten.
2.) Falls der Knoten noch nicht die maximale Anzahl von M-1 Schlüsseln hat, füge den neuen Schlüssel ein.

    Falls der Knoten bereits voll ist, splitte ihn in zwei Knoten und ziehe den Teilerschlüssel in den Vater des aktuellen Knotens hoch. Die beiden entstandenen Teilknoten haben je (M-1)/2 Schlüssel. Füge den neuen Schlüssel an der richtigen Stelle ein.

    Falls der Vaterknoten ebenfalls voll ist, wiederhole Schritt 2 rekursiv.

Wachstum eines B-Baumes, Beispiel

Gegeben sei ein B-Baum der Ordnung 3 wie folgt:

Loomis, S. 375 - Abb. 14.4

Einfügen des Schlüssels 22:

Loomis, S. 378, Abb. 14-7(a)

Einfügen des Schlüssels 41:

Abb. 14-7(b)

Einfügen der Schlüsseln 57 und 58

Loomis, Abb. 14-7(c-h)

Löschen nach den Einfüge-Operationen

Loomis, S. 389, Abb. 14-8

Löschen aus einem B-Baum

Zum *Entfernen* eines Schlüssels aus einem B-Baum der Ordnung *m* geht man umgekehrt vor. Man sucht den Schlüssel im Baum, entfernt ihn und verschmilzt gegebenenfalls einen Knoten mit einem Bruder, wenn er nach Entfernen eines Schlüssels *unterläuft*, also weniger als $\lceil m/2 \rceil - 1$ Schlüssel gespeichert hat. Ein Unterlauf der Wurzel, die ja nur einen Schlüssel gespeichert haben muß, bedeutet natürlich, daß die Wurzel keinen Schlüssel mehr speichert und nur noch einen einzigen Sohn hat. Man kann die Wurzel entfernen und den einzigen Sohn zur neuen Wurzel machen.

B-Bäume, Komplexität der Operationen

Ein B-Baum der Ordnung *m* mit N Schlüsseln hat maximal die Höhe $\log_{\lceil \frac{m}{2} \rceil}(N+1)$. Also gilt:

Die **Suche** berührt je einen Knoten pro Ebene, im schlimmsten Fall also $\log_{\lceil \frac{m}{2} \rceil}(N+1)$ Knoten.

Das **Einfügen** muß maximal pro Ebene ein Splitting durchführen; zur Suche kommen also im schlimmsten Fall nochmals $O(\log_{\lceil \frac{m}{2} \rceil}(N+1))$ Splitting-Operationen hinzu.

Das **Löschen** muß zusätzlich zur Suche maximal $O(\log_{\lceil \frac{m}{2} \rceil}(N+1))$ Verschmelzungen durchführen.

Alle Operationen im B-Baum sind $O(\log_{\lceil \frac{m}{2} \rceil}(N+1))$.

Speicherplatzausnutzung von B-Bäumen

Erwartungswerte für die in einem Knoten gespeicherte Schlüsselzahl, also für die *Speicherplatzausnutzung* eines B-Baumes der Ordnung *m* und weitere das Einfügeverfahren charakterisierende Parameter kann man analytisch berechnen. Es ergibt sich, daß man (unabhängig von *m*) eine Speicherplatzausnutzung von ln 2 = 69% erwarten kann, wenn man eine zufällig gewählte Folge von N Schlüsseln in den anfangs leeren B-Baum der Ordnung *m* einfügt, d.h. die Knoten des entstehenden B-Baumes sind nur zu gut 2/3 gefüllt.

Fügt man Schlüssel in auf-oder absteigend sortierter Reihenfolge in den anfangs leeren B-Baum ein, entstehen B-Bäume mit besonders schlechter Speicherplatzausnutzung. Die Knoten sind (in allen Fällen, in denen $N = 2 \cdot \left\lceil \frac{m}{2} \right\rceil^h$ ist) *minimal* gefüllt, d.h. die Wurzel hat nur einen und jeder andere innere Knoten nur $\left\lceil \frac{m}{2} \right\rceil$-1 Schlüssel.

Es gibt verschiedene Vorschläge, die schlechte Speicherplatzausnutzung von B-Bäumen zu verhindern. Man kann zunächst die unmittelbaren oder gar alle Brüder eines überlaufenden Knotens daraufhin untersuchen, ob man ihnen nicht Schlüssel abgeben kann, bevor man den Knoten teilt und den mittleren Schlüssel und damit eventuell auch das Überlaufproblem auf das nächsthöhere Niveau verschiebt. Andere Vorschläge zielen darauf ab, für eine Folge bereits sortierter Schlüssel B-Bäume nicht durch iteriertes Einfügen in den anfangs leeren Baum zu erzeugen, sondern möglichst optimale Anfangsstrukturen zu erzeugen in der Hoffnung, daß nachfolgende Einfügungen oder Entfernungen von Schlüsseln den Baum höchstens allmählich, d.h. für eine große Zahl solcher Operationen, stark vom Optimum abweichen lassen.

Beispiele B-Baum
aus Härder-Vorlesung
oder Loomis-Buch
oder MIS 631 a/b

Indexsequentieller Zugriff auf Plattendateien

Problem: Suchen/Einfügen/Löschen von Datensätzen per Direktzugriff in *sehr großen* Datenbeständen. Sequentielle Verarbeitung nicht realisierbar!

Idee: Anlegen eines Index zur Suche. Die Indexseiten selbst werden mit der Datei gespeichert. Die gesamte Datei ist in Seiten (Blöcke) fester Größe aufgeteilt, die direkt adressiert werden können (per Zylindernummer/Spurnummer/Blocknummer).

Die Größe e

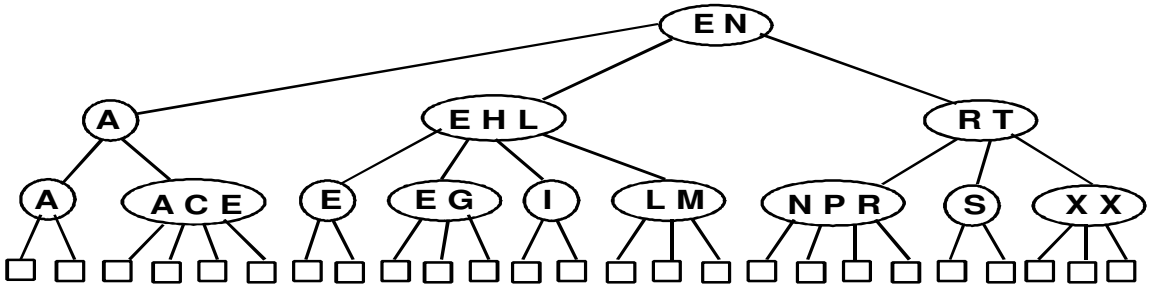
B-Bäume für indexsequentielle Dateien

ines Knotens im Baum entspricht der Größe eines Blockes der Datei.

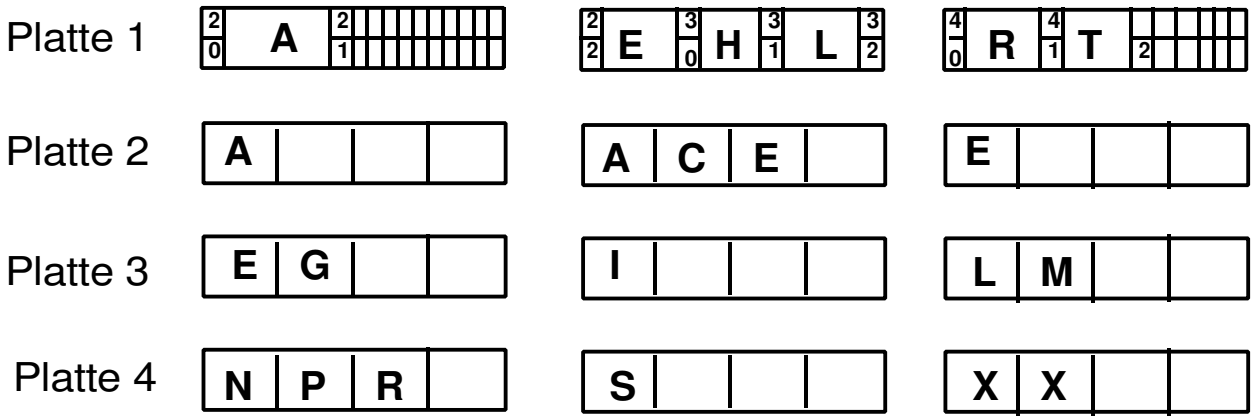
Beispiel für M = 4 und N = 13

EXTERNALSEARCHINGEXAMPLE

Baumstruktur



Plattendump



4.4 Digitale Suchbäume (Radix-Suche)

Idee:

Beim Durchlaufen eines digitalen Suchbaumes werden keine Vergleiche von ganzen Schlüsseln durchgeführt, sondern pro Schritt nur ein Vergleich des nächsten Bits oder Buchstaben des Schlüssels.

