Kapitel 4

Hashverfahren

In den Kapiteln 1 und 3 haben wir einige Methoden kennengelernt, die es erlauben, eine Menge von Datensätzen so zu speichern, daß die Operationen Suchen, Einfügen und Entfernen unterstützt werden. Jeder Datensatz ist dabei gekennzeichnet durch einen eindeutigen Schlüssel. Zu jedem Zeitpunkt ist lediglich eine (kleine) Teilmenge K aller möglichen Schlüssel K (englisch: keys) gespeichert. Statt nun bei der Suche nach einem Datensatz mit Schlüssel k mehrere Schlüsselvergleiche mit Schlüsseln aus K auszuführen, wird bei Hash-Verfahren versucht, durch eine *Berechnung* festzustellen, wo der Datensatz mit Schlüssel k gespeichert ist. Die Datensätze werden in einem linearen Feld mit Indizes $0,\ldots,m-1$ gespeichert; dieses Feld nennt man die *Hashtabelle*, m ist die *Größe* der Hashtabelle. Eine Abbildung, die *Hashfunktion* $h : K \to \{0,\ldots,m-1\}$ ordnet jedem Schlüssel k einen Index $h(k)$ mit $0 \leq h(k) \leq m-1$ zu, die *Hashadresse*. Im allgemeinen ist K eine sehr kleine Teilmenge von K; so treten etwa in einem Pascal-Programm nur wenige der $\approx 26 \cdot 36^{79}$ zulässigen Namen auf (ein Name beginnt mit einem der 26 Buchstaben, danach kommen bis zu 79 weitere Buchstaben oder Ziffern, wenn eine Programmzeile bis zu 80 Zeichen lang ist). Die Hashfunktion kann also im allgemeinen nicht injektiv sein, sondern muß verschiedene Schlüssel auf dieselbe Hashadresse abbilden. Zwei Schlüssel k, k' mit $h(k) = h(k')$ heißen *Synonyme*; befinden sich beide Schlüssel in der aktuellen Schlüsselmenge K, so ergibt sich eine *Adreßkollision*. Treten in K keine Synonyme auf, so kann jeder Datensatz in der Hashtabelle an der seiner Hashadresse entsprechenden Stelle gespeichert werden. Bei Adreßkollisionen hingegen muß eine Sonderbehandlung vorgenommen werden.

Ein Hashverfahren muß also zwei Forderungen genügen. Erstens sollen möglichst wenige Kollisionen auftreten. Dies kann angestrebt werden durch die Wahl einer „guten“ Hashfunktion. Zweitens sollen Adreßkollisionen möglichst effizient aufgelöst werden.

Die Wahl der Hashfunktion werden wir im Abschnitt 4.1 diskutieren. Die Abschnitte 4.2 und 4.3 sind ganz der Diskussion von Strategien zur Kollisionsauflösung unter verschiedenen Annahmen gewidmet. Weil auch die beste Hashfunktion Kollisionen nicht ganz 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 etwa die Zeit zum Suchen eines Schlüssels nicht abhängig von der Anzahl der gespei-

cherten Schlüssel, vorausgesetzt, daß genügend viel Speicherplatz zur Verfügung steht. Für eine Hashtabelle der Größe m, die gerade n Schlüssel speichert, nennen wir den Quotienten aus n und m, $\alpha = n/m$, den *Belegungsfaktor* der Tabelle. Die Anzahl der zum Suchen, Einfügen oder Entfernen eines Schlüssels benötigten Schritte hängt im wesentlichen vom Belegungsfaktor α ab. Dabei muß man bei manchen Hashverfahren annehmen, daß nur wenige Entferne-Operationen durchgeführt worden sind, weil diese auch im Mittel die Effizienz nachhaltig beeinträchtigen. Hashverfahren sind also gerade dann besonders effizient, wenn nach vielen anfänglichen Einfügeoperationen fast nur noch gesucht und fast nicht entfernt wird.

Im folgenden legen wir der Beschreibung der Hash-Verfahren die Definition der Hashtabelle als Feld von Datensätzen zugrunde:

```
const
  m = {eine geeignete positive ganze Zahl};
type
  datensatz = record
                k : key;
                item : itemtype
              end;
  hashadresse = 0 .. m − 1;
  hashtabelle = array [hashadresse] of datensatz;
var
  t : hashtabelle;
```

Wegen der fest gewählten Größe der Hashtabelle sind die meisten der von uns präsentierten Verfahren nur halbdynamisch. Es können nie mehr als m Datensätze (kollisionsfrei, falls zusätzlicher Speicherplatz verwendet wird) gespeichert sein. Hiervon unterscheiden sich *dynamische Hashverfahren*, bei denen die Größe der Adreßtabelle (in Sprüngen) variabel ist; wir werden solche Verfahren in Abschnitt 4.4 behandeln. Im Abschnitt 4.5 schließlich präsentieren wir ein populäres Hashverfahren für mehrdimensionale Schlüssel, das ebenfalls dynamisch ist und sich für eine Realisierung auf Externspeichermedien mit Direktzugriff eignet, das *Gridfile*.

Bei der Analyse der Effizienz von Hashverfahren geht es uns in erster Linie um die durchschnittliche Laufzeit für die Operationen Suchen, Einfügen und Entfernen. Im schlimmsten Fall sind diese Operationen extrem langsam; dieser Fall ist leicht direkt aus der Beschreibung der Verfahren ableitbar. Wir werden stets zwei Erwartungswerte C_n und C'_n angeben, bezogen auf feste Tabellengröße m. Dabei ist C_n der Erwartungswert für die Anzahl der betrachteten Einträge der Hashtabelle bei erfolg*reicher* Suche, C'_n der Erwartungswert für die Anzahl der betrachteten Einträge der Hashtabelle bei erfolg*loser* Suche. Welche Wahrscheinlichkeitsverteilung unserer Rechnung zugrundeliegt, werden wir jeweils an Ort und Stelle erläutern.

Dem Entfernen eines Datensatzes muß stets eine erfolgreiche Suche vorausgehen; entsprechend ist der Aufwand für das Entfernen gerade C_n, wenn der betreffende Eintrag lediglich als entfernt markiert wird. Dem Einfügen eines Datensatzes muß stets eine erfolglose Suche vorausgehen; entsprechend ist der Einfüge-Aufwand gerade C'_n, wenn der betreffende Datensatz einfach an der ersten gefundenen freien Stelle eingetragen wird. Betrachten wir jedoch zunächst mögliche Hashfunktionen etwas genauer.

4.1 Zur Wahl der Hashfunktion

Eine gute Hashfunktion sollte möglichst leicht und schnell berechenbar sein und die zu speichernden Datensätze möglichst gleichmäßig auf den Speicherbereich verteilen, um Adreßkollisionen zu vermeiden.

Die von der Hashfunktion zu gegebenen Schlüsseln gelieferten Hashadressen sollten also über dem Adreßbereich gleichverteilt sein, und zwar selbst dann, wenn die Schlüssel aus K alles andere als gleichverteilt sind (etwa bei der Vorliebe von Programmierern für Namen wie x, $x1$, $x2$, $y1$, $y2$, $z1$, $z2$). Daß dennoch Adreßkollisionen selbst bei einer optimal gewählten Hashfunktion wahrscheinlich sind, zeigt das *Birthday Paradox*, vgl. [52]: Wenn 23 Personen oder mehr in einem Raum sind, haben wahrscheinlich zwei davon am gleichen Tag des Jahres Geburtstag. Allgemeiner gilt: Wenn eine Hashfunktion $\sqrt{\pi m/2}$ Schlüssel auf eine Hashtabelle der Größe m abbildet, dann gibt es wahrscheinlich eine Adreßkollision (für $m = 365$ ist $\lfloor\sqrt{\pi m/2}\rfloor = 23$).

Wir werden im folgenden von nichtnegativen ganzzahligen Schlüsseln ausgehen, also $K \subseteq \mathbb{N}_0$ annehmen. Wenn Schlüssel zunächst als Zeichenfolgen gegeben sind (wie im Beispiel der Namen in Pascal-Programmen), so interpretieren wir die ihnen entsprechenden Bitfolgen einfach als positive ganze Zahlen, etwa im Dualsystem. Dann ist klar, daß eine Hashfunktion nicht nur gleichverteilte Schlüssel möglichst gleichmäßig auf den Adreßbereich streuen muß, sondern auch Häufungen (englisch: cluster) fast gleicher Schlüssel aufbrechen muß.

4.1.1 Die Divisions-Rest-Methode

Ein naheliegendes Verfahren zur Erzeugung einer Hashadresse $h(k)$, $0 \leq h(k) \leq m-1$, zu gegebenem Schlüssel $k \in \mathbb{N}_0$ ist es, den Rest von k bei ganzzahliger Division durch m zu nehmen:

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

Dann ist allerdings eine gute Wahl von m entscheidend. 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 $r^i \pm j$, 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 $r^i \pm j$ teilt. Diese Wahl hat sich in praktisch allen Fällen ausgezeichnet bewährt (vgl. [89]).

4.1.2 Die multiplikative Methode

Der gegebene Schlüssel wird mit einer irrationalen Zahl multipliziert; der ganzzahlige Anteil des Resultats wird abgeschnitten. Auf diese Weise erhält man für verschiedene Schlüssel verschiedene Werte zwischen 0 und 1; für Schlüssel $1, 2, 3, \ldots, n$ sind diese Werte ziemlich gleichmäßig im Intervall $[0, 1)$ verstreut, wie ein Satz von Vera Turán Sós [185] (vgl. auch [89]) zeigt:

> Sei Θ eine irrationale Zahl. Plaziert man die Punkte $\Theta - \lfloor\Theta\rfloor$, $2\Theta - \lfloor 2\Theta\rfloor$, $3\Theta - \lfloor 3\Theta\rfloor$, $\ldots$, $n\Theta - \lfloor n\Theta\rfloor$ in das Intervall $[0, 1]$, dann haben die $n+1$ Intervallteile höchstens drei verschiedene Längen. Außerdem fällt der nächste Punkt, $(n+1)\Theta - \lfloor(n+1)\Theta\rfloor$, in einen der größten Intervallteile.

