Kapitel 9

Hashverfahren

9.1 Einführung

Uns sind bereits Verfahren bekannt, mit denen Datensätze mit einem eindeutigen Schlüssel gespeichert werden (z.B. B*-Bäume). Statt bei der Suche nach einem Schlüssel k mehrere Schlüsselvergleiche mit der Menge $\mathcal{K}$ aller Schlüssel durchzuführen (um in dem Baum zu navigieren), wird bei Hashverfahren versucht, durch eine *Berechnung* festzustellen, wo der Datensatz mit Schlüssel k gespeichert ist.[1]

Idee des Verfahrens:
Die Datensätze werden in einem linearen Feld mit Indizes $0, .., m-1$ gespeichert. Jede Position der Tabelle nimmt maximal einen Eintrag auf. Dieses Feld nennt man *Hashtabelle*, m ist die *Größe* der Hashtabelle.
Eine Hashfunktion $h : \mathcal{K} \to 0, ..., m-1$ ordnet jedem Schlüssel k einen Index $h(k)$ mit $0 \leq h(k) \leq m-1$ zu, die *Hashadresse*. $\mathcal{K}$ ist die Menge aller Schlüssel und $K = \{k\}$ ist die Menge der auftretenden Schlüssel. Im allgemeinen ist K eine sehr kleine Teilmenge von $\mathcal{K}$ und $m \ll card\ K$. Die Hashfunktion kann also im allgemeinen nicht injektiv sein, sondern muss verschiedene Schlüssel auf dieselbe Hashadresse abbilden.

Definition:
Zwei Schlüssel k,k', $k \neq k'$ mit $h(k) = h(k')$ heißen *Synonyme*.

Befinden sich beide Schlüssel in der aktuellen Schlüsselmenge K, so ergibt sich eine *Adresskollision*, die dazu führt, dass der neu hinzugekommene Eintrag in einem anderen freien Feld gespeichert wird. In diesem Falle muss eine Sonderbehandlung vorgenommen werden, die zusätzlichen Aufwand erfordert. Treten dagegen in K keine Synonyme auf, so kann jeder Datensatz in

[1] Wir unterstellen, dass jeder Schlüssel k einzig (unique) ist, d.h. nicht mehrfach auftritt.

der Hashtabelle an der seiner Hashadresse entsprechenden Stelle gespeichert werden.
Ein Hashverfahren soll deshalb zwei Forderungen genügen:

1. Es sollen so wenig Kollisionen wie möglich auftreten. (durch geeignete Hashfunktion)

2. Adresskollisionen sollen möglichst effizient aufgelöst werden.

Da auch die beste Hashfunktion Kollisionen nicht vermeiden kann, sind Hashverfahren im schlimmsten Fall sehr ineffiziente Realisierungen der Operationen *Suchen*, *Einfügen* und *Entfernen*. Im Durchschnitt sind sie aber weitaus effizienter als Verfahren, die auf Schlüsselvergleichen basieren. So ist die Zeit zum Suchen eines Schlüssels unabhängig von der Anzahl der gespeicherten Datensätze, vorausgesetzt, dass genügend Speicher zur Verfügung steht.

Definition:
Für eine Hashtabelle der Größe m, die gerade n Schlüssel speichert, nennen wir den Quotienten aus n und m, $\alpha = \frac{n}{m}$, den *Belegungsfaktor* der Tabelle.

Die Anzahl der zum Suchen, Einfügen oder Entfernen eines Schlüssels benötigten Schritte hängt im wesentlichen vom Belegungsfaktor α ab.
Hashverfahren sind besonders effektiv, wenn in den Hastabellen anfänglich viele Einfügeoperationen vorkommen und dann fast nur noch gesucht und fast nichts entfernt wird (trifft für viele Geoinformationssysteme zu, da z.B. geografische Fakten vielfach sehr langlebig sind und normalerweise selten wieder entfernt werden).

Inhalt der folgenden Kapitel:
In Kapitel 9.2 werden wir uns mit der Wahl geeigneter Hashfunktionen beschäftigen. Kapitel 9.3 widmet sich den offenen und geschlossenen Hashverfahren, während in Kapitel 9.4 die dynamischen Hashverfahren vorgestellt werden.

9.2 Geeignete Hashfunktionen

Eine gute Hashfunktion muss möglichst leicht und schnell berechenbar sein und die zu speichernden Datensätze möglichst gleichmäßig auf den Speicherbereich verteilen (selbst wenn die Datensätze in Clustern vorliegen), um

Adresskonflikte zu vermeiden.
D.h. die von der Hashfunktion zu gegebenen Schlüsseln berechneten Hashadressen sollten über dem Adressraum möglichst gleichverteilt sein, auch dann, wenn die Schlüssel aus $\mathcal{K}$ nicht gleichverteilt sind.
Wir werden im folgenden von nichtnegativen ganzzahligen Schlüsseln k ausgehen, also $\mathcal{K} \subseteq N_0$ annehmen.
Die Anforderungen der Geoinformationssysteme an die Hashfunktionen sind anders als in der Kryptographie. Die hier vorgestellten Funktionen sind für die Kryptographie im allgemeinen nicht nutzbar.

9.2.1 Die Divisions-Rest-Methode

Ein Verfahren zur Erzeugung von Hashadressen $h(k), 0 \leq h(k) \leq m-1$ zu gegebenen Schlüsseln $k \epsilon N_0$ nutzt, den Rest von k bei ganzzahliger Division durch m:

$$h(k) = k \ mod \ m$$

m sollte kein Teiler von Zahlen der Form $r^i \pm j$ sein (i, j kleine nichtnegative ganze Zahlen ; r Basis des Zahlensystems der Schlüssel).
Dabei sollte m eine Primzahl sein, die keine solche Zahlen $r^i \pm j$ teilt. Diese Wahl hat sich in praktischen Fällen ausgezeichnet bewährt.

9.2.2 Die multiplikative Methode

Der gegebene Schlüssel k wird mit einer irrationalen Zahl ϱ multipliziert. Der ganzzahlige Anteil des Ergebnisses wird abgeschnitten ($= k\varrho - \lfloor k\varrho \rfloor$), womit man für verschiedene Schlüssel immer Werte zwischen 0 und 1 bekommt. Diese Werte sind gleichmäßig verteilt im Intervall $[0, 1)$.
Von allen möglichen irrationalen Zahlen x, $0 \leq x \leq 1$ führt der *goldene Schnitt*

$$\varrho^{-1} = \frac{\sqrt{5}-1}{2} \approx 0.6180339887$$

zu einer gleichmäßigen Verteilung. Damit ergibt sich für die Hashfunktion:

$$h(k) = \lfloor m(k\varrho^{-1} - [k\varrho^{-1}]) \rfloor$$

Neben diesen beiden Methoden gibt es noch viele andere Verfahren wie z.B. nach einer Transformation des Schlüssels (in ein anderes Zahlensystem oder durch Quadrieren) einzelne Zifferpositionen auszuwählen. Man hat herausgefunden, dass das Divisions-Rest-Verfahren im Durchschnitt die besten Resultate liefert. In Kapitel 9.3. verwenden wir zur Beschreibung der Kollisi-

onsauflösungsproblematik dieses Verfahren.