Annahme:

Die Schlüssel der Datensätze sind im Wertebereich zufällig verteilt; sie weisen keine internen Bitmuster auf, die der Gleichverteilung entgegenwirken würden.

Weiterhin: Keine gleichen Schlüssel im Datensatz.

4.4.1 Digitale Suchbäume im engeren Sinne (bitweise)

Idee

Auf jeder Eben wird genau ein Bit des Schlüssels verglichen.
Das Suchen ist identisch zu binären Suchbäumen, außer daß zur Bestimmung des Weges der Vergleich von einzelnen Bits verwendet wird.
Wird ein Schlüssel gefunden, so endet die Suche; andernfalls entscheidet das nächste Bit, in dem sich der Suchschlüssel vom aktuellen Knoten unterscheidet, ob links oder rechts weitergesucht wird.

```
int digitalsearch (int v)
 {
  struct node *x = head;
  int b = maxb;
  z->key = v;
  while (v !=v->key)
    x = (bits(v,b--,1)) ? x->r : x->1;
  return x_<infor;
 }
```

Anmerkung

Wie beim digitalen Sortieren extrahiert die Funktion
`bits(x,k,j)`
die j Bits, die k Stellen von rechts in x erscheinen.

Kommentare:

- Die Struktur des Digitalbaumes hängt von der Reihenfolge der Datenelemente beim Einfügen ab!
- Ein einmal eingefügtes Element wird später nicht mehr verschoben
- Für Suche und Einsetzen

  Zahl der Vergleiche $\approx \log_2 N$ im Mittel

  $= b$ im schlimmsten Fall

  b = Zahl der Bits im Schlüssel
- Der längste Weg im bitweisen Digitalbaum entspricht der längsten identischen Bitfolge für zwei Schlüssel, die in der Datenmenge vorkommen.

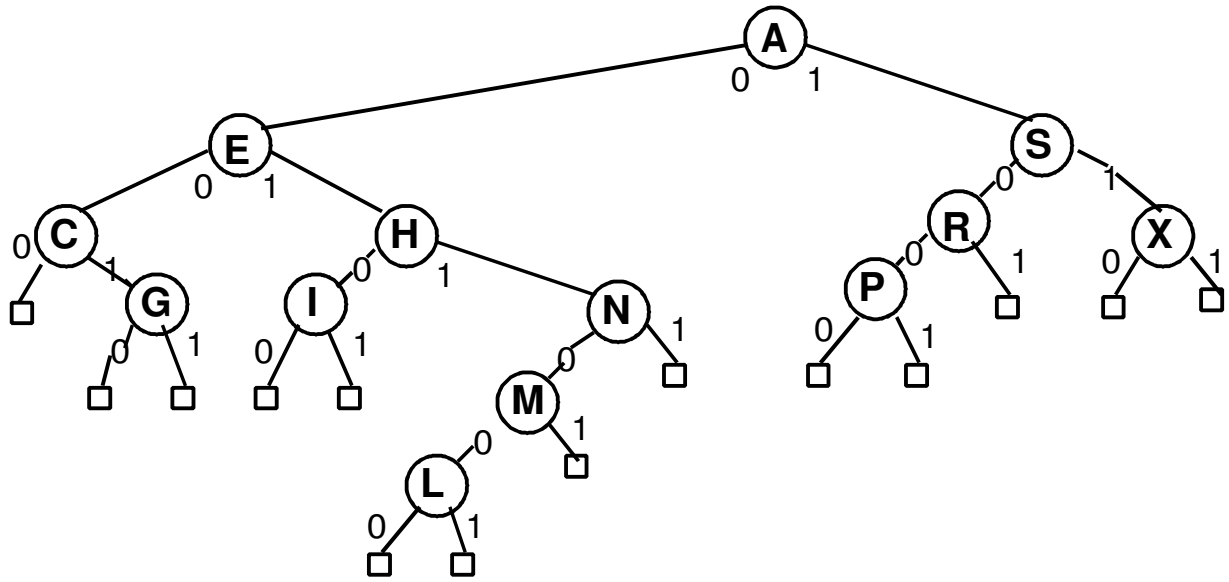
- Digitale Suchbäume sind i.a. sehr gut balanciert, wenn die Annahmen stimmen

Plausibilitätsbetrachtung:

Für jedes einzelne Bit im Schlüssel ist die Wahrscheinlichkeit für 0 und 1 gleich groß, es sind jeweils gleich viele Schlüssel im linken wie im rechten Teilbaum.

Digitale (bitweise) Suche, Beispiel

A	**00001**
S	**10011**
E	**00101**
R	**10010**
C	**000011**
H	**01000**
I	**01001**
N	**01110**
G	**00111**
X	**11000**
M	**01101**
P	**10000**
L	**01100**



Einfügen und Löschen in digitalen (bitweisen) Suchbäumen

Die Algorithmen für das Einfügen und Löschen ergeben sich aus diesen für Binärbäume durch entsprechende Modifikation, wie beim Suchen.

4.4.2 Tries

Name kommt von Re**trie**val, gesprochen wie "try".

Idee:

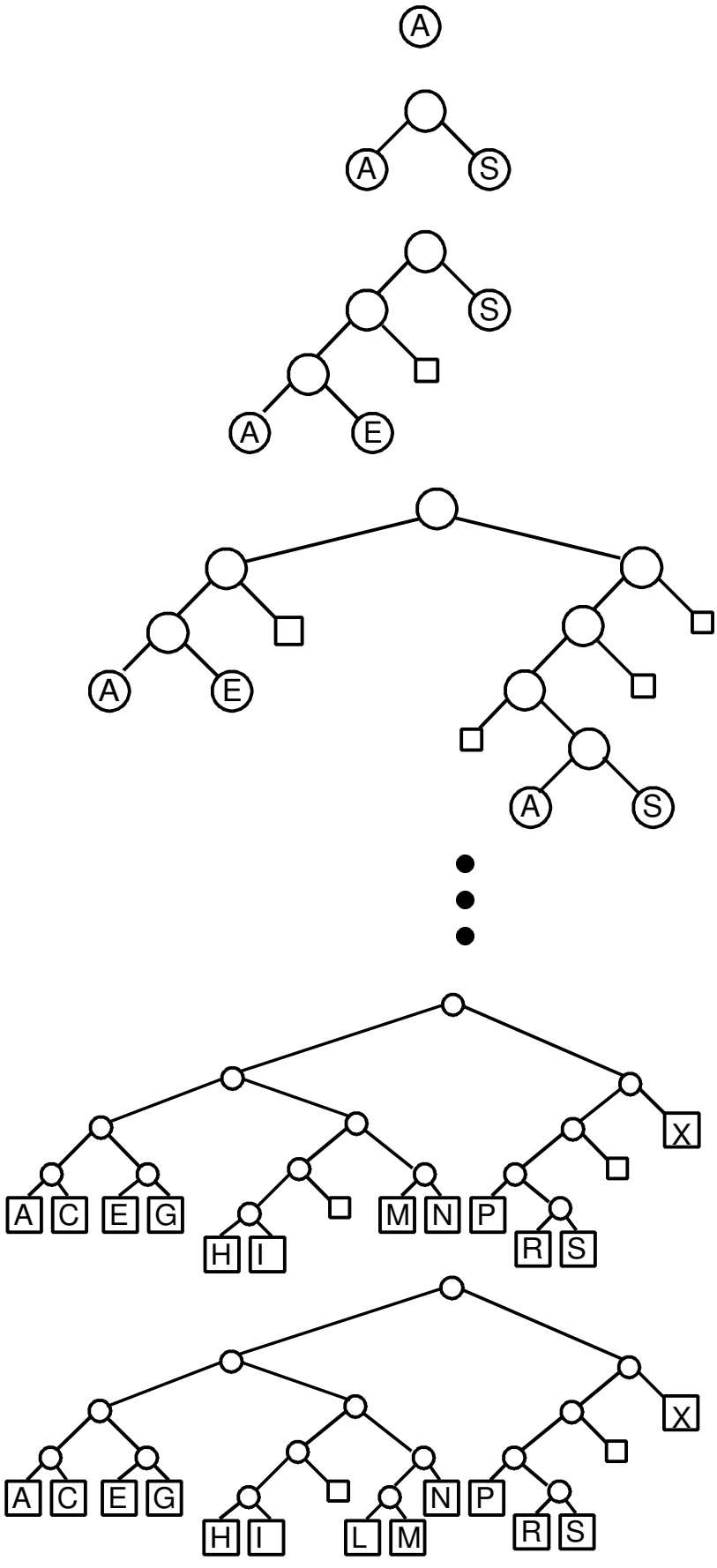
- Speichere die Schlüsselwerte selbst nur noch in Blattknoten
- Innere Knoten enthalten nur Zeiger auf Söhne.
- Die Position eines Elementes ist für eine gegebene Menge von Eingabedaten durch die Bitfolge seines Schlüssels eindeutig festgelegt. Die Reihenfolge des Einfügens spielt keine Rolle.
- Endet beim Einfügen der Weg bei einem Blattknoten, so muß der entsprechende Knoten durch einen internen Knoten mit den alten und dem neuen Knoten als Kinder ersetzt werden. Haben diese noch mehrere Bits gemeinsam, bevor das erste unterschiedliche Bit auftritt, so entstehen Zwischenknoten ohne Verzweigungen. Elemente können also durch weitere Einfügungen im Baum *verschoben werden*.