Von allen Zahlen Θ, $0 < \Theta \leq 1$, führt der *goldene Schnitt*

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

zur gleichmäßigsten Verteilung. Damit erhalten wir folgende Hashfunktion:

$$h(k) = \left\lfloor m \left(k\phi^{-1} - \lfloor k\phi^{-1} \rfloor \right) \right\rfloor$$

Insbesondere bilden die Werte $h(1), h(2), \ldots, h(10)$ für $m = 10$ gerade eine Permutation der Zahlen $0, 1, \ldots, 9$, nämlich 6, 2, 8, 4, 0, 7, 3, 9, 5, 1. Der Leser kann sich selbst davon überzeugen, daß jede dieser Hashadressen, in der gegebenen Reihenfolge betrachtet, in ein größtes Intervall zwischen zwei bereits betrachteten Hashadressen fällt und dieses Intervall gemäß dem goldenen Schnitt teilt.

Man kann die Berechnung von $h(k)$ noch beschleunigen, wenn man ganze Zahlen im Rechner als Bruchzahlen mit Dezimalpunkt vor der höchstwertigen Ziffer ansieht, und wenn man für m eine Zweierpotenz wählt; dann läßt sich die Berechnung von $h(k)$ mit einer ganzzahligen Multiplikation und einer (oder zwei) Shift-Operation(en) vornehmen. Wir wollen dies hier nicht im einzelnen erläutern; der interessierte Leser sei verwiesen auf [89] oder [175].

Neben diesen beiden Methoden gibt es noch zahlreiche andere, die z.B. nach einer Transformation des Schlüssels (in ein anderes Zahlensystem oder durch Quadrieren oder durch Falten auf kurze Länge mit Verknüpfen von Teilstücken) einzelne Ziffernpositionen auswählen. Lum, Yuen und Dodd [113] haben das Verhalten einer Reihe verschiedener Hashfunktionstypen studiert. Dazu gehören das Divisions-Rest-Verfahren, die multiplikative Methode, die Ziffernanalyse, die Mid-square-Methode, die Faltung und die algebraische Verschlüsselung. Sie haben festgestellt, daß das Divisions-Rest-Verfahren im Durchschnitt die besten Resultate lieferte. Wir werden daher ab Abschnitt 4.2 stets eine nach dem Divisions-Rest-Verfahren arbeitende Hashfunktion verwenden, wenn wir Hashverfahren und damit Strategien zur Kollisionsauflösung beschreiben.

4.1.3 Perfektes und universelles Hashing

Ist die Anzahl der zu speichernden Schlüssel nicht größer als die Anzahl der zur Verfügung stehenden Speicherplätze, gilt also für die Teilmenge K der Menge $\mathcal{K}$ aller möglichen Schlüssel $|K| \leq m$, so ist eine kollisionsfreie Speicherung von K immer möglich. Wenn wir K kennen und K fest bleibt, können wir leicht eine injektive Abbildung $h : K \to \{0,..,m-1\}$ z.B. wie folgt berechnen: Wir ordnen die Schlüssel in K lexikographisch und bilden jeden Schlüssel auf seine Ordnungsnummer ab. Wir haben damit eine *perfekte* Hashfunktion, die Kollisionen gänzlich vermeidet. Eine solche Situation (K fest und vorher bekannt) liegt z.B. dann vor, wenn den Schlüsselworten einer Programmiersprache feste Plätze in einer Symboltabelle zugeordnet werden sollen. Dieser Fall ist aber eher die Ausnahme als die Regel. Im allgemeinen kennen wir $K \subseteq \mathcal{K}$ nicht und können selbst dann, wenn $|K| \leq m$ bleibt, nicht sicher sein, daß Kollisionen vermieden werden.

Bleiben wir beim Beispiel der Verwaltung von Tabellen durch Compiler. Es könnte z.B. sein, daß eine vom Compiler fest gewählte Zuordnung von benutzerdefinierten Namen zu Plätzen in einer Symboltabelle auf besondere Vorlieben eines Programmierers für die Wahl von Namen keine Rücksicht nimmt und daher jedesmal zu vielen Kollisionen führt. Denn sobald die Hashfunktion fest gewählt ist, kann man stets viele Schlüssel finden, die sämtlich auf dieselbe Hashadresse abgebildet werden. Die einzige Möglichkeit, diese sehr unerwünschte Situation zu vermeiden, ist, die Hashfunktion zufällig aus einer sorgfältig gewählten Menge von Hashfunktionen auszuwählen. Statt anzunehmen, daß die aktuelle Schlüsselmenge K zufällig und gleichverteilt im Universum $\mathcal{K}$ aller möglichen Schlüssel gewählt wird, machen wir also eine wesentlich weniger kritische Annahme über das Hashverfahren: Wir nehmen an, daß die vom Verfahren benutzte Hashfunktion h zufällig und gleichverteilt aus einer Menge $\mathcal{H}$ möglicher Hashfunktionen gewählt wird. Die Auswahl von h ist Teil des Verfahrens und unterliegt, ganz anders als die Auswahl von $K \subseteq \mathcal{K}$, nicht einer möglicherweise sehr einseitigen Vorliebe des Benutzers. Diese Art der Randomisierung garantiert daher (ganz ähnlich wie bei randomisiertem Quicksort), daß eine schlecht gewählte Schlüsselmenge K nicht jedesmal zu vielen Kollisionen führt: Zwar kann eine einzelne Funktion $h \in \mathcal{H}$ noch immer viele Schlüssel aus K auf dieselbe Adresse abbilden. Gemittelt über alle Funktionen aus $\mathcal{H}$ ist das aber 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.

Betrachten wir also ein beliebiges, festes Paar von zwei verschiedenen Schlüsseln x und y. Dann ist die Wahrscheinlichkeit dafür, daß x und y von einer zufällig aus H gewählten Funktion h auf dieselbe Hashadresse abgebildet werden, höchstens $1/m$. Denn höchstens $1/m$ der Funktionen aus H führen zu einer Adreßkollision bei x und y.

Wir definieren eine Funktion δ, die für zwei Schlüssel x und y aus K und eine Hashfunction $h \in H$ anzeigt, ob eine Kollision vorliegt:

$$\delta(x,y,h) = \begin{cases} 1 & \text{falls } h(x) = h(y) \text{ und } x \neq y \\ 0 & \text{sonst} \end{cases}$$

Man kann δ wie folgt auf Mengen $Y \subseteq K$ von Schlüsseln und auf ganz H ausdehnen:

$$\begin{aligned} \delta(x,Y,h) &= \sum_{y \in Y} \delta(x,y,h) \\ \delta(x,y,H) &= \sum_{h \in H} \delta(x,y,h) \end{aligned}$$

Offenbar ist H universell, wenn für je zwei beliebige $x,y \in K$ mit $x \neq y$ gilt: $\delta(x,y,H) \leq |H|/m$.

Wir überlegen uns zunächst, welche Vorteile es hat, mit einer universellen Klasse H von Hashfunktionen zu arbeiten, bevor wir die Existenz solcher Klassen nachweisen. Nehmen wir an, wir wollen eine (vorher nicht bekannte) Folge von Schlüsseln aus dem Universum K aller möglichen Schlüssel in die Hashtabelle der Größe m, also auf eine der Adressen $\{0,\ldots,m-1\}$ abbilden. Sei H eine universelle Klasse von Hashfunktionen $h : K \to \{0,\ldots,m-1\}$. Dann wählen wir eine Funktion $h \in H$ zufällig aus und bilden mit ihr die Schlüssel der Reihe nach auf die Hashadressen ab. Die Hashfunktion bleibt also bei der ganzen Folge von Einfügungen fest. Soll ein Schlüssel x an der Stelle $h(x)$ gespeichert werden, so kann es natürlich sein, daß dieser Platz bereits besetzt ist. Nehmen wir an, daß zum Zeitpunkt des Einfügens von x in der Hashtabelle bereits die Menge S von Schlüsseln gespeichert ist und jeweils alle Schlüssel mit derselben Hashadresse in je einer linearen Liste zusammengefaßt werden. Es ist vernünftig, als Maß für den Aufwand zum Einfügen von x in die Hashtabelle die Anzahl der Elemente aus S zu nehmen, mit denen x kollidiert (Das wird im folgenden Abschnitt 4.2 genauer erläutert). Um diesen Aufwand abzuschätzen, berechnen wir den Erwartungswert $E[\delta(x,S,h)]$:

$$\begin{aligned} E[\delta(x,S,h)] &= \sum_{h \in H} \delta(x,S,h)/|H| \\ &= \frac{1}{|H|} \sum_{h \in H} \sum_{y \in S} \delta(x,y,h) \\ &= \frac{1}{|H|} \sum_{y \in S} \sum_{h \in H} \delta(x,y,h) \\ &= \frac{1}{|H|} \sum_{y \in S} \delta(x,y,H) \\ &\leq \frac{1}{|H|} \sum_{y \in S} |H|/m \\ &= |S|/m \end{aligned}$$

Man kann also erwarten, daß eine aus einer universellen Klasse H von Hashfuntionen zufällig gewählte Funktion h eine beliebige, noch so „einseitig“ gewählte Folge von Schlüsseln des Universums K so gleichmäßig wie nur möglich über die zur Verfügung stehenden Adressen verteilt.

Wir wollen jetzt zeigen, daß universelle Klassen von Hashfunktionen existieren und sogar relativ leicht konstruiert werden können. Dazu nehmen wir an, daß alle Schlüssel nichtnegative ganze Zahlen sind und $|K| = p$ eine Primzahl ist, d.h. wir setzen zur Vereinfachung $K = \{0,\ldots,p-1\}$ voraus.

Für zwei beliebige Zahlen $a \in \{1,\ldots,p-1\}$ und $b \in \{0,\ldots,p-1\}$ sei die Funktion $h_{a,b} : K \to \{0,\ldots,m-1\}$ wie folgt definiert:

$$h_{a,b}(x) = ((ax+b) \bmod p) \bmod m.$$

Dann gilt:

Satz 4.1 *Die Klasse $H = \{h_{a,b} \mid 1 \le a < p$ und $0 \le b < p\}$ ist eine universelle Klasse von Hashfunktionen.*

Zum Beweis überlegen wir uns zunächst, daß für festes x und y, $0 \le x, y < p$, $x \neq y$, die Zahlenpaare (r,q) mit $r = (ax+b) \bmod p$ und $q = (ay+b) \bmod p$ den gesamten möglichen Bereich aller Paare (r,q) mit $0 \le r, q < p$ und $r \neq q$ durchlaufen, wenn a und b im gesamten Bereich $1 \le a < p$ und $0 \le b < p$ gewählt werden. Denn erstens ist für jedes zulässige a und b das Paar (r,q) mit $r = (ax+b) \bmod p$ und $q = (ay+b) \bmod p$ ein zulässiges Paar (r,q) mit $0 \le r, q < p$ und $r \neq q$, da nach Voraussetzung $x \neq y$ und p prim ist. Zweitens gilt auch für jedes Paar (r,q) mit $0 \le r, q < p$ und $r \neq q$, daß sich r und q schreiben lassen in der Form $r = (ax+b) \bmod p$ und $q = (ay+b) \bmod p$ mit geeignet gewählten a und b im zulässigen Bereich $1 \le a < p$ und $0 \le b < p$. Denn weil p eine Primzahl ist, kann im Ring der Restklassen von p das System von zwei linearen Gleichungen $r \equiv (ax+b) \bmod p$ und $q \equiv (ay+b) \bmod p$ eindeutig nach a und b aufgelöst werden.