9.2.3 Perfektes und universelles Hashing [2]

Wenn die Anzahl der zu speichernden Schlüssel kleiner ist als die Anzahl der verfügbaren Speicherplätze, so gilt für die Teilmenge K der Menge $\mathcal{K}$ aller möglichen Schlüssel: $|K| \leq m$. [3] Somit ist eine kollisionsfreie Speicherung von K immer möglich. Wenn K bekannt und fest ist, kann man eine injektive Abbildung $h : K \rightarrow \{0, .., m-1\}$ wie folgt berechnen: Man ordnet die Schlüssel lexikographisch und bildet jeden Schlüssel auf seine Ordnungsnummer ab. Damit haben wir eine *perfekte* Hashfunktion, die Kollisionen vollständig vermeidet.
Dieser Fall (K bekannt und fest) ist eher die Ausnahme. Im allgemeinen kennen wir $K \subseteq \mathcal{K}$ nicht und können selbst dann, wenn $|K| \leq m$, nicht davon ausgehen, dass die Hashfunktion kollisionsfrei ist.
Wenn z.B. ein Programmierer eine gewisse Vorliebe bei der Wahl seiner Variablennamen hat, kann es durch eine ungünstige Hashfunktion zu vielen Kollisionen kommen, wenn diese Namen einem Hashverfahren unterworfen werden.
Die einzige Möglichkeit diese unerwünschte Situation zu vermeiden, ist das Einführen einer Menge von Hashfunktionen $\mathcal{H}$, aus der dann zufällig eine Hashfunktion h ausgewählt wird. Die Auswahl von h ist Teil des Verfahrens und unterliegt keiner möglicherweise einseitigen Vorliebe des Benutzers. Diese Art der Randomisierung garantiert damit, dass eine schlecht gewählte Schlüsselmenge K nicht jedes mal zu vielen Kollisionen führt. Zwar kann eine einzelne Funktion $h \in \mathcal{H}$ noch immer viele Schlüssel auf die gleiche Hashadresse abbilden, aber gemittelt über alle Funktionen aus $\mathcal{H}$ ist das nicht mehr möglich.
Sei also $\mathcal{H}$ eine endliche Kollektion von Hashfunktionen, so dass jede Funktion aus $\mathcal{H}$ jeden Schlüssel im Universum $\mathcal{K}$ aller möglichen Schlüssel auf eine Hashadresse aus $\{0, .., m-1\}$ abbildet. $\mathcal{H}$ heißt *universell*, wenn für je zwei verschiedene Schlüssel $x, y \in \mathcal{K}$ gilt:

$$\frac{|\{h \in \mathcal{H} : h(x) = h(y)\}|}{|\mathcal{H}|} \leq \frac{1}{m}$$

Mit anderen Worten, $\mathcal{H}$ ist *universell*, wenn für jedes Paar von zwei verschiedenen Schlüsseln höchstens der m-te Teil aller Funktionen der Klasse zu einer Adresskollision für die Schlüssel des Paares führen.

[2] Ergänzend zur Vorlesung
[3] $|K|$ ist Kardinalität der Menge K

9.2.4 Hashverfahren mit externer Verkettung der Überläufer

Soll eine Hashtabelle t, die bereits einen Schlüssel k enthält, ein Synonym k' von k aufnehmen, so ergibt sich ein Adresskonflikt, da der Platz $h(k) = h(k')$ bereits belegt ist. Eine Art die Überlaufer zu speichern ist es, sie außerhalb der Hashtabelle in dynamisch veränderbaren Strukturen abzuspeichern. So kann man alle Überläufer zu jeder Hashadresse in einer linearen, einfach verketteten Liste ablegen, die dann dynamisch erweitert wird, sobald an dieser Hashadresse ein Adresskonflikt auftritt.

Methode: *Separate Verkettung der Überläufer*
Jedes Element der Hashtabelle $t = t[i]$, $i = 0, .., m-1$ ist Anfangselement einer Überlaufkette (verkettete lineare Liste).
•*Suchen* nach Schlüssel k: Beginne die Suche bei $t[h(k)]$. Wurde der Schlüssel gefunden, ist die Suche beendet. Wenn k nicht gefunden wurde, gehe zum nächsten Überläufer und prüfe diesen. Auf diese Weise wird die gesamte Überlaufkette geprüft. Sobald k in der Kette gefunden wurde, ist die Suche beendet. Ist das Ende der Überlaufkette erreicht, noch bevor k gefunden wurde, endet die Suche erfolglos.
•*Einfügen* eines Schlüssels k: Suche nach k. Die Suche verläuft erfolglos (anderenfalls wird k nicht eingefügt) und endet am Ende einer Überlaufskette oder bei $t[h(k)]$. In diesem Fall trage k in $t[h(k)]$ ein, ansonsten erzeuge ein neues Listenelement und hänge es ans Ende der Überlaufkette, die bei $t[h(k)]$ beginnt.
•*Entfernen* eines Schlüssels k: Suche nach k. Die Suche verläuft erfolgreich (sonst kann k nicht entfernt werden). Steht k in der Hashtabelle, so streiche k dort. Falls eine Überlaufkette bei $t[h(k)]$ beginnt, so übertrage das erste Element der Überlaufkette nach $t[h(k)]$ und entferne es aus der Überlaufkette. Ist k ein Element der Überlaufkette, so entferne diese Element aus der Überlaufkette.