Algorithmen

Wie üblich bei den digitalen, bitweisen Suchbäumen, aber Speicherung der Schlüssel nur in Blattknoten. Eventuell müssen auf dem Pfad Knoten mit nur einem Sohn eingefügt bzw. auch wieder gelöscht werden.

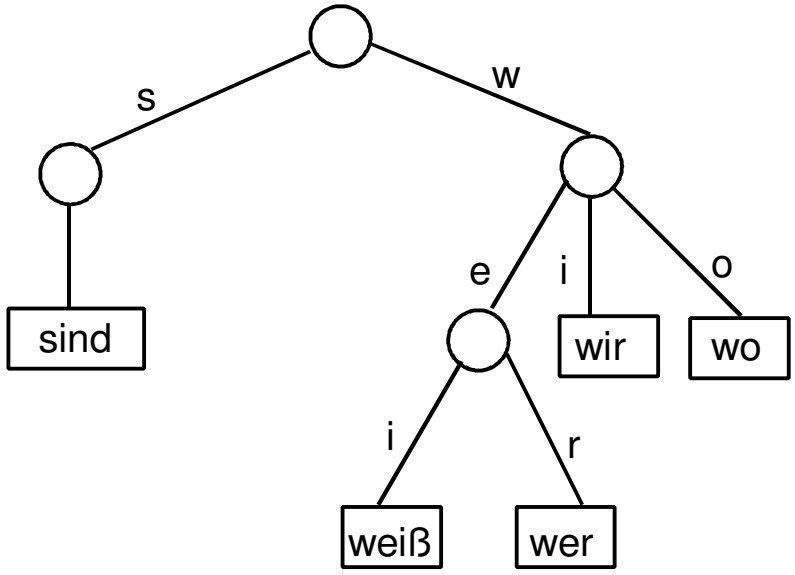
Tries, Beispiel 1

A	**00001**
S	**10011**
E	**00101**
R	**10010**
C	**000011**
H	**01000**
I	**01001**
N	**01110**
G	**00111**
X	**11000**
M	**01101**
P	**10000**
L	**01100**



Trie, Beispiel 2

Vergleich nicht mehr bitweise, sondern buchstabenweise.
Abspeicherung der Schlüsselmenge {wer, weiß, wo, wir, sind}



Anmerkung: Da es Schlüsselwerte nur in den Blättern geben darf, müssen alle Schlüssel gleiche Länge haben oder eventuell einzelne Schlüssel künstlich verlängert werden, z.B. durch Anhängen von Leerzeichen. Man betrachte die Einfügung von "wort" in obigem Beispiel.

Trie, Komplexität

- Ein Trie, der aus N zufälligen Schlüsseln mit b Bits erzeugt wurde, hat im Durchschnitt ≈ N/ln2 ≈ 1,44 N Knoten.
- Das Suchen und Einfügen im (zufällig erzeugten) Trie erfordert
  - im Mittel lg N Bitvergleiche
  - im schlimmsten Fall b Bitvergleiche.

Kommentare zu den digitalen Suchbäumen:

- Anders als bei Binärbäumen und digitalen, bitweisen Suchbäumen gibt es bei den Tries Knoten mit nur einem Sohn.
- Pro Such- und Einsetzvorgang nur noch ein Schlüsselvergleich, stückchenweise
- Erfordert effiziente Operationen zur Bitmanipulation in der Maschine.
- Bei zufälliger Schlüsselverteilung gute Balancierung, Komplexität im Mittel O(lg N), im schlimmsten Fall O(b).
- Nachteil: gängige Zeichencodes erzeugen ungleich verteilte Bitmuster!!

Zusammenhang zwischen Tries und Huffman-Codes

Man kann "den Spieß umdrehen" und eine Trie-Struktur zur Definition eines optimalen Zeichencodes verwenden:

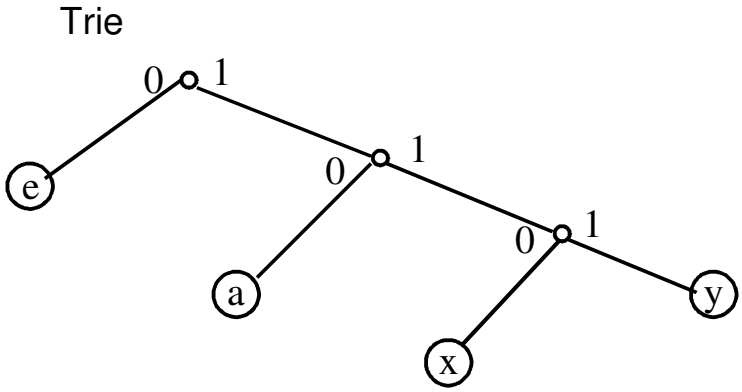
Gegeben sei ein Alphabet von Zeichen und ihre Häufigkeitsverteilung. Man finde einen Zeichencode mit variabler Länge (variabler Anzahl von Bits pro Zeichen) so, daß der Speicherbedarf minimiert wird. Ein solcher Code heißt **Huffman-Code**.

Beispiel:

Alphabet und Häufigkeiten:

Zeichen	%
e	50
a	30
x	10
y	10

Trie



Code-Tabelle:

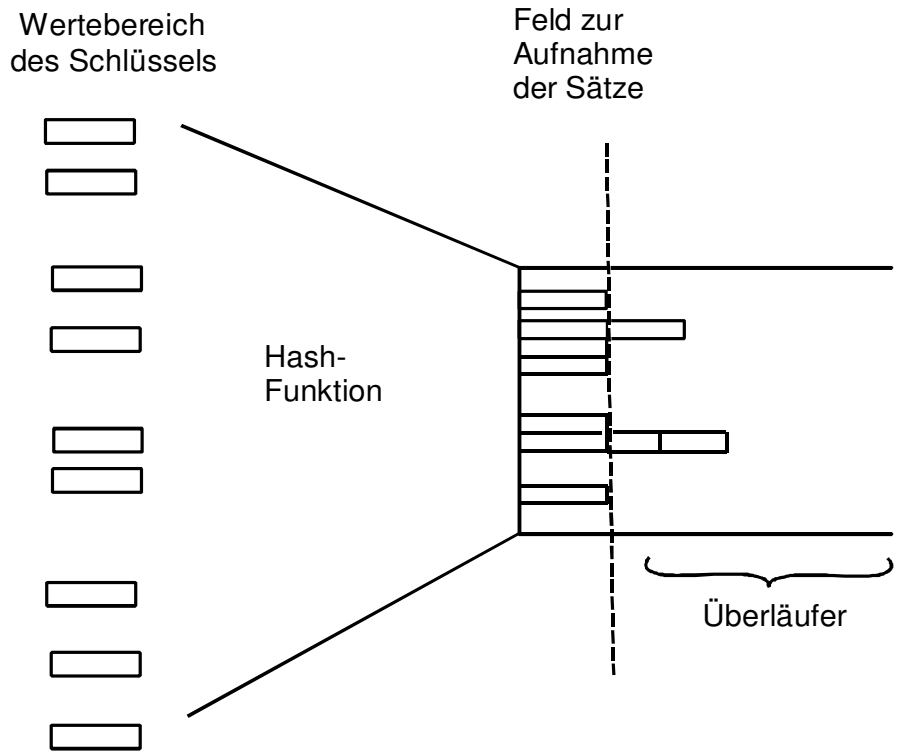
Zeichen	Code
e	0
a	10
x	110
y	111

4.5 Hashing

Motivation: Finde ein Verfahren, das den Zugriff auf Datensätze mit Aufwand O(1) ermöglicht.