Nun gilt für eine Funktion $h_{a,b} \in H$, daß sie x und y auf dieselbe Hashadresse abbildet, also $h_{a,b}(x) = h_{a,b}(y)$, genau dann, wenn $(ax+b) \equiv (ay+b) \bmod m$ ist. Um abzuschätzen, wieviele Funktionen aus H x und y auf dieselbe Adresse abbilden, genügt es also abzuschätzen, für wieviele Paare (q,r) mit $0 \le q$, $r < p$ und $q \neq r$ die Zahlen q und r in dieselbe Restklasse modulo m fallen. Für festes q, $0 \le q < p$, kann es offenbar höchstens $(p-1)/m$ Zahlen $r \neq q$ geben mit $q \equiv r \bmod m$. Damit gibt es unter den $p \cdot (p-1)$ Zahlenpaaren (q,r) mit $q \equiv (ax+b) \bmod p$ und $r \equiv (ay+b) \bmod p$, $1 \le a < p$, $0 \le b < p$, höchstens $p(p-1)/m$ viele, für die q und r in dieselbe Restklasse modulo m fallen. Also ist

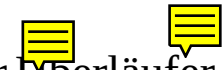
$$|\{h \in H : h(x) = h(y)\}| \le p \cdot (p-1)/m = |H|/m$$

und damit H universell. □

Der gerade bewiesene Satz legt die folgende Strategie zur Wahl einer Hashfunktion nahe. Nehmen wir an, wir wissen, wieviele Schlüssel auf einen gegebenen Bereich von m Adressen abgebildet werden. Dann wählen wir eine Primzahl p, die größer oder gleich der Zahl der Schlüssel ist, und wählen zwei Zahlen a und b zufällig im Bereich $1 \le a < p$ und $0 \le b < p$. Dann ist $h_{a,b}$ eine „gute“ Hashfunktion.

Man beachte, daß wir keinerlei Voraussetzungen über die Größe des Adreßbereichs gemacht haben. Es schadet also nicht, m beispielsweise als Zweierpotenz zu wählen. Klassen universeller Hashfunktionen wurden erstmals von Carter und Wegman in [25] vorgestellt. Eine Verallgemeinerung des Begriffs der universellen Klasse von Hashfunktionen und weitere Eigenschaften solcher Klassen findet man z.B. in [123]. Dort werden auch Verfahren zur Konstruktion perfekter Hashfunktionen diskutiert, die effizienter sind als das von uns zu Anfang dieses Abschnitts angegebene naive Verfahren.

4.2 Hashverfahren mit Verkettung der Überläufer



Soll in eine Hashtabelle t, die bereits den Schlüssel k enthält, ein Synonym k' von k eingefügt werden, so ergibt sich eine Adreßkollision. Der Platz $h(k) = h(k')$ ist bereits besetzt, und k', ein *Überläufer*, muß anderswo gespeichert werden. Eine einfache Art, Überläufer zu speichern, ist die, sie außerhalb der Hashtabelle abzulegen, und zwar in dynamisch veränderbaren Strukturen. So kann man etwa die Überläufer zu jeder Hashadresse in einer linearen Liste verketten; diese Liste wird an den Hashtabelleneintrag angehängt, der sich durch Anwendung der Hashfunktion auf die Schlüssel ergibt.

Beispiel:

Größe der Hashtabelle $m = 7$;
$K = \{0, 1, \ldots, 500\}$;
$h(k) = k \bmod m$;

wir zeigen hier nur die zu den Datensätzen gehörenden Schlüssel (nicht die ganzen Datensätze).

Nach Einfügen der Schlüssel 12, 53, 5, 15, 2, 19, 43 in dieser Reihenfolge in die anfangs leere Hashtabelle ergibt sich die in Abbildung 4.1 gezeigte Situation. Dabei haben wir Überläufer jeweils am Ende der aktuellen Überlaufkette angefügt.

Methode: *Separate Verkettung der Überläufer*
Jedes Element der Hashtabelle t ist Anfangselement einer Überlaufkette (verkettete lineare Liste).

Suchen nach Schlüssel k: Beginne bei $t[h(k)]$ und folge den Verweisen der Überlaufkette, bis entweder k gefunden wurde (erfolgreiche Suche) oder das Ende der Überlaufkette erreicht ist (erfolglose Suche).

Einfügen eines Schlüssels k: Suche nach k; die Suche verläuft erfolglos (sonst wird k nicht eingefügt) und endet am Ende einer Überlaufkette oder bei $t[h(k)]$. Im letzteren Fall trage k in $t[h(k)]$ ein; sonst erzeuge ein neues Listenelement und hänge es ans Ende der Überlaufkette an.

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

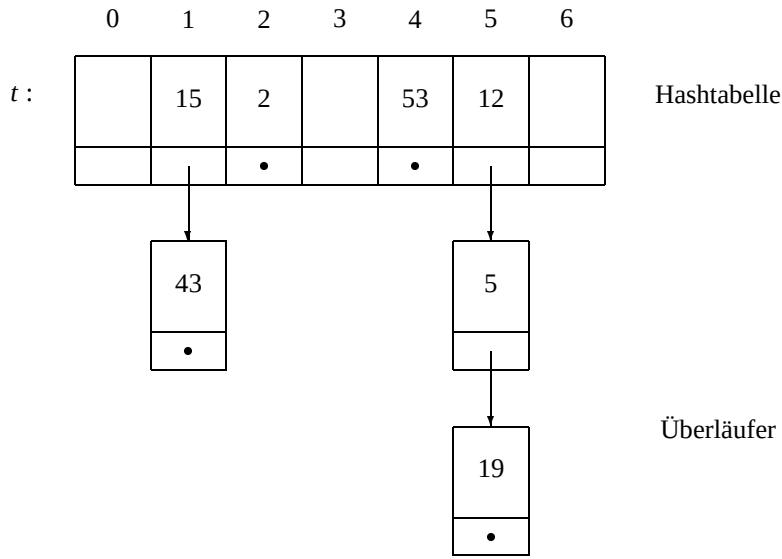


Abbildung 4.1

$t[h(k)]$ und entferne es aus der Überlaufkette. Steht k in einem Element der Überlaufkette, so entferne dieses Element aus der Überlaufkette.

Mit wenigen Modifikationen der zu Beginn dieses Kapitels angegebenen Definition der Hashtabelle läßt sich diese Methode leicht in Pascal-ähnlicher Notation beschreiben. Allerdings fällt auf, daß die unterschiedlichen Fälle (Schlüssel in Hashtabelle oder in Überlaufkette, gegebenenfalls Nachziehen des ersten Überläufers in die Hashtabelle beim Entfernen, usw.) einige Abfragen erfordern, die die Laufzeit der Operationen spürbar beeinträchtigen. Wenn man bereit ist, unter Umständen etwas Speicherplatz zu opfern, so kann man auch einfach *alle* Datensätze in den Überlaufketten speichern; in der Hashtabelle benötigt man dann nur Zeiger auf den Listenanfang.

Das obige Beispiel ist dann wie in Abbildung 4.2 darstellbar. Diese Methode ist als *direkte Verkettung* der Überläufer bekannt. Im Unterschied zur separaten Verkettung der Überläufer wird man hier weniger Speicherplatz benötigen, wenn Datensätze ziemlich groß sind, weil man bei direkter Verkettung bei leeren Hashtabellenplätzen nur wenig Speicherplatz ungenutzt läßt. Sind jedoch die Datensätze klein und über die Hashtabelle gleichmäßig verteilt, etwa ein Datensatz pro Hashadresse, so benötigt man bei direkter Verkettung der Überläufer natürlich mehr Speicherplatz, und zwar für die Anfangszeiger auf Überlaufketten.

Methode: *Direkte Verkettung der Überläufer*
Jedes Element der Hashtaelle ist ein Zeiger auf eine (Überlauf-) Kette.

Suchen nach Schlüssel k: Beginne bei $t[h(k)]\uparrow$ und folge den Verweisen der Überlaufkette, bis entweder k gefunden wurde (erfolgreiche Suche) oder das Ende der Überlaufkette erreicht ist (erfolglose Suche).

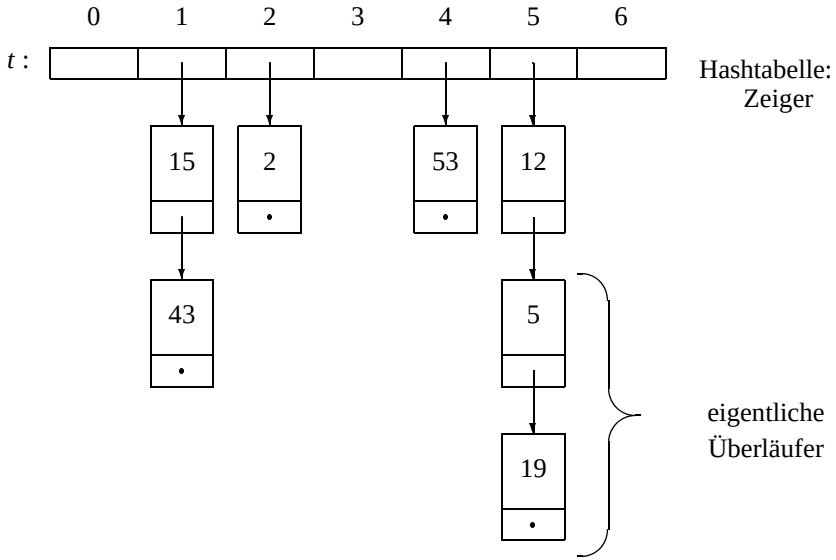


Abbildung 4.2

Einfügen eines Schlüssels *k*: Suche nach *k*; die Suche endet erfolglos am Ende einer Überlaufkette (sonst wird *k* nicht eingefügt). Schaffe ein neues Listenelement und hänge es ans Ende der Überlaufkette an.