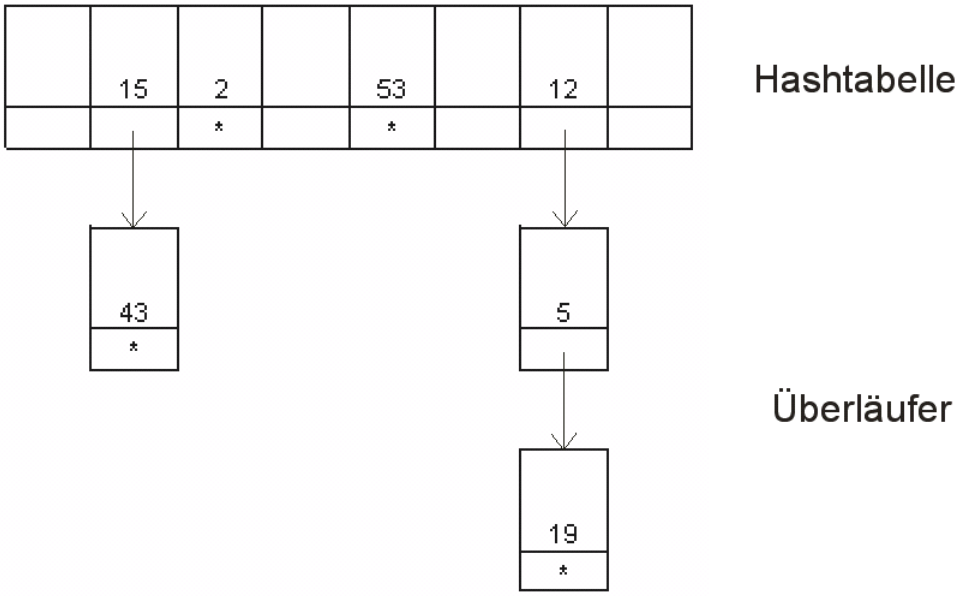


Abbildung 9.1: Separate Verkettung der Überläufer

9.2.5 Hashverfahren mit direkter Verkettung der Überläufer

Jedes Element der Hashtabelle ist ein Zeiger auf eine Kette, d.h. der erste an der Position $t[h(k)]$ einzutragende Schlüssel erzeugt bereits das erste Element der Kette.
•*Suchen* nach Schlüssel k: Beginne bei $t[h(k)]$ und folge den Verweisen der Kette, bis entweder k gefunden wurde (erfolgreiche Suche) oder das Ende der Kette erreicht wurde (erfolglose Suche).
•*Einfügen* eines Schlüssels k: Suche nach k. Die Suche endet erfolglos am Ende einer Kette (sonst wird k nicht eingefügt). Erstelle ein neues Listenelement und hänge es am Ende der Kette an.
•*Entfernen* eines Schlüssels k: Suche nach k. Die Suche verläuft erfolgreich (sonst kann k nicht entfernt werden) und endet bei einem Element der Kette. Entferne dieses Element aus der Kette.

Im wesentlichen handelt es sich bei dieser Methode um Operationen in einfach verketteten Listen.

Analyse: Betrachten wir als erstes die Methode der direkten Verkettung der Überläufer. Wir nehmen an, dass die Hashfunktion alle Hashadressen mit gleicher Wahrscheinlichkeit und von Operation zu Operation unabhängig liefert. D.h. die Wahrscheinlichkeit, dass bei der j-ten Operation die Adresse j' ausgewählt wird $(0 \leq j' \leq m-1)$,ist unabhängig von j stets gleich $1/m$, für alle j'.

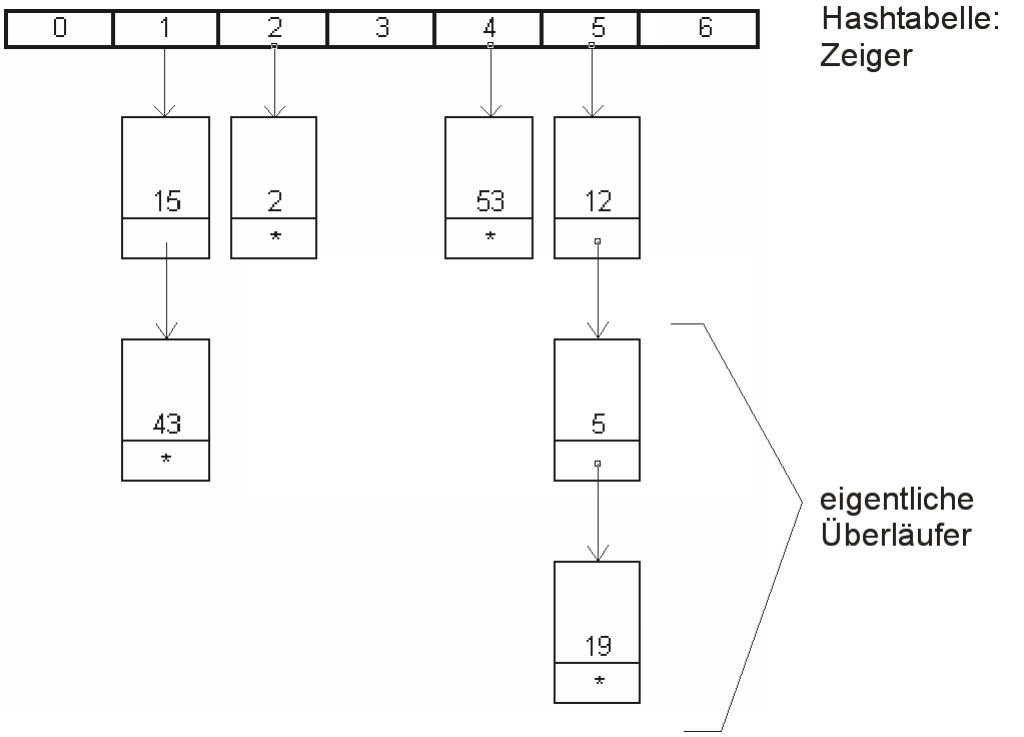


Abbildung 9.2: Direkte Verkettung der Überläufer

Im Mittel ist die Anzahl der bei der erfolgreichen Suche nach einem Schlüssel betrachteten Einträge

$$C_n = \frac{1}{n}\sum_{j=1}^{n}(1+(j-1)/m) = 1 + \frac{n-1}{2m} \approx 1 + \frac{\alpha}{2} \quad ,$$

wenn nach jedem Schlüssel mit gleicher Wahrscheinlichkeit gesucht wird.
Bei einer erfolglosen Suche nach k betrachten wir alle Einträge der bei $h[(k)]$ beginnenden Überlaufkette. Die durchschnittliche Anzahl der Einträge in einer Kette ist gerade n/m, wenn n Einträge auf m Ketten verteilt sind. Da dies auch der Belegungsfaktor α ist, erhalten wir:

$$C'_n = \alpha$$

Nach den angegebenen Methoden der separaten und direkten Verkettung der Überläufer ist klar, dass die Effizienz der erfolgreichen Suche auch gleichzeitig die Effizienz der Entferne-Operation ist, und dass das Einfügen gerade so effizient ist wie die erfolglose Suche.
Die Effizienz der erfolglosen Suche lässt sich verbessern, indem man die Überlaufkette sortiert hält. Dann braucht man bei der erfolglosen Suche nicht mehr bis zum Listenende suchen, sondern kann im Mittel bereits in der Mitte der Überlaufliste mit der Suche aufhören. Der Preis dafür sind teurere