Idee: Alle Elemente werden in einem Feld (array) fester Größe m gespeichert, wobei die Position im Feld aus dem Schlüssel des Elementes berechnet wird. Die Funktion zur Berechnung der Speicher-Position aus dem Schlüssel heißt *Hash-Funktion*.

Problem: Die Hash-Funktion bildet mehrere Schlüssel auf dieselbe Position ab, so daß Kollisionen auftreten können. Es muß eine Kollisionsbehandlung für die Überläufer durchgeführt werden.



4.5.1 Statisches Hashing

- Die Hash-Funktion bildet die Schlüssel auf den Indexbereich des Feldes ab.
- Sie muß einfach zu berechnen sein und soll möglichst wenige Überläufer erzeugen, d.h. eine gute Gleichverteilung der Schlüssel bewirken.

Grundidee:

- Interpretation des Schlüsselwertes als Zahl
- Berechnung des Restes modulo m : h (k) = k mod m

Diese Methode heißt *Divisions-Rest-Methode.*

Beispiel 1

Die Schlüssel sind die Zahlen von 1 bis 100 (statisch festgelegt) m = 100.
h(k) = k ist eine *perfekte Hash-Funktion*.

Beispiel 2:

Abspeichern der Anzahl der Werktage für alle Monate, Monatsnamen gegeben als Buchstabenfolge mit 3 Buchstaben: JAN, FEB, MAE, APR, MAI, JUN, JUL, AUG, SEP, OKT, NOV, DEZ.
Man suche eine Hash-Funktion h: {JAN, FEB,..., DEZ} -> {0, 1, ..., m} (mit $m \geq 11$), die die Monatsnamen auf Speicherpositionen abbildet. Auf der Speicherposition wird die Zahl der Werktage gespeichert.

Hash-Funktion z.B. (nr die "Nummer" eines Buchstabens im Alphabet):
h: $(Char)^3$ -> 0...14 mit $h(x_1x_2x_3) = (7 * nr(x_1) + 5 * nr(x_2)+2 * nr(x_3)) \bmod 15$.
Für die gegebene Schlüsselmenge JAN, FEB, ..., DEZ ist h sogar eine injektive Abbildung:

$x_1x_2x_3$ =	JAN	FEB	MAE	APR	MAI	JUN	JUL	AUG	SEP	OKT	NOV	DEZ
$h(x_1x_2x_3)$=	13	11	1	3	9	8	4	6	10	5	7	0

Dieses Beispiel bezieht sich auf eine *statische Schlüsselmenge*, d.h. die Schlüssel ändern sich nicht mehr durch Einfügen oder Löschen, wohl aber die eigentlichen Daten. Das entsprechende Feld hat 14 Positionen (Indizierung von 0 bis 13), von denen zwei (Indizes 2 und 12) nicht belegt sind. Wenn man nun zu einem Monatsnamen $x_1x_2x_3$ Daten suchen oder ändern will, dann berechnet man $h(x_1x_2x_3)$ und hat direkt die richtige Position im Speicher ermittelt. Man hat also nur eine Berechnung durchzuführen und findet die Daten mit einem einzigen Zugriff (0(1)), nimmt aber dafür einen Speicherplatzverlust für die nicht belegten Plätze des Feldes in Kauf.

Faltung für größere Schlüssel

Ist der Schlüssel zu groß, um die Divisions-Rest-Methode direkt anwenden zu können (d.h. größer als ein Maschinenwort), oder gibt es im Schlüssel eine Struktur, die einer Gleichverteilung entgegenwirken würde (z.B. Schlüssel aus binär codierten Dezimalzahlen), so wird eine *Faltung f* durchgeführt. Die Hash-Funktion wird dann auf den gefalteten Schlüssel angewendet: pos = h(f(k))

Beispiel: Schlüssel k = 124398033

$f_1(k) = 1+2+4+3+9+8+0+3+3 = 33$, oder

$f_2(k) = 124 + 398 + 033 = 555$

Berechnung mit Hilfe des Horner-Schemas

Lange Bitmuster können auch schrittweise mit dem Horner-Schema gefaltet werden.

Beispiel:
Gegeben sei ein Schlüssel mit folgendem Bitmuster:

10110001011001011001011000111101110001110101100101110 01

Das entspricht, als Ganzzahl interpretiert:

$22 * 32^{10} + 5 * 32^9 + 18 * 32^8 + 25 * 32^7 + 12 * 32^6 + 15 * 32^5 + 14 * 32^4 + 7 * 32^3 + 11 * 32^2 + 5 * 32^1 + 25$

Die Berechnung kann nun so erfolgen:

(((((((((22* 32+5)32+18)32+25)32+12)32+15)32+14)32+7)32+11)32+5)32+25

Programm

```
unsigned hash (char *v)
  {
    int h;
    for (h = 0; *v != '\0'; v++)
      h = (64*h + *v) % m;
    return h;
  }
```

Wahl des Divisors

Beim Divisions-Rest-Verfahren ist es wichtig, den Divisor geschickt zu wählen. Ist etwa m eine gerade Zahl, so ist $h(k)$ gerade, wenn k gerade ist; ist k ungerade, so ist auch $h(k)$ ungerade. Das ist für viele Schlüssel schlecht, z.B. dann, wenn die letzte Dualziffer einen Sachverhalt repräsentiert (0 = männlich, 1 = weiblich). Ebenfalls schlecht wäre die Wahl von m als Potenz der Basis des Zahlensystems, in dem Schlüssel dargestellt sind. So liefert etwa $m = 2^i$ die letzten i Bits der Dualdarstellung von k für $h(k)$; die restlichen Bits gehen überhaupt nicht in die Betrachtung ein. Ähnliche Argumente zeigen, daß m keine der Zahlen $r^j \pm j$, i und j kleine nichtnegative ganze Zahlen, teilen sollte, wobei r die Basis des Zahlensystems der Schlüssel ist.
Eine gute Wahl ist die, m als Primzahl zu wählen, die keine solche Zahl $r^j \pm j$ teilt. Diese Wahl hat sich in den meisten praktischen Fällen ausgezeichnet bewährt.

Kollisionsbehandlung

Wenn zwei Schlüssel auf dieselbe Adresse abgebildet werden, muß eine Kollisionsbehandlung durchgeführt werden.

Verfahren 1: **Anhängen als verkettete Liste**

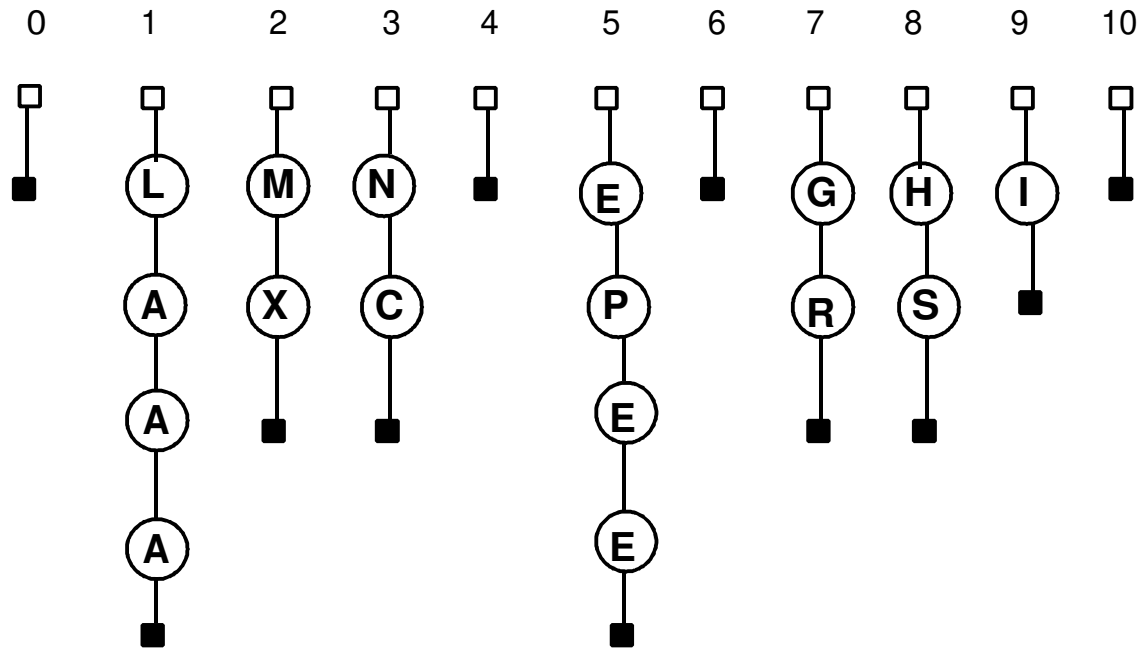
Alle Elemente, die auf dieselbe Adresse abgebildet werden, werden als verkettete Liste an das erste gespeicherte Element angehängt.