Entfernen eines Schlüssels *k*: Suche nach *k*; die Suche verläuft erfolgreich (sonst kann *k* nicht entfernt werden) und endet bei einem Element der Überlaufkette. Entferne dieses Element aus der Überlaufkette.

Im wesentlichen handelt es sich hierbei also stets um Operationen in linearen verketteten Listen. Wir ergänzen die zu Beginn dieses Kapitels angegebenen Definitionen, damit wir die beschriebenen Prozeduren genauer angeben können.

```
type
  zeiger = ↑listenelement;
  listenelement = record
                    k : key;
                    item : itemtype;
                    next : zeiger
                  end;
  hashtabelledirekt = array [hashadresse] of zeiger;
  listenoperation = (suche, einfüge, entferne);

procedure operation (op: listenoperation; var z: zeiger;
          var ds: datensatz);
{führt in Liste ab Zeiger z Operation op mit Datensatz ds aus}
begin
  if z = nil
```

```
        then {Listenende}
          case op of
            suche : write(`Wert tritt nicht auf');
            entferne : write(`Wert nicht in der Liste');
            einfüge : begin
                        new(z);
                        z↑.k := ds.k;
                        z↑.item := ds.item;
                        z↑.next := nil
                      end
          end
        else
          if z↑.k =ds.k
            then
              case op of
                suche : ds.item := z↑.item; {liefere item in ds ab}
                entferne : z := z↑.next; {ändere Zeiger in der Liste}
                einfüge : write(`ist bereits vorhanden')
              end
            else operation(op, z↑.next, ds)
    end {operation}
```

Die Verwendung dieser Prozedur ist klar: Soll beispielsweise nach einem Datensatz mit Schlüssel k gesucht werden, so wird für eine Variable

var *ds* : *datensatz*

nach der Zuweisung

$ds.k := k$ {*Suchschlüssel*}

die Prozedur

operation (*suche*, $t[h(ds.k)]$, *ds*)

aufgerufen. Wenn ein Datensatz mit Schlüssel k gefunden wurde, so enthält $ds.item$ die entsprechende Information.

Analyse: Betrachten wir zunächst die Methode der direkten Verkettung der Überläufer. Wir nehmen an, daß die Hashfunktion alle Hashadressen mit gleicher Wahrscheinlichkeit (Gleichverteilung) 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'.

Bei einer erfolglosen Suche nach k betrachten wir alle Einträge der bei $t[h(k)]$ 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:

$$C'_n = \alpha$$

Ist bei einer erfolgreichen Suche k um i Listenelemente vom Listenanfang $t[h(k)]$ entfernt, so betrachten wir gerade diese i Einträge. Sehen wir uns die Schlüssel einmal in der Reihenfolge an, in der sie eingefügt worden sind. Beim Einfügen des j-ten Schlüssels ist die durchschnittliche Listenlänge gerade $(j-1)/m$. Also betrachten wir bei einer späteren Suche nach dem j-ten Schlüssel gerade $1+(j-1)/m$ Einträge im Durchschnitt, wenn stets am Listenende eingefügt wird und kein Datensatz entfernt wurde. Im Mittel ist die Anzahl der bei der erfolgreichen Suche nach einem Schlüssel betrachteten Einträge also

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

wenn nach jedem Schlüssel mit gleicher Wahrscheinlichkeit gesucht wird. Man beachte, daß diese und die folgenden Analysen das Entfernen von Schlüsseln nicht berücksichtigen.

Die Analyse der Effizienz der erfolglosen und erfolgreichen Suche bei separater Verkettung ist etwas komplizierter; sie kann nachgelesen werden bei [89]. Wir geben hier nur das Resultat wieder:

$$C'_n \approx \alpha + e^{-\alpha}; \qquad C_n \approx 1+\frac{\alpha}{2}$$

Nach den angegebenen Methoden der direkten und separaten 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 i[illegible]e die erfolglose Suche.

Die Effizienz der erfolglosen Suche läßt sich verbessern, wenn man die Überlaufketten sortiert hält. Dann muß man beim erfolglosen Suchen nicht stets bis zum Listenende suchen, sondern kann im Mittel schon in der Mitte der Liste der Überläufer aufhören (man beachte, daß die erfolglose Suche für $\alpha \leq 1$ im Mittel schneller ist als die erfolgreiche). Diese Modifikation lohnt sich besonders bei häufiger erfolgloser Suche, also etwa am Anfang des Aufbaus der Hashtabelle und der Überlaufketten durch fortgesetztes Einfügen. In Fällen mit sehr begrenzter Dynamik, etwa in dem für die Anwendung von Hashverfahren typischen Fall vieler Einfügungen in einer Initialisierungsphase und vieler Suchanfragen danach, kann es attraktiver sein, die Effizienz der erfolgreichen Suche zu steigern, z.B. durch Verwendung selbstanordnender Listen (vgl. Kapitel 3).

Die direkte oder separate Verkettung der Überläufer weist einige wesentliche Vorzüge gegenüber anderen Hashverfahren auf, die wir im folgenden noch erläutern werden (vgl. Abschnitt 4.3). Der Erwartungswert (s.o.) 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. Tabelle 4.1 vermittelt einen Eindruck von der Effizienz der Verkettung der Überläufer; die angegebenen Werte errechnen sich durch Einsetzen von α in die jeweilige Formel.

Anzahl bei der Suche betrachteter Einträge	separate Verkettung		direkte Verkettung	
	erfolgreich	erfolglos	erfolgreich	erfolglos
$\alpha = 0.50$	1.250	1.110	1.250	0.50
0.90	1.450	1.307	1.450	0.90
0.95	1.475	1.337	1.475	0.95
1.00	1.500	1.368	1.500	1.00

Tabelle 4.1

Die entscheidenden Nachteile der Methoden der Verkettung der Überläufer sind der Speicherplatzbedarf für die Zeiger und die Tatsache, daß selbst dann Platz für Überläufer außerhalb der Hashtabelle benötigt wird, wenn in der Hashtabelle noch viele Plätze frei sind. Andere Hashverfahren, die ohne zusätzlichen Speicherplatz auskommen, werden wir im nächsten Abschnitt präsentieren.

4.3 Offene Hashverfahren

Im Unterschied zur Verkettung der Überläufer *außerhalb* der Hashtabelle versucht man bei offenen Hashverfahren, Überläufer *in* der Hashtabelle unterzubringen. Wenn also beim Versuch, den Schlüssel k in die Hashtabelle an Position $h(k)$ einzutragen, festgestellt wird, daß $t[h(k)]$ bereits belegt ist, so muß man nach einer festen Regel einen anderen, nicht belegten Platz (eine *offene* Stelle) finden, an dem man k unterbringen kann. Da man von vornherein nicht wissen kann, welche Plätze belegt sein werden und welche nicht, definiert man für jeden Schlüssel eine Reihenfolge, in der alle Speicherplätze, und zwar einer nach dem anderen, betrachtet werden. Sobald ein betrachteter Platz frei ist, wird der Schlüssel dort gespeichert. Die Folge der zu betrachtenden Speicherplätze für einen Schlüssel nennt man die *Sondierungsfolge* zu diesem Schlüssel. Methoden, die diesem Schema folgen, hat W.W. Peterson 1957 [146] *offene Hashverfahren* genannt. Von den zahlreichen Varianten offener Hashverfahren werden wir nur einige der wichtigsten erläutern; dabei geht es fast immer um die Wahl einer geeigneten Sondierungsfolge.

Natürlich ist das *Entfernen* von Schlüsseln bei all diesen Verfahren problematisch. Ein bereits in der Hashtabelle vorhandener Schlüssel k versperrt ja einem neu einzufügenden Schlüssel k' im allgemeinen einen Platz, den k' gemäß seiner Sondierungsfolge betrachtet. Der neue Schlüssel k' weicht also auf einen anderen Platz (später in der Sondierungsfolge) aus. Wird nun k entfernt, so kann k' nicht wiedergefunden werden, weil der leergewordene Platz von k in der Sondierungsfolge von k' vor dem aktuellen Platz von k' auftritt. In diesem Fall wird k bei den meisten Verfahren dann auch nicht wirklich entfernt, sondern lediglich als entfernt markiert. Wird ein neuer Schlüssel eingefügt, so wird der Platz von k als frei angesehen; wird ein Schlüssel gesucht, so wird der Platz von k als belegt angesehen. Der Effizienz von Hashverfahren ist diese Vorgehensweise nicht besonders zuträglich; für offene Hashverfahren gilt daher in besonderem Maße die Annahme, daß fast nur eingefügt und gesucht und fast nie entfernt wird.

Wir definieren nun ein Schema für offene Hashverfahren, das sich für die meisten (aber nicht alle) der offenen Hashverfahren als Grundlage eignet.

Methode: *Offene Hashverfahren*
Sei $s(j,k)$ eine Funktion von j und k so, daß $(h(k) - s(j,k)) \bmod m$ für $j = 0, 1, \ldots, m-1$, eine Sondierungsfolge bildet, d.h. eine Permutation aller Hashadressen. Es sei stets noch mindestens ein Platz in der Hashtabelle frei.

Suchen nach Schlüssel k: Beginne mit Hashadresse $i = h(k)$. Solange k nicht in $t[i]$ gespeichert ist und $t[i]$ nicht frei ist, suche weiter bei $i = (h(k) - s(j,k)) \bmod m$, 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 $i = (h(k) - s(j,k)) \bmod m$, 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.