Einfügeoperationen.
Die separate und direkte Verkettung der Überläufer weist einige wesentliche Vorteile gegenüber anderen Hashverfahren auf. Ein Belegungsfaktor von mehr als 1 ist möglich, d.h. selbst wenn die Datenmenge mehr als vorgesehen wächst, funktionieren diese Verfahren noch korrekt. Einträge können richtig gelöscht werden. Damit spart man Speicher und Laufzeit für Bereiche, die als gelöscht markiert sind. Die direkte Verkettung eignet sich besonders für den Einsatz von Externspeichern. So könnte man etwa die Hashtabelle im Internspeicher halten und auf die Datenseiten verketten. Der Zugriff erfolgt als seitenweises Lesen vom Sekundärspeicher.
Die folgende Tabelle vermittelt einen Eindruck von der Effizienz der Verkettung der Überläufer.

Anzahl bei der Suche betrachteter Einträge [4]

Belegungsfaktor	Separate Verkettung		direkte Verkettung	
	erfolgreich	erfolglos	erfolgreich	erfolglos
α=0.50	1.250	1.110	1.250	0.50
α=0.90	1.450	1.307	1.450	0.90
α=0.95	1.475	1.337	1.475	0.95
α=1.00	1.500	1.368	1.500	1.00

9.3 Offene Hashverfahren

Im Unterschied zur Verkettung der Überläufer außerhalb der Hashtabelle versucht man bei offenen Hashverfahren, Überläufer *in* der Hashtabelle unterzubringen. Wenn es also bei einer Einfügeoperation zu einem Adresskonflikt kommt, muss man nach einer festen Regel einen freien Platz in der Hashtabelle finden. Als *Sondierung* bezeichnet man die Folge, in der die Speicherplätze betrachtet werden, die für ein Einfügen in Frage kommen. Wenn dann ein freier Platz gefunden wurde, wird der Schlüssel an dieser Stelle eingefügt.
Das *Entfernen* von Schlüsseln aus der Hashtabelle ist bei allen offenen Hashverfahren problematisch. Ein bereits in der Hashtabelle vorhandener Schlüssel k versperrt einem neu einzufügenden Schlüssel k' einen Platz, den k' gemäß seiner Sondierungsfolge belegen würde. Damit muss k' auf einen anderen Platz ausweichen. Löscht man nun k aus der Hashtabelle, so kann man k' nicht mehr in der Hashtabelle finden, da die Position $h(k')$ leer ist. Darum wird in den meisten Verfahren der Platz lediglich als gelöscht markiert. Bei erneutem Einfügen kann dieser Platz genutzt werden und die Suche ist ohne Umsortierung der Hashtabelle möglich. Siehe dazu das Beispiel beim linearen Sondieren.

[4] Ottmann & Widmayer - Algorithmen und Datenstrukturen, Seite 181 Tabelle 4.1

9.3.1 Die Grundoperationen bei Offenen Hashverfahren

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 frei.
•*Suchen* nach Schlüssel k:
1. Beginne die Suche mit Hashadresse $i = h(k)$.
2. Prüfe, ob k in $t[i]$ gespeichert ist - dann ist die Suche erfolgreich, oder ob $t[i]$ frei ist - dann ist die Suche erfolglos. Wenn nichts davon zutrifft mache weiter mit 3.
3. Nimm als neue Suchposition $i = (h(k) - s(j,k)\ mod\ m)$ und gehe zurück zu 2.
•*Einfügen* eines Schlüssels k: Wir nehmen an, dass k nicht schon in t vorkommt. 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. Bei erfolgreicher Suche markiere die Stelle i, an der k gefunden wurde, als gelöscht. Wenn k nicht gefunden wurde, kann k auch nicht entfernt werden.

9.3.2 Lineares Sondieren als Beispiel für offene Hashverfahren

Beim linearen Sondieren hat $s(j,k)$ die Form $s(j,k) = j$.
•*Einfügen* eines Schlüssels k: Beginne bei der Adresse $i = h(k)$. Solange k nicht in $t(i)$ gespeichert und $t(i)$ nicht frei ist, suche weiter bei $i = i - 1$. Falls $i < 0$, setze $i = m - 1$. Ist $t(i)$ frei, so trage k an dieser Stelle ein. Wenn alle t belegt sind, ist die Tabelle vollständig gefüllt, und es ist kein weiteres Einfügen möglich.
•*Suchen* eines Schlüssels k: Man berechne $h(k)$ und beginne die Suche an der Stelle $i = h(k)$. Solange der Inhalt von $t(i) \neq k$ und $t(i)$ nicht leer ist, setze $i = i - 1$. Bei $i < 0$ setze $i = m - 1$. Ist $t(i) = k$, so hat man den Eintrag gefunden. Ist $t(i)$ leer, so wurde k nicht gefunden. Man beachte, dass man bei erfolgloser Suche in einer vollen Hashtabelle noch eine Endlossuche abfangen muss, da die Suche nie an einer leeren Stelle terminieren kann.
•*Entfernen* eines Schlüssels k: Suche k in der Hashtabelle. Wenn k an der Stelle $t(i)$ gefunden wurde, markiere die Stelle $t(i)$ als gelöscht. Bemerkung: Die Stelle $t(i)$ darf nicht leer werden, da die Suche sonst nicht mehr funktioniert!
Aus Performancegründen kann es sinnvoll sein die Tabelle nach vielen Löschungen neu aufzubauen.

Analyse: Eine Analyse der Effizienz des linearen Sondierens zeigt, dass für die durchschnittliche Anzahl der bei erfolgreicher bzw. erfolgloser Suche betrachteten Einträge C_n bzw. C'_n gilt:

$$C_n \approx \frac{1}{2}(1 + \frac{1}{1-\alpha})$$

$$C'_n \approx \frac{1}{2}(1 + \frac{1}{(1-\alpha)^2})$$

Beispiel: Grösse der Hashtabelle m=7, $\mathcal{K} = 0, 1, .., 500$, $h(k) = k \ mod \ m$, $s(j,k) = j$ (lineares Sondieren).

Einfügen der Schlüssel 12, 53, 5, 15, 2, 19 in eine leere Hashtabelle. Die Schlüssel 12 und 53 lassen sich dabei direkt einfügen

	0	1	2	3	4	5	6
t:					53	12	

Einfügen von 5 ergibt h(5) = 5 mod 7 = 5, dieser Platz ist belegt. Der nächste Index der Sondierungsfolge ist 4, ebenfalls belegt; der nächste Index ist 3, nicht belegt. Die 5 wird beim Index 3 eingefügt.