Beispiel

Gegebene Schlüsselemente und Hash-Funktion (m = 11)

Schlüssel:	**A**	**S**	**E**	**A**	**R**	**C**	**H**	**I**	**N**	**G**	**E**	**X**	**A**	**M**	**P**	**L**	**E**
Hash:	*1*	*8*	*5*	*1*	*7*	*3*	*8*	*9*	*3*	*7*	*5*	*2*	*1*	*2*	*5*	*1*	*5*

Repräsentation im Speicher



Komplexität beim statischen Hashing mit verketteten Überläufern

1.) Hash-Funktion, Auffinden des ersten Elements

O(1)

2.) Durchlaufen der Überläufer-Kette
Annahme: die verketteten Listen sind sortiert
Ist N >>m, so ist die durchschnittliche Länge der Listen N/m.
Die erfolglose Suche erfordert einen Aufwand von N/m.
Die erfolgreiche Suche erfordert einen Aufwand von N/(2·* m).

Vorteile:

Die separate Verkettung der Überläufer weist einige wesentliche Vorzüge gegenüber anderen Verfahren auf:
Der Erwartungswert und die Varianz für die Anzahl der betrachteten Einträge sind niedrig. Ein Belegungsfaktor von mehr als 1 ist möglich; d.h., selbst wenn die zu verwaltende Datenmenge mehr als vorgesehen wächst, so arbeitet das Verfahren noch korrekt. Echte Entfernungen von Einträgen sind möglich; eine Belastung von Speicher und Laufzeit durch als gelöscht markierte Einträge wird vermieden. Die direkte Verkettung der Überläufer eignet sich für den Einsatz mit Externspeichern; so könnte man etwa die Hashtabelle im Internspeicher halten und Datenseiten verketten.

Kollisionsbehandlung, Verfahren 2:
Weiterprobieren oder Sondierung (probing)

Im Gegensatz zur separaten Verwaltung von Überläufern in verketteten Listen werden die Überläufer auf freien Plätzen innerhalb der Hashtabelle untergebracht. Deshalb heißen Verfahren dieser Art auch offene Hashverfahren. Es wird eine *Sondierungsfolge* festgelegt, mit der von der belegten Hashadresse ausgehend weiter gesucht wird. Im einfachsten Fall ist dies eine lineare Suche im Speicher (linear probing).

Algorithmus Weiterprobieren

Sei *s(j,k)* eine Funktion von *j* und *k*, so daß
(h(k)-s(j,k)) mod *m* für *j* = 0,1,...,*m*-1,
eine Sondierungsfolge bildet, d.h. eine Permutation aller Hashadressen. Es sei stets noch mindestens ein Platz in der Hashtabelle t frei.

Suchen nach Schlüssel k: Beginne mit Hashadresse *i = h(k)*. Solange *k* nicht in *t[i]* gespeichert ist und *t[i]* nicht frei ist, suche weiter bei *i = (j(k)-s(j,k))* mod *m*, für aufsteigende Werte von *j*. Falls *t[i]* belegt ist, wurde *k* gefunden; sonst war die Suche erfolglos.
Einfügen eines Schlüssels k: Wir nehmen an, daß *k* nicht schon in *t* vorkommt (das kann durch eine Suche festgestellt werden). Beginne mit Hashadresse *i = h(k)*. Solange *t[i]* belegt ist, mache weiter bei *i = (h(k)-s(j,k)* mod *m*, für steigende Werte von *j*. Trage *k* bei *t[i]* ein.
Entfernen eines Schlüssels k: Suche nach Schlüssel *k*. Verläuft die Suche erfolgreich und ist *i* die Adresse, an der *k* gefunden wird, dann markiere *t[i]* als entfernt; sonst kommt *k* nicht in *t* vor und kann auch nicht entfernt werden.

Für *s(j,k) = j* erhält man das lineare Weiterprobieren (linear probing), also die Sondierungsfolge

$$h(k), h(k)-1, h(k)-2, ..., 0, m-1, ..., h(k)+1$$

Nachteil:

Es bilden sich schnell Häufungen (Cluster);die Daten sind in der Hash-Tabelle nicht mehr gleichverteilt, was die mittleren Längen der Sondierungsfolgen erhöht.

"Linear Probing", Programmbeispiel

Es sei s(j, k) = -j, also Weiterprobieren aufsteigend im Speicher, modulo m.

Algorithmus Linear-Probing

```
static struct node
  { char *key; int info; };
static struct node a[M+1];

hashinitialize()
  {
    int i;
    for (i = 0; i <=M; i++)
      { a[i].key = " "; a[i].info = -1; }
  }
hashinsert(char *v, int info)
  {
    int  x = hash(v);
    while (strcmp(" ",a[x].key))
         x = (x+1) % M;
    a[x].key = v; a[x].info = info;
}
```

Lineares Weiterprobieren, Beispiel 1

Beispiel:

Größe der Hashtabelle $m = 7$;
Schlüsselmenge = {0,1,...,500};
$h(k) = k \bmod m$;
$s(j,k) = j$ (lineares Sondieren).

Dann führen die Schlüssel 12, 53, 15, 2, 19, in dieser Reihenfolge in die leere Hashtabelle eingefügt, zu folgenden Situationen: Nach Einfügen von 12, 53:

	0	1	2	3	4	5	6
t:					53	12	

Nach Einfügen von 5: $h(5) = 5 \bmod 7 = 5$ ist belegt; der nächste Index der Sondierungsfolge ist 4, ebenfalls belegt; der nächste Index ist 3, nicht belegt:

	0	1	2	3	4	5	6
t:				5	53	12	

Nach Einfügen von 15, 2, 19, Sondierungsfolge 5-4-3-2-1-0:

	0	1	2	3	4	5	6
t:	19	15	2	5	53	12	

Alternativen: andere Sondierungsfolgen, z.B. quadratisches Sondieren, oder Sondierungsfolgen in Abhängigkeit vom Schlüssel *k*. Diese vermeiden die Bildung von Häufungen (clusters).

Kollisionsbehandlung, Verfahren 3: Double Hashing:

Idee:

Im Kollisionsfall wird eine schlüsselabhängige Schrittweite $h_2(k)$ verwendet. h_2 ist eine zweite Hashfunktion. Dies entspricht der Verwendung einer Sondierfunktion $s(j,k) = j \cdot h_2(k)$

Anforderung:

- Alle Testsequenzen sollen jede Adresse im Feld generieren können, d.h. $h_2(k)$ muß so gewählt werden, daß für alle Schlüssel k die Sondierungsfolge eine Permutation der Hashadressen bildet. Das bedeutet, daß $h_2(k) \neq 0$ sein muß und m nicht teilen darf.
- h_2 muß verschieden von der ersten Hashfunktion h sein, sonst Clusterbildung.
- einfach zu berechnen

üblich: $h_2 = k \mod S$

mit $S<m$

S und m haben **keinen gemeinsamen Teiler**

(dies ist trivial, wenn m als Primzahl gewählt wurde)

Double Hashing, Beispiel

m = 19

$h_1(k) = k \bmod 19$

$h_2(k) = 8-(k \bmod 8)$

Schlüssel:	**A**	**S**	**E**	**A**	**R**	**C**	**H**	**I**	**N**	**G**	**E**	**X**	**A**	**M**	**P**	**L**	**E**
Hash 1:	*1*	*0*	*5*	*1*	*18*	*3*	*8*	*9*	*14*	*7*	*5*	*5*	*1*	*13*	*16*	*12*	*5*
Hash 2:	*7*	*3*	*3*	*7*	*6*	*5*	*8*	*7*	*2*	*1*	*3*	*8*	*7*	*3*	*8*	*4*	*3*

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*
	A																	
S																		
					E													
	A							**A**										
																		R
			C															
								A								**H**		
									I									
														N				
							G											
					E			**A**			**E**							
					E								**X**		**A**			
	A							**A**										
S			**C**			**M**							**X**			**H**		
		P			**E**								**X**			**H**		
												L						
					E			**A**			**E**			**N**			**E**	

Komplexität von Hash-Verfahren

Hashverfahren erlauben eine sehr schnellen Direktzugriff auf Datensätze. Ist m>N und h geschickt gewählt, so sind Überläufer selten, und der Aufwand für das Suchen, Einfügen und Löschen ist

$$\approx \mathbf{O(1)}$$

Kommentare

Die verschiedenen Verfahren zur Behandlung von Kollisionen haben Vor- und Nachteile; sie sind in Abhängigkeit von der Anwendung zu wählen.