Es ist leicht, dieses Schema in ein Programmstück zu übersetzen, wenn wir voraussetzen, daß wir etwa über eine entsprechende Markierung feststellen können, ob ein Platz $t[i]$ frei, belegt oder als entfernt markiert ist. Wir verwenden dazu die Definitionen

```
type
  zustand = (frei, belegt, entfernt);
  markentabelle = array [hashadresse] of zustand;
var
  marke : markentabelle
```

Anfangs sind alle Plätze frei. Der Wert von $marke[i]$ gibt den Zustand des Platzes $t[i]$ an. Wir nehmen an, daß der **mod**-Operator zur Berechnung von Adressen gemäß der Sondierungsfolge wie beschrieben verwendet werden kann, daß also

$(h(ds.k) - s(j, ds.k))$ **mod** m zyklisch bezüglich des Bereichs $0 \,..\, m-1$ von $h(ds.k)$ aus um $s(j, ds.k)$ Positionen links liegt.

```
procedure Suchen (var ds: datensatz; var t: hashtabelle);
{sucht in der Hashtabelle t nach Datensatz mit Schlüssel ds.k und liefert
 ds.item oder eine Meldung über die Erfolglosigkeit der Suche}
var
   i, j : hashadresse;
begin
   j := 0; {Anzahl inspizierter Einträge}
   repeat
      i := (h(ds.k) - s(j, ds.k)) mod m;
      j := j + 1
   until (t[i].k = ds.k) or (marke[i] = frei);
   if marke[i] = belegt
      then ds.item := t[i].item
      else write(`Suche erfolglos beendet')
end
```

```
procedure Einfügen (ds: datensatz; var t: hashtabelle);
{fügt Datensatz ds in Hashtabelle t ein}
var
   i, j : hashadresse;
begin
   j := 0; {Anzahl inspizierter Einträge}
   repeat
      i := (h(ds.k) - s(j, ds.k)) mod m;
      j := j + 1
   until marke[i] <> belegt;
   t[i] := ds;
   marke[i] := belegt
end
```

```
procedure Entfernen (k: key; var t: hashtabelle);
{entfernt Datensatz mit Schlüssel k aus der Hashtabelle t}
var
   i, j : hashadresse;
begin
   j := 0; {Anzahl inspizierter Einträge}
   repeat
      i := (h(k) - s(j, k)) mod m;
      j := j + 1
   until (t[i].k = k) or (marke[i] = frei);
   if marke[i] = belegt
      then marke[i] := entfernt  {entferne}
      else write(`Schlüssel nicht vorhanden')
end
```

Natürlich hätten auch die drei angegebenen Prozeduren wegen ihrer großen Ähnlichkeit zu einer einzigen vereinigt und entsprechend parametrisiert werden können; wir überlassen dies dem interessierten Leser. Um die Grundidee nicht zu verschleiern, haben wir bei den angegebenen Prozeduren keine Vorkehrungen gegen mehrfaches Einfügen des gleichen Schlüssels getroffen und nicht sichergestellt, daß wirklich immer ein freier Platz in der Hashtabelle existiert.

4.3.1 Lineares Sondieren

Beim linearen Sondieren ergibt sich für den Schlüssel k die Sondierungsfolge

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

also die Sondierungsfunktion

$$s(j,k) = j.$$

Beispiel:

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

Dann führen die Schlüssel 12, 53, 5, 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	

Das lineare Sondieren (englisch: linear probing) ist zwar ein sehr einfaches Verfahren, hat aber auch einige Nachteile. Im gezeigten Beispiel etwa ist nach Einfügen von 12, 53, 5 die Wahrscheinlichkeit für einen neu einzufügenden Schlüssel, in der Hashtabelle an einer gewissen Hashadresse gespeichert zu werden, für die verschiedenen Hashadressen drastisch verschieden. Im Eintrag $t[2]$ werden alle Schlüssel k mit $h(k) = 2$ oder $h(k) = 3$ oder $h(k) = 4$ oder $h(k) = 5$ gespeichert, im Eintrag $t[1]$ dagegen nur alle Schlüssel k mit $h(k) = 1$. Bei einer uniformen Hashfunktion, die die Schlüssel mit gleicher Wahrscheinlichkeit auf jede der Hashadressen abbildet, hat in der beschriebenen Situation $t[2]$ die Chance $4/7$, mit dem nächsten Schlüssel belegt zu werden, während diese Chance für $t[1]$ nur $1/7$ beträgt. Lange belegte Teilstücke der Hashtabelle haben also eine stärkere Tendenz zu wachsen als kurze. Dieser Effekt wird noch verstärkt, weil lange belegte Teilstücke zu größeren zusammenwachsen (englisch: to coalesce), wenn die Lücken zwischen ihnen geschlossen werden. Als Folge dieses Phänomens der *primären Häufung* (englisch: *primary clustering*) verschlechtert sich die Effizienz des linearen Sondierens drastisch, sobald sich der Belegungsfaktor α dem Wert 1 nähert.

Analyse: Eine Analyse der Effizienz des linearen Sondierens [89] zeigt, daß für die durchschnittliche Anzahl der bei erfolgloser bzw. erfolgreicher Suche betrachteten Einträge C'_n bzw. C_n gilt:

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

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

Tabelle 4.2 vermittelt durch Einsetzen einiger Werte für α in diese beiden Formeln einen Eindruck von der Effizienz des linearen Sondierens.

Anzahl betrachteter Einträge	lineares Sondieren	
	erfolgreich	erfolglos
$\alpha = 0.50$	1.5	2.5
0.90	5.5	50.5
0.95	10.5	200.5
1.00	—	—

Tabelle 4.2

4.3.2 Quadratisches Sondieren

Um die primäre Häufung des linearen Sondierens zu vermeiden, wird beim quadratischen Sondieren für Schlüssel k um $h(k)$ herum mit quadratisch wachsendem Abstand nach einem freien Platz gesucht. Die Sondierungsfolge für Schlüssel k ist

$$h(k),\ h(k)+1,\ h(k)-1,\ h(k)+4,\ h(k)-4,\ \ldots$$

Die Sondierungsfunktion ist für die Sondierungsfolge $h(k)-s(j,k)$ definiert als

$$s(j,k) = (\lceil j/2 \rceil)^2\,(-1)^j$$

Wenn m eine Primzahl der Form $4i+3$ ist, dann ist garantiert, daß die Sondierungsfolge eine Permutation der Hashadressen 0 bis $m-1$ ist, vgl. [155].

Beispiel: Füge 12, 53, 5, 15, 2, 19 in die anfangs leere Hashtabelle ein. Das ergibt nach Einfügen von 12, 53, 5 (Sondierungsfolge $h(5)$, $h(5)+1$), 15, 2:

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

Nach Einfügen von 19 (Sondierungsfolge $h(19)=5$, $5+1$, $5-1$, $(5+4) \bmod 7 = 2$, $5-4=1$, $(5+9) \bmod 7 = 0$):

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

Zwar ist hier die primäre Häufung vermieden, aber ein anderes Phänomen, die *sekundäre Häufung* (englisch: *secondary clustering*), beeinträchtigt die Effizienz: Zwei Synonyme k und k' durchlaufen stets dieselbe Sondierungsfolge, behindern sich also auf Ausweichplätzen. Das gilt natürlich ebenfalls für das lineare Sondieren. Im angegebenen Beispiel war das der Fall für die Schlüssel 5 und 19. Würde man ein weiteres Synonym von 5 (etwa 26) einfügen, so würden sowohl Schlüssel 5 als auch 19 den neu einzufügenden Schlüssel behindern.

Analyse: Eine Analyse der Effizienz des quadratischen Sondierens zeigt (vgl. [89]), daß für die durchschnittliche Anzahl der bei erfolgreicher bzw. erfolgloser Suche betrachteten Einträge C_n bzw. C'_n gilt:

$$\begin{aligned} C_n &\approx 1+\ln\left(\frac{1}{1-\alpha}\right)-\frac{\alpha}{2} \\ C'_n &\approx \frac{1}{1-\alpha}-\alpha+\ln\left(\frac{1}{1-\alpha}\right) \end{aligned}$$

Tabelle 4.3 vermittelt durch einige in diese Formeln eingesetzte Werte für α einen Eindruck von der Effizienz des quadratischen Sondierens.

Anzahl betrachteter Einträge	quadratisches Sondieren	
	erfolgreich	erfolglos
$\alpha = 0.50$	1.44	2.19
0.90	2.85	11.40
0.95	3.52	22.05
1.00	—	—

Tabelle 4.3

4.3.3 Uniformes und zufälliges Sondieren

Die beim linearen und beim quadratischen Sondieren auftretenden Probleme der primären und sekundären Häufung liegen in der Unabhängigkeit der Sondierungsfunktion von k begründet. Die Sondierungsfolge ist für alle Synonyme die gleiche. Schlüssel werden sich natürlich weniger behindern, wenn die Sondierungsfolge auch für Synonyme variiert. Im Idealfall landen Schlüssel in rein zufällig gewählten Plätzen der Hashtabelle mit gleicher Wahrscheinlichkeit für jeden Platz. Diesen Fall realisiert das *uniforme Sondieren* (englisch: *uniform probing*). Hier ist die Folge $s(j,k)$ für $j = 0,1,\ldots,m-1$ eine Permutation der Hashadressen, die nur von k abhängt, und zwar so, daß jede der $m!$ möglichen Permutationen mit gleicher Wahrscheinlichkeit verwendet wird. Man vermutet [188], daß uniformes Sondieren die Anzahl der Kollisionen beim Einfügen minimiert, also bezüglich des Einfügens optimal ist; die asymptotische Optimalität ist bereits bekannt [198]. Es ist jedoch sehr aufwendig, uniformes Sondieren praktisch zu realisieren; das *zufällige Sondieren* (englisch: *random probing*) bietet sich als Alternative mit fast gleicher Effizienz an. Hierbei wählt man, abhängig von k, eine zufällige Hashadresse für $s(j,k)$. Im Gegensatz zum uniformen Sondieren kann es also vorkommen, daß ein bereits für $s(j,k)$ gewählter Wert für $s(j',k)$, $j' > j$, nochmals gewählt wird, bevor alle Plätze der Hashtabelle betrachtet wurden. Wir werden noch sehen (vgl. Abschnitt 4.3.4), daß auch ein anderes, weniger aufwendig zu realisierendes Verfahren fast die gleiche Effizienz bietet wie das uniforme (und das zufällige) Sondieren.