	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	

Wenn wir jetzt die 53 löschen wollen, markieren wir den Index als gelöscht und entfernen die 53. (* markiert den Eintrag als gelöscht)

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

Jetzt versuchen wir die 5 zu finden. h(5) = 5 mod 7 = 5, dieser Platz ist belegt, also suchen wir bei Position 4 weiter. Dieser Platz ist frei, ABER als gelöscht markiert. Darum wird die Suche bei Position 3 fortgesetzt, wo die 5 gefunden wird. Wäre die Position 4 nicht als gelöscht markiert gewesen, hätten wir die 5 nicht gefunden!

Einfügen von 33: 33 *mod* 7 = 5. Dieser Platz ist belegt. Lineares Sondieren liefert im ersten Schritt den freien (als gelöscht markierten) Platz 4. Einfügen der 33 am Index 4.

	0	1	2	3	4	5	6
t:	19	15	2	5	33	12	

9.4 Dynamische Hashverfahren

Die bisherigen *statischen Hashverfahren* hatten den Nachteil, dass sie *für die Zukunft* angelegt werden mussten. Anfangs hat man leere Seiten, die dann sukzessiv gefüllt werden. Überschreitet bei offenen Verfahren das Datenvolumen die Kapazität der Hashtabelle, ist eine Reorganisation nötig. Das Umorganisieren ist sehr zeitaufwändig, da alle Einträge neu gehasht werden müssen.
Im Gegensatz dazu stehen die *dynamischen Hashverfahren*. Dort wird die Tabelle bei Bedarf erweitert. Der Idealzustand ist der folgende:
Die Hashtabelle enthält jeweils einen Zeiger auf höchstens ein Bucket[5] und es gibt keine Überlaufbuckets. Wenn ein Überlauf eintritt, wird die Tabelle erweitert. Die Tabelle mit den Zeigern residiert dabei komplett im Hauptspeicher, die Buckets dagegen im Sekundärspeicher.

9.4.1 Abbildung des Hashwertes auf einen Baum

Ein Teil des Hashwertes wird als Mittel zur Navigation in einem Baum benutzt, die Blätter zeigen auf Buckets mit den Daten.
Wenn ein Überlauf eintritt, wird die Höhe des Baumes lokal vergrößert. (Beispiel später)
Ergebnisse:
• Wenn die Hashfunktion nahezu eine Gleichverteilung erzeugt, ist der Baum nahezu ausgeglichen.
• Nicht nur für die Geometrie anwendbar, auch Geometrie und Sachdaten gemeinsam.
• Splitt: die Daten eines Buckets werden nach der nächsten Stelle des Hashwertes aufgeteilt.
• Es entstehen keine Überlauf-Buckets, die nicht in die Zugriffsstruktur passen.
• Güte des Verfahrens ist von der Hashfunktion abhängig.
• Splitprozess: es ist nicht vorhersehbar, welches Bucket als nächstes überläuft.
• Wenn ein Bucket voll ist, muss es geteilt werden, unabhängig von der Arbeitslast auf der Maschine. Ausweg: Füllgrad > festgelegte Grenze und Rechenzeit vorhanden $\mapsto$ Teilung
• Hashwert n-Bit $\mapsto$ 2^n Buckets adressierbar; Bucketgröße fest $\mapsto$ Maximale Anzahl der Datensätze fest. (Nachteil)
• Finden eines Buckets der Tiefe d' erfordert d' Vergleichsoperationen.

$$d' \leq \log_2(\frac{N}{C})$$

[5]Bucket: Eine Speichereinheit, die eine bestimmte Anzahl an Einträgen aufnehmen kann, z.B. eine Seite einer Festplatte.

N Anzahl der Einträge; C Kapazität eines Buckets (in Einträgen gemessen)

Für die Beispiele wird eine Bucketgröße von 4 angenommen.

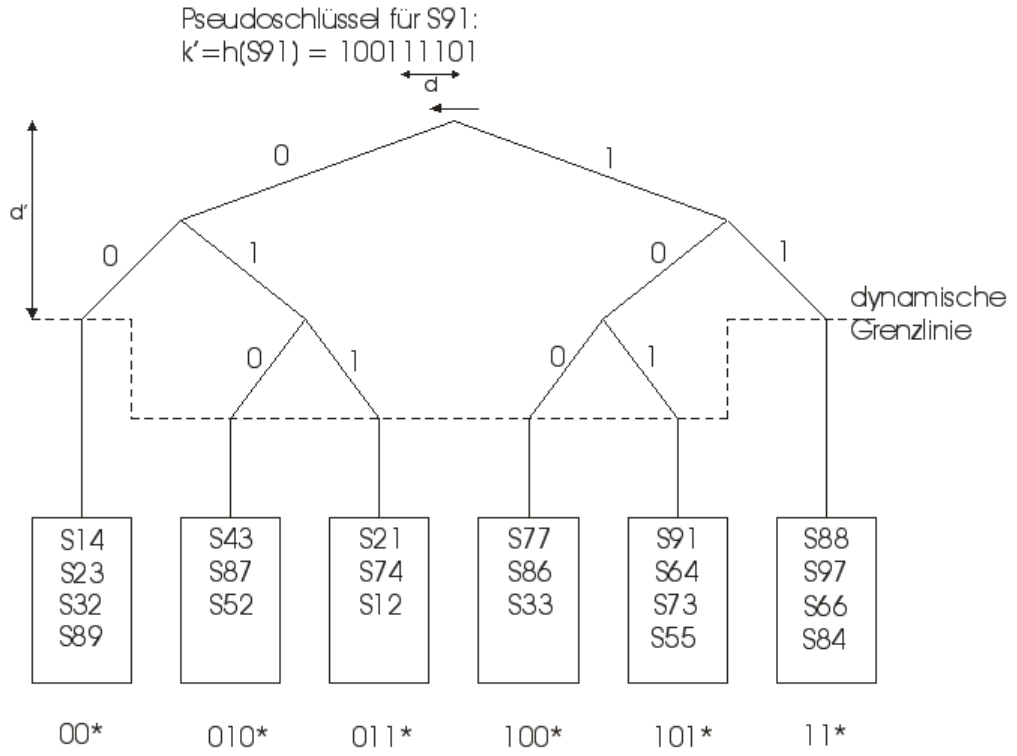


Abbildung 9.3: Prinzipielle Abbildung der Pseudoschlüssel auf einem Binärbaum

Abbildung 9.3 sei unser Ausgangsbaum, in den wir den Schlüssel $S22$ einfügen wollen. Dazu bilden wir als erstes den Pseudoschlüssel von $S22$: $k' = h(S22) =$ 1 0011 0111. Damit muss $S22$ in das Bucket $11*$. Allerdings ist das Bucket bereits mit 4 Einträgen voll, so dass wir das Bucket teilen müssen. Damit entstehen 2 neue Buckets, das erste $110*$ Bucket mit $S88$ und $S97$ sowie das $111*$ Bucket mit $S66$, $S84$ und $S22$.
Daraus ergibt sich der Binärbaum, der in Abbildung 9.4 dargestellt ist.