Hashverfahren sind auch zum schnellen Direktzugriff auf Sekundärspeicher geeignet. Dann hält man die Hashtabelle im Hauptspeicher, jeder Eintrag enthält nicht den Datensatz selbst, sondern einen Zeiger auf den Datensatz auf dem externen Speicher.

Eine effiziente sequentielle Verarbeitung der Datensätze ist nicht möglich!

4.5.2 Dynamisches Hashing

Alle statischen Hash-Verfahren haben einen entscheidenden Nachteil:
wenn N a priori unbekannt ist oder stark variiert, sind sie sehr ineffizient.

Denn:
Bei offenen Hashverfahren (Sondierung) ist das Einfügen von Schlüsseln über den vorgesehenen Speicherplatz hinaus unmöglich; bei Hashverfahren mit Verkettung der Überläufer ist dies prinzipiell zwar möglich, beeinträchtigt aber die Effizienz der Verfahren erheblich. Im Extremfall degeneriert nach unvorhergesehen vielen Einfügeoperationen ein Hashverfahren mit Verkettung der Überläufer zur Verwaltung relativ weniger, sehr langer verketteter linearer Listen. Im anderen Extremfall wird eine sehr große Hashtabelle für nur wenige Einträge freigehalten.

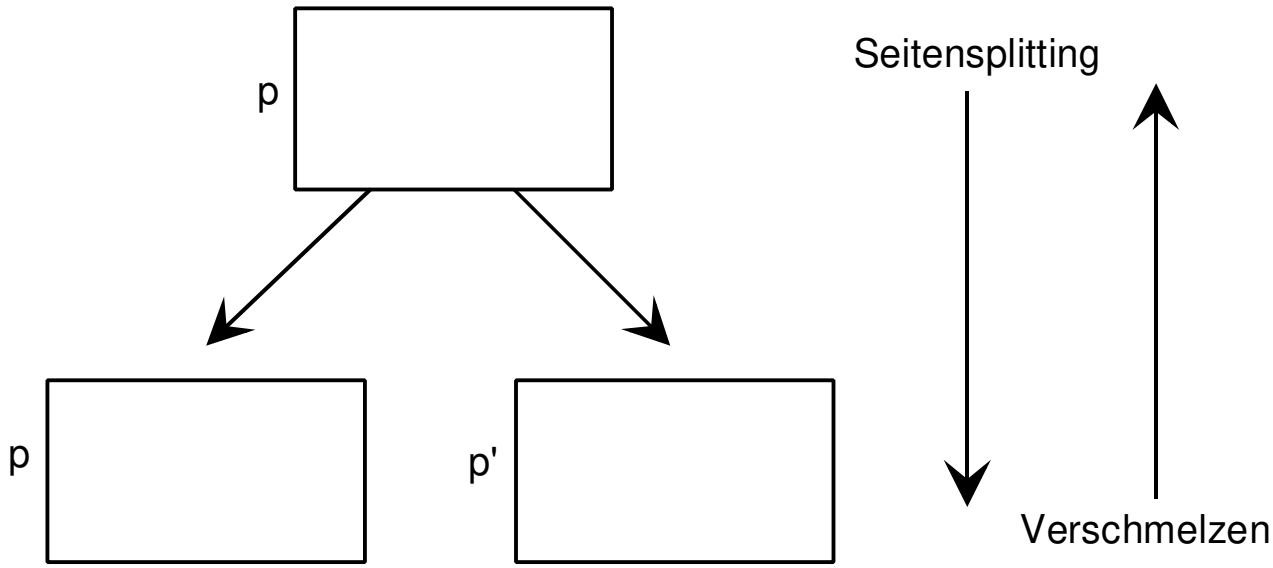
Abhilfe: *dynamische Hash-Verfahren*

Annahme:

Die Datensätze werden in Blöcken (buckets) fester Größe gespeichert, jeweils *b* Datensätze passen in einen Block.

Splitting

Alle dynamische Hash-Verfahren beruhen auf einem Splitting von Blöcken, um nachträglich mehr Platz zu schaffen. Schrumpft der Datenbestand, so werden Blöcke verschmolzen.



Eine Seite p wird wie folgt gesplittet: die auf der Seite p gespeicherten Datensätze werden irgendeiner Regel folgend in zwei gleich große Gruppen eingeteilt und eine neue leere Seite p' wird irgendwie bereitgestellt. Die Datensätze in einer der Gruppen verbleiben in der Seite p (der alten Seite), während die Datensätze der anderen Gruppe in die Seite p' (die neue Seite) umgespeichert werden. Die Seiten p und p' heißen Brüder. Wenn die Anzahl der Datensätze, die in zwei Brüderseiten gespeichert sind, kleiner wird als b, können die beiden Seiten verschmolzen werden, wobei sie dann eine Seite frei machen.

Die verschiedenen Verfahren des dynamischen Hashings unterscheiden sich dahin, **wie** ein Block gesplittet wird, **welcher** Block gesplittet wird und **wann** ein Splitting ausgelöst wird.

Virtuelles Hashing

Erstmals angegeben von Litwin (1978)

Idee

- Bei Überschreiten eines bestimmten Belegungsfaktors wird der Speicherbereich verdoppelt.

- Eine neue Hash-Funktion $_{i+1}$ wird bestimmt, die die Schlüssel auf den gesamten Bereich abbildet.

- Ein Bitvektor mit einem Bit pro altem Block zeigt an, ob für eine alte Seite noch die alte Hash-Funktion h_i oder für diese Seite und ihren Bruder schon die neue Hash-Funktion h_{i+1} gilt.

- Wird eine der neuen Seiten in Betrieb genommen, so werden die in der alten Seite vorhandenen Schlüssel gemäß der neuen Hash-Funktion h_{i+1} auf die alte Seite und ihren Bruder verteilt.

Virtuelles Hashing, Beispiel

m_0 = 5, 5 Blöcke (buckets)

$h_0(k)$ = k mod 5

$h_1(k)$ = k mod 10

usw.

Auslöser für Verdopplung: ein Block läuft über Auslöser für Splitting eines Blocks: der alte Bruder läuft über.

147

a)

0	1	2	3	4
535		802		
215	186	482		484
385	396	257	613	549
180	521	152	268	404
h_0	h_0	h_0	h_0	h_0

L=0

b)

0	1	2	3	4	5	6	7	8	9
535	186								
215	396	802		484					
385	521	482	613	549			147		
180		152	268	404			257		
h_0	h_0	h_0	h_1	h_0			h_1		
0	0	1	0	0					

L=1

Virtuelles Hashing, Suchen und Löschen

Suche

Um auf einen Satz zuzugreifen, berechnen wir zunächst $h_0(k)$, prüfen das Bit $B_0[h_0(k)]$, und wenn dieses Bit eine 1 ist, wissen wir, daß die Seite bereits gesplittet worden ist, folglich kann die Adresse durch Berechnung von $h_1(k)$ gefunden werden.

Löschen

Nach Löschen eines Satzes prüfen wir, ob die Seite mit ihrer Bruderseite verschmolzen werden kann. Die Datei läßt sich immer kontrahieren, wenn alle Bits in der Tabelle B_{L-1} gleich 0 sind.

Virtuelles Hashing, Komplexität und Kommentare

Komplexität

Annahme: Der Bitvektor kann im Hauptspeicher gehalten werden. Der Aufwand zum Suchen eines Schlüssels innerhalb von einem Block ist vernachlässigbar.