Analyse: Wir betrachten hier nur die Effizienz des uniformen Sondierens; die des zufälligen Sondierens ist geringfügig schlechter. Jeder Schlüssel wird an einem zufällig gewählten Platz in der Hashtabelle abgespeichert, also ist jede der $\binom{m}{n}$ möglichen Belegungen der m Plätze durch n Schlüssel mit $m-n$ freien Plätzen gleichwahrscheinlich. Die Wahrscheinlichkeit (englisch: probability) p_i, daß genau i Plätze inspiziert werden müssen, um den $(n+1)$-ten Schlüssel einzufügen, ist die Anzahl der Situationen, in denen $i-1$ bestimmte Plätze belegt sind und ein bestimmter Platz frei ist, bezogen auf die Anzahl aller Situationen mit n Schlüsseln, also

$$p_i = \binom{m-i}{n-i+1} \Big/ \binom{m}{n},$$

da außer den $i-1$ fest vorgegebenen noch $n-(i-1)$ Schlüssel auf $m-i$ Plätze verteilt sind. Die durchschnittliche Anzahl betrachteter Plätze beim Einfügen, d.h. bei erfolgloser Suche, ist dann

$$
\begin{aligned}
C'_n &= \sum_{i=1}^{m} i \cdot p_i \\
&\left\{ \text{mit } \sum_{i=1}^{m} p_i = 1 \right\} \\
&= m+1-\sum_{i=1}^{m}(m+1)p_i - \sum_{i=1}^{m}(-i\cdot p_i) \\
&= m+1-\sum_{i=1}^{m}(m+1-i)p_i \\
&\left\{ \text{mit } \binom{a}{b} = \binom{a}{a-b} \right\} \\
&= m+1-\sum_{i=1}^{m}(m+1-i)\binom{m-i}{m-n-1} \Big/ \binom{m}{n} \\
&\left\{ \text{mit } \binom{a}{b} = \frac{b+1}{a+1}\binom{a+1}{b+1} \right\} \\
&= m+1-\sum_{i=1}^{m}(m+1-i)\frac{(m-n)}{(m-i+1)}\binom{m-i+1}{m-n} \Big/ \binom{m}{n} \\
&= m+1-(m-n)\sum_{i=1}^{m}\binom{m-i+1}{m-n} \Big/ \binom{m}{n} \\
&\left\{ \text{mit } \sum_{c=1}^{a}\binom{c}{b} = \binom{a+1}{b+1} \right\} \\
&= m+1-(m-n)\binom{m+1}{m-n+1} \Big/ \binom{m}{n} \\
&= m+1-(m-n)\frac{(m+1)}{(m-n+1)} \\
&= \frac{(m+1)}{(m-n+1)} \\
&\approx \frac{m}{m-n} = \frac{1}{1-\alpha} = 1+\alpha+\alpha^2+\alpha^3+\ldots
\end{aligned}
$$

Die Potenzreihe für C'_n kann intuitiv interpretiert werden: Mit Wahrscheinlichkeit 1 muß mindestens ein Platz inspiziert werden, mit Wahrscheinlichkeit α muß mehr als ein Platz inspiziert werden, mit Wahrscheinlichkeit α^2 müssen mehr als zwei Plätze inspiziert werden, usw.

Die durchschnittliche Anzahl der inspizierten Plätze bei erfolgreicher Suche ist bei offenen Hashverfahren

$$C_n = \frac{1}{n}\sum_{i=0}^{n-1} C'_i,$$

also

$$\begin{aligned} C_n &= \frac{1}{n}\sum_{i=0}^{n-1}\frac{m+1}{m-i+1} \\ &= \frac{m+1}{n}\left(\frac{1}{m+1}+\frac{1}{m}+\cdots\frac{1}{m-n+2}\right) \end{aligned}$$

$$\left\{\text{mit } H_n = \sum_{i=1}^{n}\frac{1}{i},\ \text{die } n\text{-te harmonische Zahl}\right\}$$

$$= \frac{m+1}{n}\left(H_{m+1} - H_{m-n+1}\right)$$

$$\left\{\text{mit } H_n \approx \ln n + \gamma\left(+\frac{1}{2n} - \frac{1}{12n^2}+\cdots\right),\ \text{wobei } \gamma = 0.57\ldots\right\}$$

$$\begin{aligned} &\approx \frac{1}{\alpha}\left(\ln(m+1)+\gamma-\ln(m-n+1)-\gamma\right) \\ &= \frac{1}{\alpha}\ln\left(\frac{m+1}{m-n+1}\right) \\ &\approx \frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right) \end{aligned}$$

Zusammenfassend gilt also für uniformes und näherungsweise auch für zufälliges Sondieren:

$$\begin{aligned} C'_n &\approx \frac{1}{1-\alpha} \\ C_n &\approx \frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right) \end{aligned}$$

Tabelle 4.4 vermittelt durch einige in diese Formeln eingesetzten Werte für α einen Eindruck von der Effizienz des uniformen Sondierens.

Anzahl betrachteter Einträge	uniformes Sondieren	
	erfolgreich	erfolglos
$\alpha = 0.50$	1.39	2
0.90	2.56	10
0.95	3.15	20
1.00	—	—

Tabelle 4.4

4.3.4 Double Hashing

Die Effizienz des uniformen Sondierens wird bereits annähernd erreicht, wenn man statt einer zufälligen Permutation für die Sondierungsfolge eine zweite Hashfunktion verwendet; die gewählte Sondierungsfolge für Schlüssel k ist

$$h(k),\ h(k)-h'(k),\ h(k)-2\cdot h'(k),\ \ldots\ ,\ h(k)-(m-1)h'(k),$$

jeweils modulo m, wenn $h'(k)$ die zweite Hashfunktion bezeichnet. Für die Sondierungsfunktion ergibt sich

$$s(j,k) = j \cdot h'(k).$$

Dabei muß $h'(k)$ so gewählt werden, daß für alle Schlüssel k die Sondierungsfolge eine Permutation der Hashadressen bildet. Das bedeutet, daß $h'(k) \neq 0$ sein muß und m nicht teilen darf. Wählen wir m als Primzahl, dann gilt dies sicher für jedes $h'(k)$ und für alle k; diese Wahl von m ist ja auch günstig für die Divisions-Rest-Methode bei der Hashfunktion h. Wählt man $h'(k)$ abhängig von $h(k)$, so werden manche (oder gar alle) Synonyme die gleiche Sondierungsfolge haben; eine gewisse sekundäre Häufung ist die Folge. Das kann man vermeiden, wenn man $h'(k)$ von $h(k)$ unabhängig wählt, wenn also für zwei verschiedene Schlüssel k und k' gilt:

$$p\left[h(k)=h(k') \text{ und } h'(k)=h'(k')\right] = p\left[h(k)=h(k')\right] \cdot p\left[h'(k)=h'(k')\right],$$

wobei $p[Bedingung]$ die Wahrscheinlichkeit dafür ist, daß die angegebene Bedingung gilt. Anders ausgedrückt heißt das: Sind zwei Schlüssel Synonyme bezüglich h, so sind sie mit Wahrscheinlichkeit $1/m'$ Synonyme bezüglich h' (m' ist die Anzahl der Werte, die die Funktion h' annehmen kann); also sind zwei Schlüssel mit Wahrscheinlichkeit $1/(m\cdot m')$ Synonyme bezüglich h und h' gleichzeitig.

Ist m eine Primzahl und $h(k) = k \bmod m$, so erfüllt $h'(k) = 1 + k \bmod (m-2)$ die obigen Anforderungen (das ist besser als $1 + k \bmod (m-1)$, weil $m-1$ gerade ist).

Beispiel: Wir betrachten wieder den Fall, daß die Schlüssel 12, 53, 5, 15, 2, 19 in die anfangs leere Hashtabelle der Größe 7 eingefügt werden sollen. Wir wählen also $h(k) = k \bmod 7$ und $h'(k) = 1 + k \bmod 5$. Die Sondierungsfolge für k ist $h(k)$, $h(k)-h'(k)$, $h(k)-2h'(k)$, jeweils *modulo m*. Das ergibt nach Einfügen von 12, 53:

	0	1	2	3	4	5	6
t:					53	12	

Nach Einfügen von 5 (Sondierungsfolge ist $h(5) = 5 \bmod 7 = 5, 5-(1+5 \bmod 5) = 4$, $5-2=3$), 15, 2:

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

Nach Einfügen von 19 (Sondierungsfolge ist $h(19) = 19 \bmod 7 = 5$, $5-(1+19 \bmod 5) = 0$):

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

Beim Einfügen des Schlüssels 19 müssen hier also lediglich zwei Plätze (nämlich $t[5]$ und $t[0]$) inspiziert werden, während es beim linearen und beim quadratischen Sondieren jeweils sechs Plätze waren.

Double Hashing ist genauso effizient wie uniformes Sondieren; der theoretische Unterschied ist minimal, wenn $h'(k)$ unabhängig von $h(k)$ gewählt wird. Da Double Hashing ein leicht implementierbares Verfahren ist, bietet es sich als praktisch einsetzbares offenes Hashverfahren an. Entsprechend ist es die Grundlage für zwei Methoden, bei denen versucht wird, auf Kosten der Einfügezeit die Effizienz der erfolgreichen Suche zu verbessern.

Verbesserung der erfolgreichen Suche

Methoden zur Verbesserung der erfolgreichen Suche basieren auf der Erkenntnis, daß die durchschnittliche Suchzeit für erfolgreiche Suche bei Hashverfahren ohne Häufung mit unterschiedlicher Reihenfolge des Einfügens der Schlüssel variiert. So ist etwa die durchschnittliche Suchzeit im gerade betrachteten Beispiel für die erfolgreiche Suche (Suchzeit(12) + Suchzeit(53) + Suchzeit(5) + Suchzeit(15) + Suchzeit(2) + Suchzeit(19))/6 $= (1+1+3+1+1+2)/6 = 1.5$. Fügt man die Schlüssel jedoch in der Reihenfolge 53, 5, 15, 2, 19, 12 ein, so ergibt sich die Situation

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

und damit eine durchschnittliche erfolgreiche Suchzeit von $8/6 = 1.33\ldots$. Für die Einfügereihenfolge 12, 5, 19, 53, 2, 15 ergibt sich die durchschnittliche erfolgreiche Suchzeit zu $10/6 = 1.66\ldots$.

In Fällen, in denen wesentlich häufiger erfolgreich gesucht wird als eingefügt, kann es daher lohnend sein, die Schlüssel beim Einfügen eines neuen Schlüssels so zu reorganisieren, daß die Suchzeit verkürzt wird. So berichten etwa Bell und Kaman [13], daß ein COBOL-Compiler beim Übersetzen 735 Einträge in eine Symboltabelle vorgenommen und diese Tabelle 10988 Mal angesprochen hat; das sind etwa 14 Suchoperationen pro Einfügung.

Brents Algorithmus

Betrachten wir in unserem Beispiel des Einfügens der Schlüssel 12, 53, 5, 15, 2, 19 die Operation des Einfügens von Schlüssel 5. In der Situation

t:	0	1	2	3	4	5	6
					53	12	

wird Schlüssel 5 nach Inspektion der Plätze 5, 4, 3 bei $t[3]$ eingetragen:

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

Die durchschnittliche Suchzeit ist damit $(1+1+3)/3 = 5/3 = 1.66\ldots$. Die Adreßkollision von Schlüssel 5 mit Schlüssel 12 in $t[5]$ hätte man aber auch anders lösen können. Statt Schlüssel 12 in $t[5]$ zu belassen, hätte man Schlüssel 5 in $t[5]$ eintragen können und Schlüssel 12 weiter sondieren lassen können. Die Sondierungsfolge für Schlüssel 12 wäre dann $12 \bmod 7 = 5$, $5-(1+12 \bmod 5) = 2$ und die Situation damit

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

mit einer durchschnittlichen Suchzeit für die erfolgreiche Suche von $(1+1+2)/3 = 4/3 = 1.33\ldots$. Die entstandene Situation entspricht gerade derjenigen, die sich beim Einfügen von 5, 53, 12 ergibt. Es ist also eine für die erfolgreiche Suche günstige Einfügereihenfolge simuliert worden.