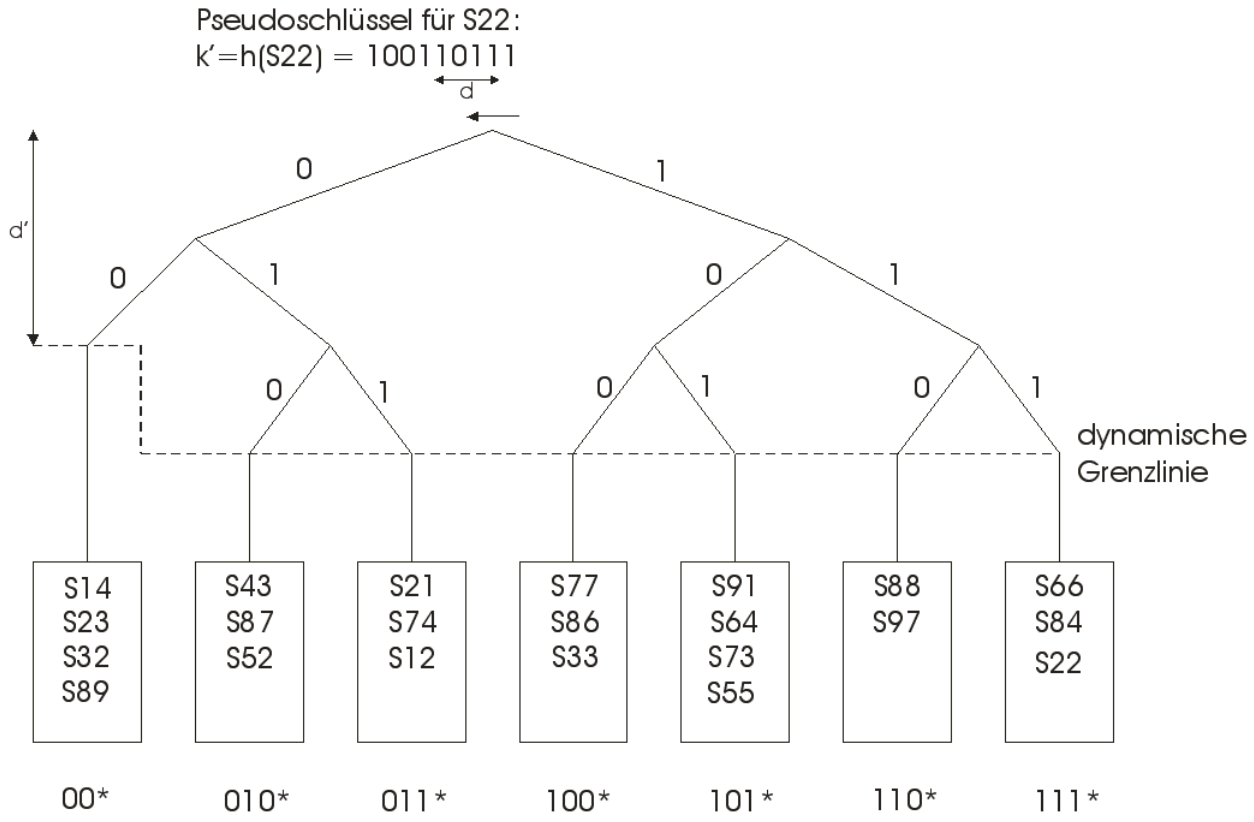


Abbildung 9.4: Das Beispiel aus Abbildung 3 nach dem Einfügen von $S22$.

9.4.2 Erweiterbares Hashing

Unterschied zum vorherigen: Navigationsstruktur ist flach, bzw. hat eine feste Tiefe. (Abbildung auf Seite 14)
Vorteil: Ein Leseprozess im Directory und das richtige Bucket ist bekannt.
Nachteil: Wie schon im Verfahren des Kapitel 9.4.1, ist es nicht vorhersehbar, welches Bucket gesplittet wird, auch der Zeitpunkt kann nicht bestimmt werden.