- Suche: O(1)
- Einfügen: O(1)
- Löschen: O(1)

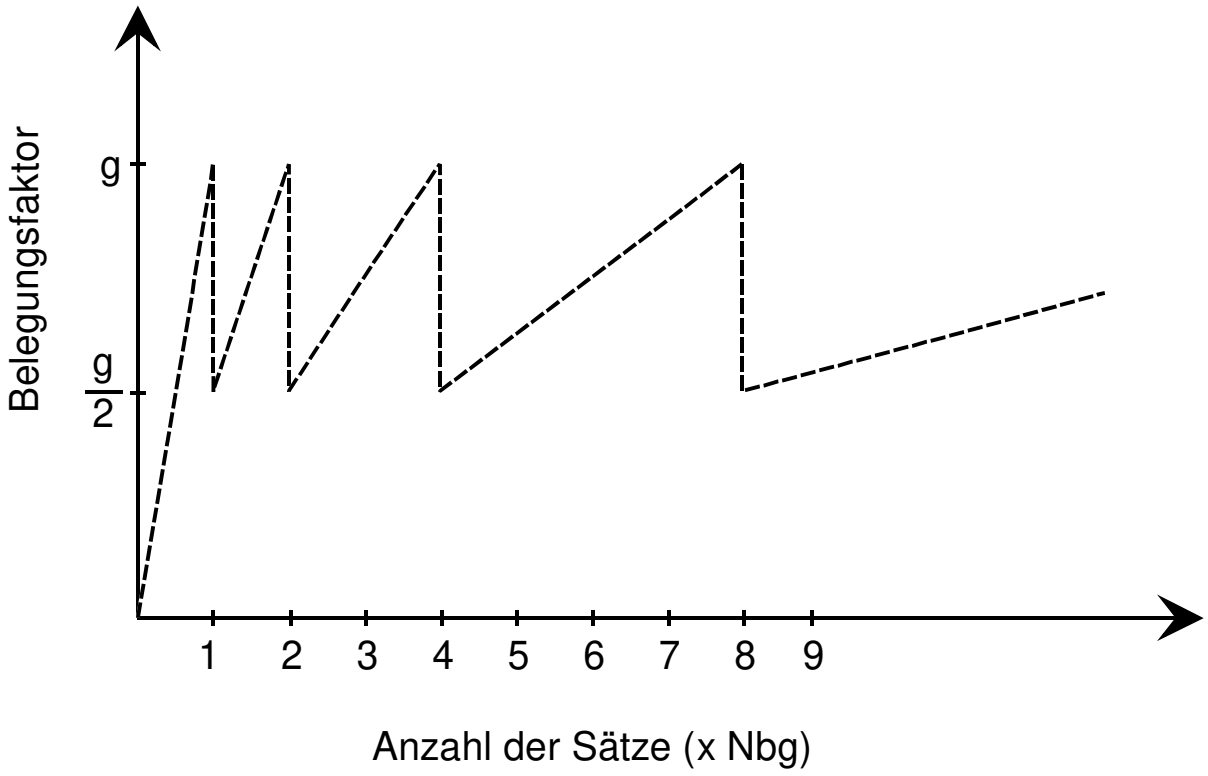
Kommentare

- Es gibt keine Überläufer
- Im Normalbetrieb sehr effizient
- Aber: Speicherplatzausnutzung sehr schlecht wegen der Verdoppelung!

Variante des VH mit besserer Speicherplatzausnutzung

- Verdoppelung des Speicherbereichs erst, wenn ein festgelegter Belegungsfaktor g überschritten wird (z.B. g=0,9). Ansonsten Überlaufblöcke mit getrennter Verkettung.

Entwicklung der Belegungsfaktors bei wachsender Datei:



Komplexität

Mittlere Speicherplatzausnutzung: ln 2 ≈ 0,693

(Beweis kompliziert)

Anzahl der Zugriffe bei der Suche

Seiten-größe	Erfolgreiche Suche			Erfolglose Suche		
	min	mittel	max	min	mittel	max
5	1.11	1.17	1.29	1.27	1.42	1.70
10	1.03	1.09	1.22	1.13	1.36	1.81
20	1.00	1.04	1.15	1.02	1.27	1.88
50	1.00	1.01	1.08	1.00	1.19	1.89

g=0,9

Erweiterbares Hashing (Extendible Hashing)

erstmals von Fagin et. al. (1979)

Idee

- Indirekte Adressierung über eine kleine Tabelle

- Das Ergebnis wird eine Tabelle (Verzeichnis) für indirekten Zugriff eingerichtet; sie hat 2^d, $d \geq 0$ Einträge. d heißt *Tiefe* der Tabelle

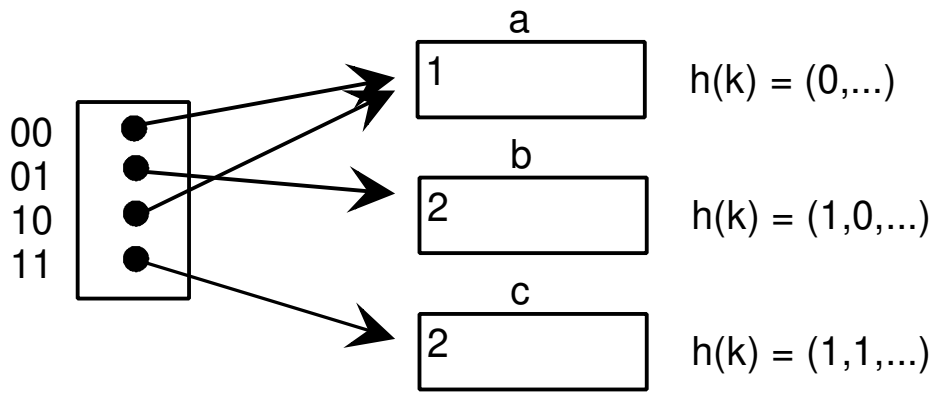
- Wenn die Tabelle die Tiefe d hat, benötigen wir d Bits des Schlüssels, um den Speicherblock des Satzes zu bestimmen. Der Eintrag D(q) in der Tabelle mit $q=\sum_{j=0}^{d-1} b_j 2^j$ liefert uns die gesuchte Seitenadresse.

- Mehrere Tabelleneinträge können durchaus auf dieselbe Seite zeigen. Beim Wachsen wird *gegebenenfalls die Tabelle verdoppelt, nicht der Speicherbereich selbst.*

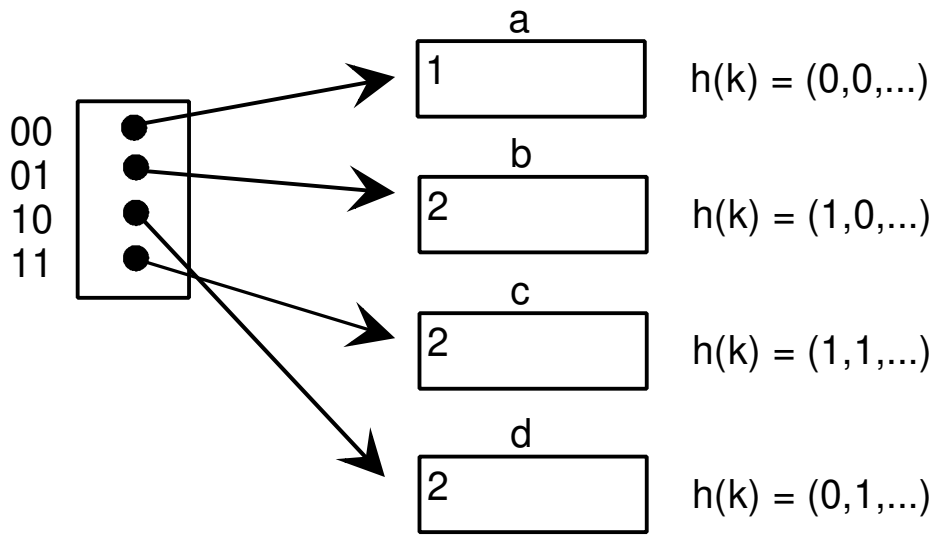
Erweiterbares Hashing, Beispiel

Tief der Tabelle d=2

Alle Bitmuster, die mit 0 enden, passen in Block a:

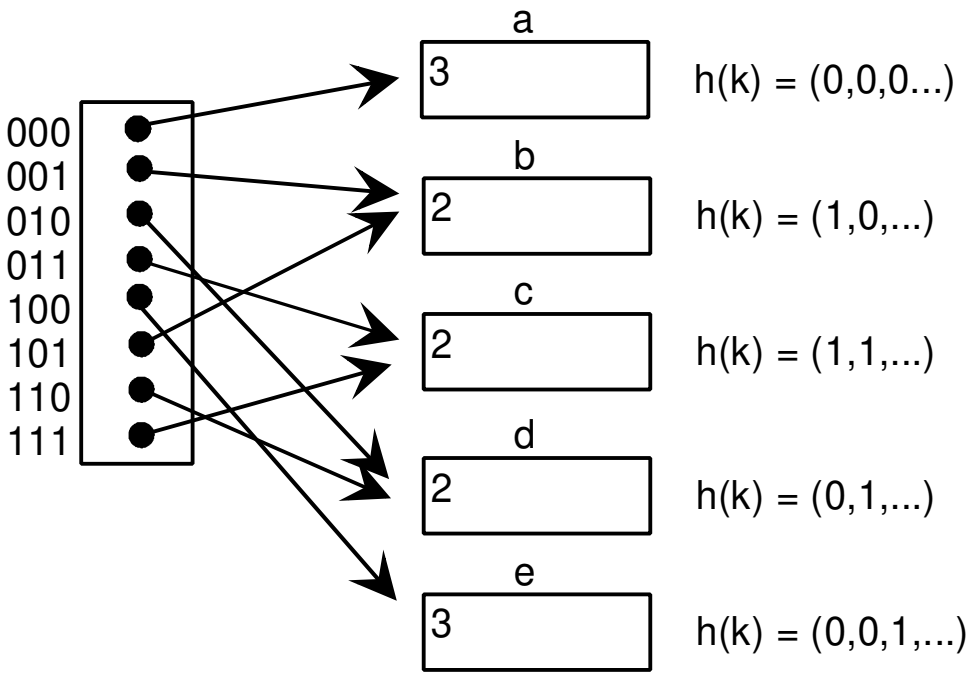


Nun wächst die Datei; Block a läuft über und wird gesplittet:



Verdopplung der Tabelle

Immer dann, wenn ein Block gesplittet werden muß, auf den nur ein Tabelleneintrag zeigt, muß die Tabelle verdoppelt werden. Die zweite Hälfte wird als Kopie der ersten Hälfte initialisiert. Danach kann die volle Seite gesplittet werden. Der Block a sei derjenige, der überläuft.