Methode: *Brents Algorithmus*

Einfügen eines Schlüssels k: Beginne mit Hashadresse $i = h(k)$. Solange $t[i]$ belegt ist, betrachte die beiden Hashadressen $b = (i - h'(k)) \bmod m$ und $b' = (i - h'(k')) \bmod m$ mit $k' = t[i].k$: Ist $t[b]$ frei oder $t[b']$ belegt, fahre fort mit $i = b$; andernfalls trage k an Hashadresse i ein und fahre fort mit $k = k'$ und $i = b'$. Jetzt ist $t[i]$ frei; trage k bei $t[i]$ ein.

```
procedure BrentEinfügen (ds: datensatz; var t: hashtabelle);
{fügt Datensatz ds in Hashtabelle t ein}
var
  i, b, bb : hashadresse;
begin
  i := h(ds.k);
  while marke[i] = belegt do
    begin
      b := (i − h'(ds.k)) mod m;
      bb := (i − h'(t[i].k)) mod m;
      if (marke[b] = frei) or (marke[bb] = belegt)
        then i := b
        else begin
               vertausche (ds, t[i]);
               i := bb;
             end;
    end;
  t[i] := ds;
  marke[i] := belegt
end
```

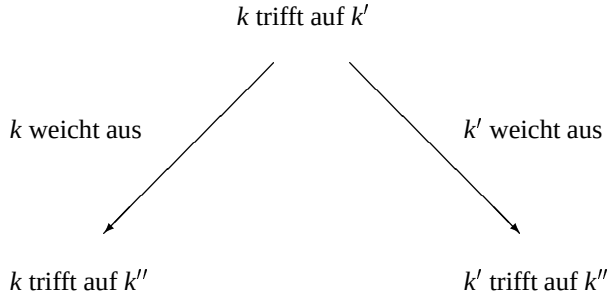
Brents Analyse [23] zeigt, daß die Zeit für erfolglose Suche unverändert bleibt, aber die Zeit für erfolgreiche Suche auch bei voller Hashtabelle stets unter durchschnittlich 2.5 inspizierten Einträgen liegt:

$$\begin{aligned} & \approx \frac{1}{1-\alpha} \\ C_n & \approx 1+\frac{\alpha}{2}+\frac{\alpha^3}{4}+\frac{\alpha^4}{15}-\frac{\alpha^5}{18}+\cdots < 2.5 \end{aligned}$$

Binärbaum-Sondieren

Das Binärbaum-Sondieren (englisch: binary tree hashing) kann als eine Fortführung von Brents Idee angesehen werden ([69], [116]). Wenn Schlüssel k an Hashadresse $h(k)$ nicht in die Hashtabelle eingetragen werden kann, weil sich dort schon ein Schlüssel k' befindet, so gibt es zwei Möglichkeiten. Entweder bleibt k' an seinem Platz, und für k wird gemäß der Sondierungsfolge für k ein anderer Platz gesucht, oder k' wird in der Hashtabelle durch k ersetzt, und für k' wird gemäß der Sondierungsfolge für k' ein anderer Platz gesucht.

Wenn sich auf eine der beiden Alternativen sogleich ein leerer Platz findet, so wird der entsprechende Schlüssel dort eingetragen und das Einfügen ist beendet. Andernfalls werden beide Alternativen analog weiterverfolgt. Hier liegt der Unterschied zu Brents Algorithmus, bei dem nur die erste Alternative weiterverfolgt wird. Nach jeder Inspektion eines belegten Platzes ergibt sich eine weitere mögliche Sondierungsfolge, nämlich die zu dem in der Hashtabelle gespeicherten Schlüssel. Insgesamt hat die Abfolge der Inspektionen einzelner Plätze die Gestalt eines Binärbaumes:



Dieser Binärbaum wird niveauweise inspiziert; sobald ein freier Platz angetroffen wird, wird der ausweichende Schlüssel dort eingetragen. Die Analyse von Gonnet, Munro [69] ergibt für die erfolgreiche Suche:

$$C_n \approx 1 + \frac{\alpha}{2} + \frac{\alpha^3}{4} + \frac{\alpha^4}{15} + \cdots < 2.2$$

Allerdings sind die Kosten für das Einfügen, insbesondere bei fast voller Hashtabelle, relativ hoch; in der beispielhaft betrachteten Anwendung des COBOL-Compilers spielt das aber kaum eine Rolle.

4.3.5 Ordered Hashing

Erinnern wir uns an das Beschleunigen der erfolglosen Suche beim Hashing mit Verkettung der Überläufer. Dort haben wir die Überlaufketten sortiert mit Schlüsseln belegt. Statt beim Einfügen einen Schlüssel hinten an die Überlaufkette anzuhängen, haben wir den Schlüssel in der Überlaufkette an der Position, die sich durch die Sortierung ergab, eingefügt. Das kann man, ebenso wie die Beschleunigung der erfolgreichen Suche beim Double Hashing, als die Simulation einer günstigeren Einfügereihenfolge der Schlüssel mit unsortierten Überlaufketten ansehen. Dieses Prinzip wollen wir nun auch auf offene Hashverfahren anwenden. Die Synonyme des gesuchten Schlüssels k sollen also in der Hashtabelle so abgespeichert sein, daß wir sie gemäß der Sondierungsfolge für k in sortierter Reihenfolge antreffen; wir wollen so den Fall simulieren, daß diese Schlüssel in sortierter Reihenfolge eingefügt worden sind. Wir legen uns hier (willkürlich) auf eine aufsteigende Sortierung fest.

Betrachten wir ein weiteres Mal das bekannte Beispiel. Haben wir die Schlüssel 12, 53, 5, 15, 2, 19 in die anfangs leere Hashtabelle nach dem Prinzip des Double Hashing eingefügt, so ergibt sich folgende Situation:

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

Die Synonyme 12, 5, 19 sind in dieser Reihenfolge eingefügt worden; entsprechend steht an der Hashadresse $h(12) = h(5) = h(19) = 5$ der Schlüssel 12. Bei einer Suche nach Schlüssel 5 finden wir also zunächst in $h(5)$ die 12. Die Suche muß also fortgesetzt werden. Nehmen wir nun an, die Schlüssel seien in aufsteigend sortierter Reihenfolge eingefügt worden; dann könnten wir aus $h(5) = 12 > 5$ bereits schließen, daß Schlüssel 5 nicht in der Hashtabelle vorkommt; die erfolglose Suche könnte also früher abgebrochen werden.

Soweit gleicht das *Ordered Hashing* dem Sortieren der Überlaufketten beim Verketten der Überläufer. Es gibt aber zwei wesentliche Unterschiede, die durch die Unterlegung eines offenen Hashverfahrens bedingt sind. Erstens kann ein neu einzufügender Schlüssel nicht einfach in die Sondierungskette eingefügt werden; Schlüssel, die in der Sondierungskette folgen, müssen eventuell in dieser Kette nach hinten rücken (vgl. Einfügen in ein Array, Kapitel 1). Zweitens kann ein Schlüssel, der in der Sondierungskette nach hinten rückt, auf einen bereits belegten Platz treffen; das kann vielen Schlüsseln in der Sondierungskette passieren. Der Platz, auf den ein solcher Schlüssel trifft, ist im allgemeinen nicht mit einem Synonym belegt.

Der erste Unterschied ist nicht besonders problematisch, denn beim herkömmlichen Einfügen eines neuen Schlüssels werden ja ohnedies alle besetzten Plätze in der Sondierungsfolge inspiziert. Der zweite Unterschied ist schon eher beunruhigend, denn er bedeutet doch, daß die Verschiebung in einer Sondierungskette auf andere übergreifen kann.

Fügen wir als Beispiel die Schlüssel 2, 12, 15, 53, 5 in die anfangs leere Hashtabelle mit Double Hashing ein (vgl. Abschnitt 4.3.4). Nach Einfügen von 2, 12, 15, 53 ergibt sich:

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

Nun soll Schlüssel 5 eingefügt werden, nach dem Prinzip des Ordered Hashing. Da in $h(5) = 5$ ein größerer Schlüssel, nämlich 12, gespeichert ist, wird 12 von 5 verdrängt, und 5 wird in $t[5]$ gespeichert. Für Schlüssel 12 muß nun gemäß seiner Sondierungsfolge ein neuer Platz gesucht werden; in unserem Beispiel ist das wegen $h'(12) = 3$ der Platz $t[2]$. Platz $t[2]$ ist aber bereits belegt mit Schlüssel 2, und 2 ist kein Synonym von Schlüssel 12. Nun gibt es zwei Möglichkeiten: Entweder Schlüssel 2 bleibt in $t[2]$, und Schlüssel 12 sucht weiter, oder Schlüssel 12 wird in $t[2]$ gespeichert, und Schlüssel 2 sucht einen anderen Platz. Die zweite Möglichkeit scheidet aus, denn, würden wir Schlüssel 12 in $t[2]$ speichern und dann nach Schlüssel 2 suchen, so würde die Suche bereits bei $t[2]$ erfolglos abgebrochen werden müssen, obwohl Schlüssel 2 in der Hashtabelle gespeichert ist. Wir folgen also auch hier der Regel, daß der kleinere zweier konkurrierender Schlüssel den umkämpften Platz besetzt; der größere sucht weiter. Der nächste für Schlüssel 12 zu inspizierende Platz ist somit $t[6]$; er ist frei und Schlüssel 12 wird eingefügt:

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

Nach Einfügen der Schlüssel 19 und 43 ergibt sich schließlich mit $h(19) = 5$, $h'(19) = 5$, $h(43) = 1$, $h'(43) = 4$ und $h'(53) = 4$:

	0	1	2	3	4	5	6
t :	19	15	2	53	43	5	12

Methode: *Ordered Hashing für offene Hashverfahren*
Sei $s(j,k)$ eine Funktion von j und k so, daß $(h(k) - s(j,k)) \bmod m$ für $j = 0, 1, \ldots, m-1$ eine Sondierungsfolge, d.h. eine Permutation aller Hashadressen bildet. Es sei stets noch mindestens ein Platz in der Hashtabelle frei (das ist im obigen Beispiel nicht der Fall, macht aber die algorithmische Beschreibung einfacher).