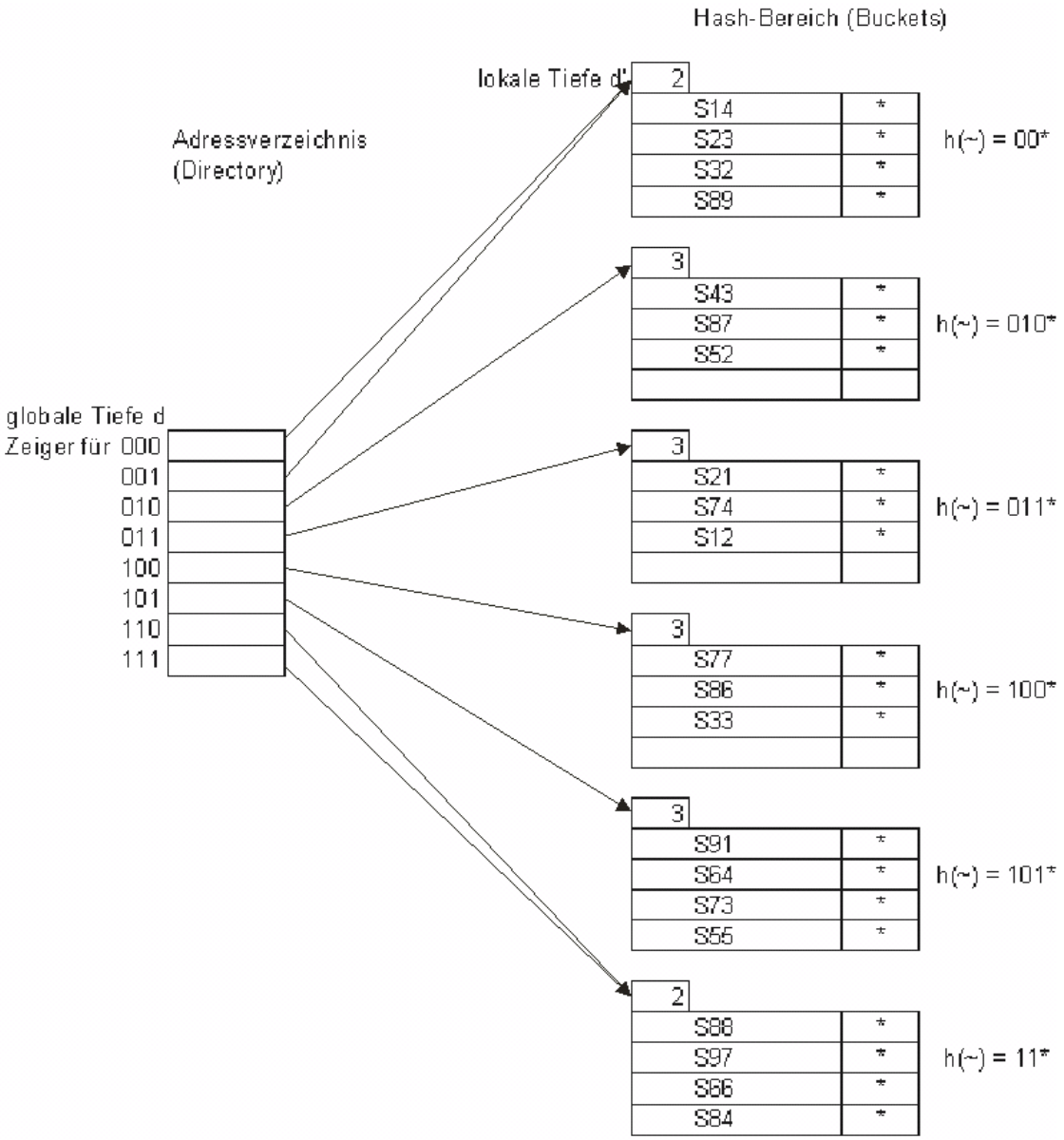


Abbildung 9.5: Gestreute Speicherungsstruktur beim Erweiterbaren Hashing. Darstellung des Anwendungsbeispiels

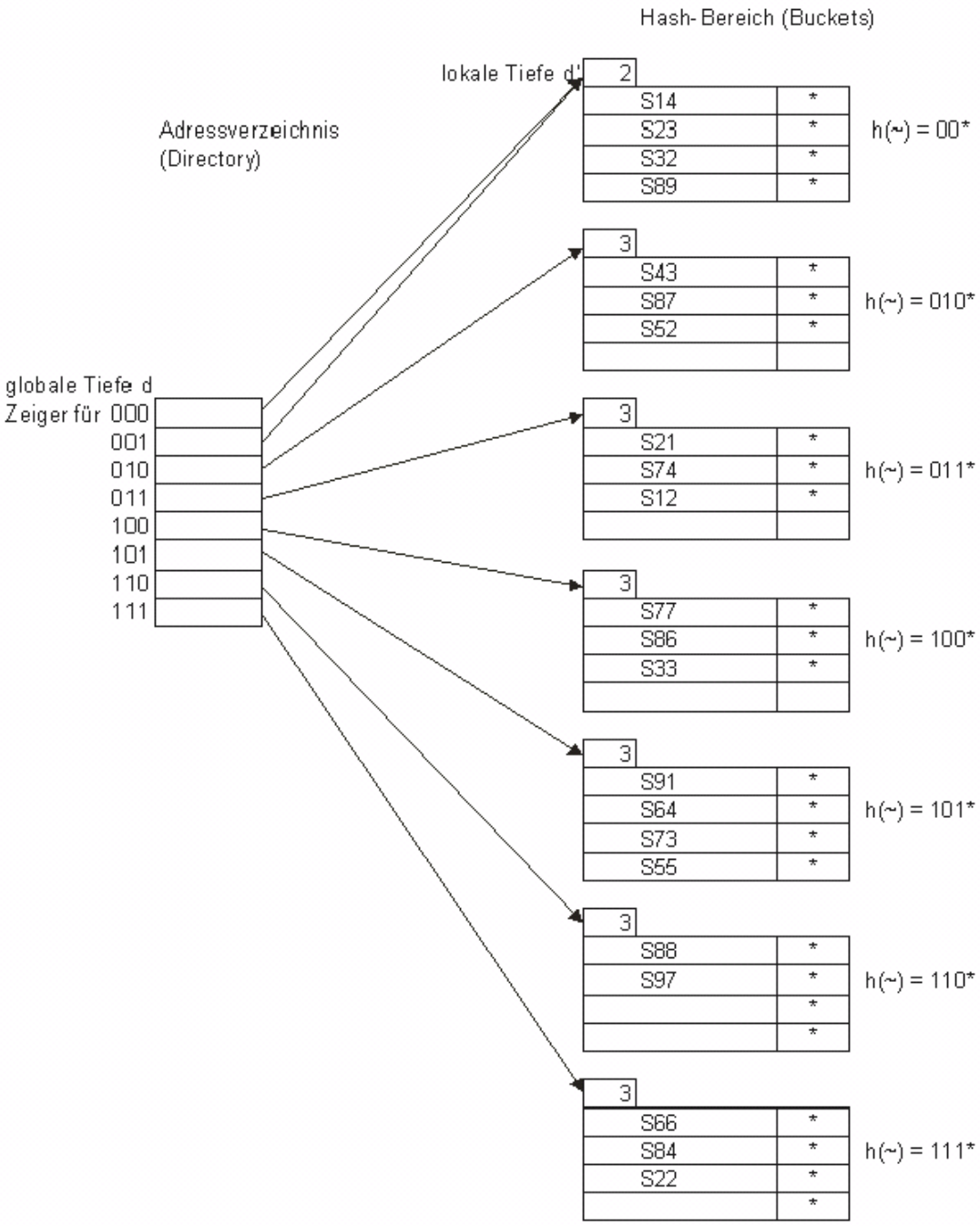


Abbildung 9.6: Gestreute Speicherungsstruktur nach dem Einfügen von $S22$

Diese Darstellung ermöglicht jetzt aber den Einsatz einer effizienten Suchstruktur. egal welche Attribute der Geo-Datenbank zur Bildung der Hashwerte verwendet wurden, jetzt sind die Hashwerte alle gleich lang, damit kann man sie als Zahlen interpretieren und mit diesen einen B*-Baum aufbauen.