Erweiterbares Hashing, Komplexität und Kommentare

Komplexität

Annahme: Die Tabelle kann im Hauptspeicher gehalten werden.

- Suche: O(1)
- Einfügen: O(1)
- Löschen: O(1)

Speicherplatzbelegung: $\ln 2 \approx 0{,}693$

Kommentare

- bessere Speicherplatzausnutzung als VH in der Grundversion
- bessere Zugriffszeiten als VH mit verspäteter Verdopplung und verketteten Überläufen
- aber: größere Tabellen im Hauptspeicher als VH. Muß eventuelle selbst auf den Sekundärspeicher ausgelagert werden.

Lineares Hashing

Erstmals angegeben von Litwin (1979)
Die bisher behandelten Methoden der Dynamischen Hashing benutzen Tabellen im Hauptspeicher und splitten jeweils die überlaufende Seite. Sie können ohne verkettete Überlaufblöcke auskommen.

Lineares Hashing kommt ohne Tabelle aus, verkettet aber Überläufer.

Idee

- Es wird eine Folge von Hash-Funktionen h_0, h_1, h_2, ... definiert wie bei VH.

- Wenn der Belegungsfaktor überschritten ist, wird irgendein Block gesplittet (nicht unbedingt ein überlaufender Block). Der überlaufende Satz wird als verketteter Überläufer gespeichert. Der zu splittende Block wird durch einen Zeiger p angezeigt, der inkrementiert wird; d.h., alle Blöcke < p sind bereits gesplittet, alle ≥ p noch nicht.

- Durch das Splitting vergrößert sich bei jedem Überlauf der Speicherbereich, und die Wahrscheinlichkeit weiterer Überläufer sinkt.

- Wird eine Block gesplittet, der Überläufer hat, so werden alle Datensätze auf die Originalseite und den Bruder gleichmäßig verteilt.

Lineares Hashing, Beispiel

m_0 = 5, 4 Sätze passen in eine Seite

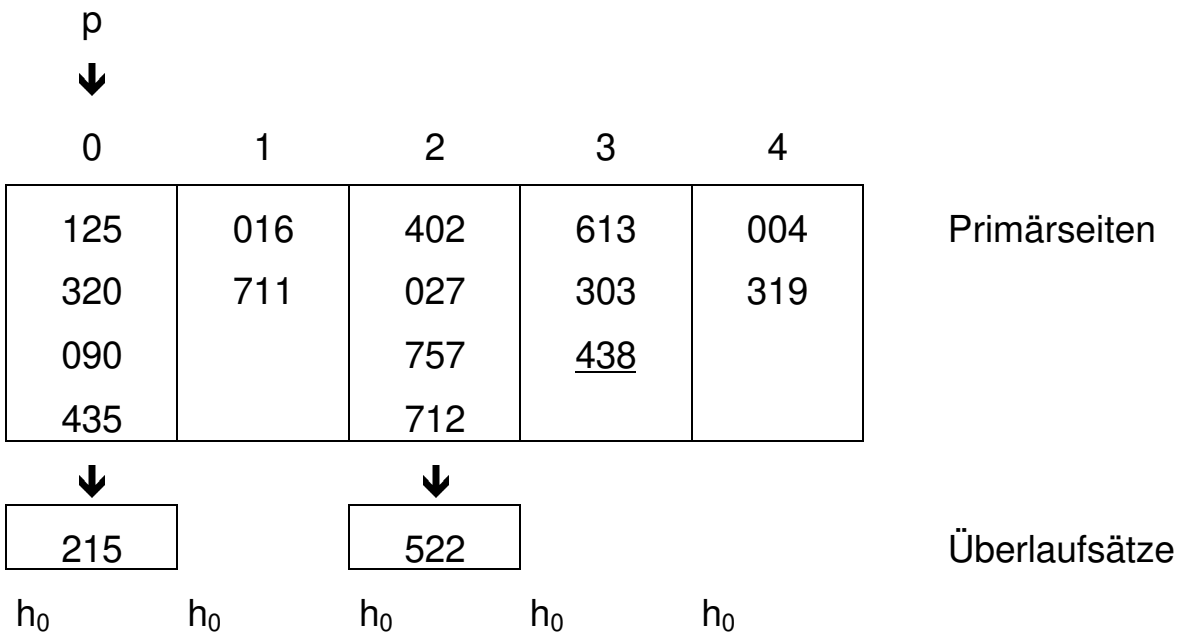
$h_0(k)$ = k mod 5

$h_1(k)$ = k mod 10

usw.

Expansion des Speicherbereichs bei Belegungsfaktor > 0,8

Ausgangspunkt



Einfügen von Satz 438 hat den Belegungsfaktor auf 0,85 angehoben. Block 5 wird neu angelegt, die Sätze aus Block 0 verteilt. Zeiger p gibt an, daß für alle Blöcke i < p und ihre Brüder i+5 die neue Hash-Funktion h_1 gilt, für alle Blöcke $p \le j \le 4$ die alte Hash-Funktion h_0. Ist p=5, so sind bereits alle Blöcke auf h_1 umgestellt, und das Spiel beginnt von vorne mit h_1 und h_2 und p=0.

Lineares Hashing, Suchen

Suche

Wenn wird den Wert von p kennen, können wir die zu einem Satz gehörende Seite immer noch finden, ohne auf irgend eine andere Seite zugreifen zu müssen. Der gesuchte Satz ist entweder in dieser Seite selbst oder in der Überlaufkette, die von dieser Seite ausgeht, zu finden. Um die Adresse der jeweiligen Seite zu bestimmen, berechnen wir zunächst $j_0(k)$. Wenn $h_0(k) \geq p$ ist, wurde diese Seite noch nicht gesplittet, und $h_0(k)$ ist die gewünschte Adresse. Wenn aber $h_0(k) < p$ ist, war die Seite bereits gesplittet worden, und folglich wird die gewünschte Adresse durch $h_1(k)$ geliefert. In unserem Beispiel hat das Splitten von Seite 0 das Verhalten der Datei verbessert, weil ein Überlaufsatz verschwunden ist.

Varianten

Es gibt viele Möglichkeiten, den Auslöser für das Splitting zu bestimmen, z.B. auch das Auftreten eines Überläufers (wobei allerdings in der Regel eine andere Seite gesplittet wird). Sie unterscheiden sich geringfügig in der Speicherplatzausnutzung und im Retrival-Verhalten.

Lineares Hashing, Komplexität und Kommentare

Komplexität

Da es Überläufer gibt, ist die Anzahl der Zugriffe in der Regel > 1, aber nicht abhängig von N.

- Suche: O(1)
- Einfügen: O(1)
- Löschen: O(1)

Speicherplatzausnutzung: typischerweise 0,6 bis 0,8, je nach Splitting-Auslöser-Funktion

Kommentare

- keine Tabelle, nur Zeiger p, daher Bedarf an zusätzlichem Speicher minimal
- erfordert Verkettung von Überläufern, dadurch etwas komplexere Programmierung

Hash-Verfahren, Zusammenfassung

- **Hervorragend geeignet für den Direktzugriff auf sehr große Dateien**
- **Ungeeignet für sequentiellen Zugriff**
- Alle Operationen O(1)
- Man kann *entweder* die Speicherplatzausnutzung *oder* die mittlere Anzahl von Zugriffen auf Blöcke optimieren
- Bei Dateien mit stark variierender Größe sind dynamische Hash-Verfahren vorzuziehen