9.4.3 Lineares Hashing

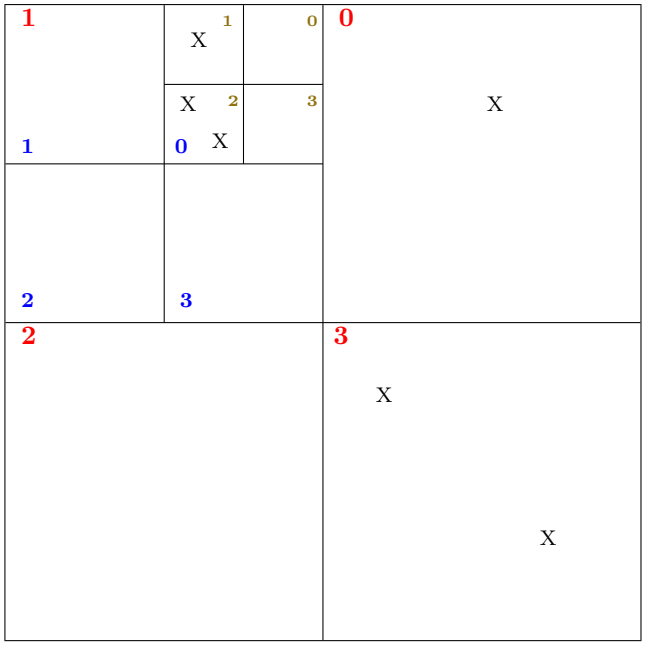
grundsätzliche Neuerung:
Entkopplung des Bucket-splitting vom Überlauf, d.h. es gibt Überlaufbuckets die (hoffentlich) später abgebaut werden.
1. Füllgrad $= \frac{Zahl\ der\ Datensätze}{(Zahl\ Datensätze/Bucket)*(Anzahl\ der\ Reg.+Überlaufbuckets)}$
2. Splitting: Es gibt einen Splittingzeiger, der auf einen der $0..(2^n-1)$ Buckets zeigt. k sei addressiert.
Wenn der Füllgrad T ein vorgegebenes Level übersteigt, wird der k-te Bucket geteilt. Der Inhalt wird mit einer verbesserten Hashfunktion h_{n+1} auf die Buckets k und $2^n + k$ verteilt. Der Splitzeiger wird um eins erhöht.
Wurde ein leeres Bucket geteilt, hat man Pech gehabt. Wurde aber ein Bucket mit Überlauf geteilt, kann ein Überlaufbucket entfernt werden.
Das Verfahren sichert, dass jedes Bucket reihum geteilt wird. Dann hat man 2^{n+1} Buckets. Da die 100%-ige Auslastung und das Splitten entkoppelt sind, kann es passieren, dass leere Buckets splitten bzw. dass Überlaufbuckets bestehen bleiben. Es besteht aber die Hoffnung, dass bei Gleichverteilung die Überlaufbuckets regelmäßig aufgelöst werden. Das Verfahren versagt bei schlechter Hashfunktion, denn diese führt zu einer 'schiefen' Lage der Hashwerte.
3. Merging: Konsequenterweise muss beim Unterschreiten des Füllgrades ein Merging stattfinden - dazu gibt es wenige Untersuchungen.

9.4.4 Hashing mit mehreren Attributen

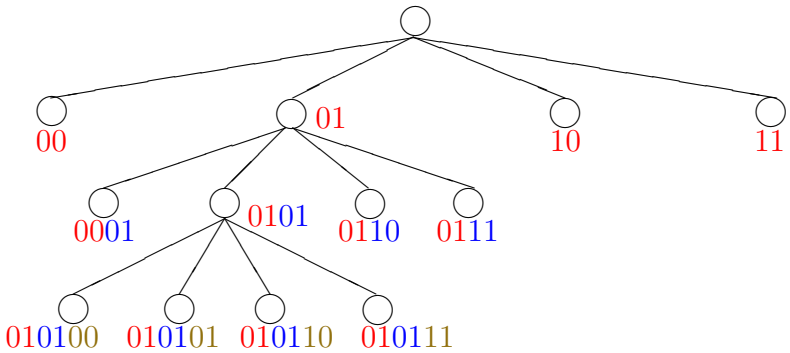
Bisher hatten wir Hashing nur mit einem Attribut (skalar). Wir wollen aber mehrdimensional speichern. Bei allen Lösungsansätzen bleibt das Problem der Nichterhaltung von Nachbarschaftsbeziehungen. Eine Darstellung des Problems findet sich in Rahm/Härder. Ein Lösungsansatz: Verhashen jedes Attributs und Verbinden der Hashwerte. Die Unterscheidung erfolgt nach der Art der Verbindung.
1: einfache Verbindung des Schlüsselhashs
$h(x)$ und $h(y)$ als Gesamtfunktion. D.h. man hasht als erstes mit $h(x)$ und dann mit $h(y)$. Der Schlüssel ergibt sich als Konkatenation der Teilergebnisse. Unterstützt sehr gut exakte Anfragen. $\Longrightarrow$ Ähnlichkeit mit invertierter Liste - eine Komponente bevorzugt.
2: Bitinterleaving / Bitverschachtelung:
$h_1(x) = x_1x_2..x_n$
$h_2(y) = y_1y_2..y_n$;evt. $h_i = i$ Identität

Bilde $k = x_1y_1x_2y_2x_3y_3..x_ny_n$ Verallgemeinerung auf n verschiedene Attribute. $\Longrightarrow$ Zugriffsstruktur auf einen Quadtree mit z-Fädelung. (siehe Beispiel)
Jede Ebene des Quadtrees hat 2 Bit. Damit hat man unterschiedlich lange Bitfolgen, aber die Bitfolgen entsprechen den Adressen der Blattknoten. Diese werden dann lexikographisch sortiert.
Mit dieser lexikographischen Ordnung kann man einen B*-Baum aufbauen. Dieser verzweigt sich nicht mit 4 Söhnen, sondern mit ≈ 100.
Quadtree: logische Teilung, evt. Nachbarschaft
B*-Baum: physischer Zugriff

Beispiel: $h_1(x) = x_1, x_2, .., x_n$ und $h_2(y) = y_1, y_2, .., y_n$.
$k = x_1y_1x_2y_2x_3y_3..x_ny_n$



Der zugehörige Quadtree hat die folgende Gestalt:



9.5 Literatur

Ottmann / Widmayer: Algorithmen und Datenstrukturen; Spektrum Akademischer Verlag; Heidelberg, Berlin 2002

Rahm / Härder: Datenbanksysteme. Konzepte und Techniken der Implementierung; 2. Auflage; Springer-Verlag; Berlin, Heidelberg 2001

Rigaux / Scholl / Voisard: Spatial Databases with Application to GIS; 2.Auflage; Morgan Kaufmann Publishers; San Francisco, Kalifornien, USA 2001