Suchen nach Schlüssel k: Beginne mit Hashadresse $i = h(k)$. Solange k nicht in $t[i]$ gespeichert ist, $t[i]$ nicht frei und $t[i].k < k$ ist, suche weiter bei $i = (h(k) - s(j,k)) \bmod m$, für aufsteigende Werte von j. Falls $t[i]$ belegt und $t[i].k = k$ ist, so wurde k gefunden; sonst war die Suche erfolglos.

Einfügen eines Schlüssel k: Wir nehmen an, daß k nicht schon in t vorkommt. Beginne mit Hashadresse $i = h(k)$. Solange $t[i]$ belegt ist, vertausche $t[i].k$ mit k, falls $k < t[i].k$, und mache weiter beim nächsten zu k gehörigen i. Trage k bei $t[i]$ ein.

Entfernen eines Schlüssels k: wie bisher.

```
    procedure orderedSuchen (var ds: datensatz; var t: hashtabelle);
    {sucht in der Hashtabelle t gemäß Ordered Hashing nach dem Daten-
     satz mit Schlüssel ds.k und liefert ds.item oder eine Meldung über die
     Erfolglosigkeit der Suche}
    var
      i, j : hashadresse;
    begin
      j := 0; {Anzahl inspizierter Einträge}
      repeat
        i := (h(ds.k) - s(j,k)) mod m;
        j := j + 1
      until (t[i].k = ds.k) or (marke[i] = frei)
{*}           or ((marke[i] = belegt) and (t[i].k > ds.k));
{*}   if (marke[i] = belegt) and (t[i].k = ds.k)
        then ds.item := t[i].item  {Suche erfolgreich}
        else write('Suche erfolglos beendet')
    end


    procedure orderedEinfügen (ds : datensatz; var t: hashtabelle);
    {fügt Datensatz ds gemäß Ordered Hashing in Hashtabelle t ein}
    var
      i : hashadresse;
    begin
{*}     i := h(ds.k);
        while marke[i] <> frei do
```

```
            begin
{∗}           if ds.k < t[i].k
{∗}              then vertausche(ds, t[i]);
{∗}           i := (i − s(1, ds.k)) mod m
            end;
        t[i] := ds;
        marke[i] := belegt
    end
```

Zu Beginn des Abschnitts 4.3 haben wir die entsprechende Beschreibung der Methoden und Prozeduren für offene Hashverfahren ohne Ordnung angegeben; die dort gemachten Bemerkungen gelten hier ebenfalls. In den Prozeduren *orderedSuchen* und *orderedEinfügen* haben wir die Programmzeilen mit einem $\{*\}$ kenntlich gemacht, die neu hinzugekommen oder geändert worden sind. Überdies haben wir beim *orderedEinfügen* die Struktur der Schleife ein wenig geändert. Bemerkenswert ist, daß wir beim Einfügen die neue Hashadresse nicht in der allgemeinen Form $i = (h(ds.k) - s(j, ds.k)) \bmod m$ berechnen können, weil ja für einen Schlüssel einer anderen Sondierungsfolge (kein Synonym für k) dessen Position j in dessen Sondierungsfolge unbekannt ist. Soll in unserem Beispiel in der Situation

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

mit Double Hashing der Schlüssel 5 eingefügt werden, so wird Schlüssel 12 auf Platz $t[2]$ verdrängt, wo sich bereits ein Schlüssel befindet. Sofern dieser Schlüssel von Schlüssel 12 verdrängt wird (etwa wenn statt 2 dort 72 stünde), muß die neue Hashadresse für den verdrängten Schlüssel berechnet werden können ohne Kenntnis darüber, wie oft dieser Schlüssel bereits ausgewichen ist. Das geht bei den bisher von uns betrachteten Verfahren nur in den Fällen, in denen für jeden Schlüssel in der Sondierungsfolge aufeinanderfolgende Plätze um einen festen Betrag versetzt sind. Das heißt, daß das Ordered Hashing nicht auf das quadratische Sondieren, das zufällige Sondieren und das pseudozufällige Sondieren anwendbar ist; es ist anwendbar für lineares Sondieren und Double Hashing. In beiden Fällen läßt sich die Sondierungsfolge statt durch $(h(k) - s(j,k)) \bmod m$ für $j = 0, \ldots, m-1$ auch durch $i = h(k)$ im ersten Schritt und $i = i - s(1,k)$ danach berechnen, weil $s(j,k) - s(j-1,k) = s(1,k)$ für alle j, $1 \le j \le m-1$ gilt.

Überlegen wir uns nun, wann denn das Prinzip des Ordered Hashing korrekt ist. Sei $p_0(k)$, $p_1(k)$, …, $p_{m-1}(k)$ die Sondierungsfolge für Schlüssel k, also $p_j(k) = (h(k) - s(j,k)) \bmod m$. Der Suchalgorithmus liefert stets das richtige Resultat, wenn für jeden Schlüssel k auf Platz $p_j(k)$ gilt, daß alle Schlüssel auf Plätzen $p_i(k)$, $i < j$, kleiner sind als k. Dann nämlich werden beim Suchen nach k die Plätze $p_i(k)$, $i < j$, und schließlich $p_j(k)$ inspiziert. Man beachte, daß diese Forderung natürlich nichts über die Schlüssel auf Plätzen $p_{i'}$, $i' > j$, impliziert. Anfangs, also bei leerer Hashtabelle, ist die Forderung für alle gespeicherten Schlüssel trivialerweise erfüllt. Wir überlegen uns nun, daß diese Forderung nach dem Einfügen eines Schlüssels erfüllt bleibt, wenn sie davor erfüllt war. Das ist aber offensichtlich, denn wenn beim Einfügen an einer Stelle p ein Schlüssel

eingetragen wird, dann war entweder $t[p]$ frei, oder in $t[p]$ war ein größerer Schlüssel gespeichert. Ein Schlüsselwert auf einem Platz $p_i(k)$, wobei k auf Platz $p_j(k)$, $j > i$, gespeichert ist, kann also nur verringert werden.

Jetzt ist auch klar, daß als *entfernt* markierte Plätze nicht einfach wieder belegt werden können. Ein solcher Platz kann aber natürlich mit einem kleineren Schlüssel wieder belegt werden. In der Prozedur *orderedEinfügen* könnte man also die Zeilen

```
{*}  if ds.k < t[i].k
{*}     then vertausche(ds, t[i]);
```

ersetzen durch

```
{*}  if ds.k < t[i].k
{*}     then
{*}        if marke[i] = entfernt
{*}           then exit while-loop
{*}           else vertausche(ds, t[i]);
```

und damit manche der als *entfernt* markierten Plätze wiederverwenden.

Amble und Knuth [8] haben gezeigt, daß eine Menge von Schlüsseln unabhängig von der Reihenfolge ihres Einfügens mit Ordered Hashing immer gleich auf die Plätze einer Hashtabelle verteilt wird; also ergibt sich stets dieselbe Situation, als hätte man die Schlüssel sortiert eingefügt. Um dies einzusehen, nehmen wir an, daß es zwei verschiedene Situationen (Belegungen der Hashtabelle) für dieselbe Schlüsselmenge gibt; mindestens ein Schlüssel befindet sich demnach in beiden Situationen nicht am gleichen Platz. Betrachten wir jetzt den kleinsten Schlüssel k, der sich in beiden Situationen nicht am selben Platz befindet. Einmal landet er am Platz $p_i(k)$, das andere Mal am Platz $p_j(k), i \neq j$. Sei nun $i < j$ (sonst vertausche i mit j). Befindet sich k am Platz $p_j(k)$, so befindet sich ein kleinerer Schlüssel $k' < k$ am Platz $p_i(k)$; in der anderen Situation befindet sich k' jedoch nicht am Platz $p_i(k)$, denn dort befindet sich ja k. Also befindet sich k' in beiden Situationen an verschiedenen Plätzen, und $k' < k$; ein Widerspruch zur Annahme. Damit ist klar, daß es nur eine Anordnung einer Menge von Schlüsseln mit Ordered Hashing in einer Hashtabelle gibt.

Analyse: Die Effizienz der erfolgreichen Suche ändert sich durch Anwendung des Prinzips des Ordered Hashing im Durchschnitt nicht (gegenüber dem zugrundeliegenden Verfahren), wohl aber die der erfolglosen Suche. Bei Ordered Hashing ist eine erfolglose Suche genauso teuer wie die erfolgreiche Suche wäre, wenn sich der gesuchte Schlüssel außer den tatsächlich eingetragenen Schlüsseln in der Hashtabelle befände:

$$orderedC'_n = C_{n+1} \approx C_n$$
$$orderedC_n = C_n$$

Die Anzahl der beim Einfügen inspizierten Einträge ist nur geringfügig höher als beim zugrundeliegenden Verfahren. Mit Ordered Hashing ist es also gelungen, die Kosten für die erfolglose Suche auf die Kosten für die erfolgreiche Suche zu reduzieren, um den Preis etwas erhöhter Einfügekosten.

4.3.6 Robin-Hood-Hashing

Wir haben gesehen, wie man die Effizienz von Double Hashing für die erfolgreiche Suche durch Brents Algorithmus oder durch Binärbaum-Sondieren und für die erfolglose Suche durch Ordered Hashing verbessern kann. Dies gelang durch geeignetes Umordnen von Schlüsseln anläßlich einer Einfügeoperation. Bei Brents Variation des Double Hashing dient das Umordnen von Schlüsseln dazu, die durchschnittliche Effizienz der erfolgreichen Suche zu verbessern, also den Erwartungswert der Länge von Sondierungsfolgen zu verringern; Binärbaum-Sondieren ist eine natürliche Verallgemeinerung mit demselben Ziel. *Robin-Hood-Hashing* ([26], [27]) ordnet ebenfalls Schlüssel beim Einfügen um, aber mit dem Ziel der Verringerung der Länge der *längsten* Sondierungsfolge.

Methode: *Robin-Hood-Hashing*
Einfügen eines Schlüssels k: Beginne mit Hashadresse $i = h(k)$. Solange $t[i]$ belegt ist, vergleiche die relative Position j der Adresse i in der Sondierungsfolge von k mit der relativen Position j' der Adresse i in der Sondierungsfolge von $k' = t[i].k$: Ist $j' \geq j$, so fahre fort mit $i = (i - h'(k)) \bmod m$, sonst trage k bei $t[i]$ ein und fahre fort mit $k = k'$ und $i = (i - h'(k')) \bmod m$. Jetzt ist $t[i]$ frei; trage k bei $t[i]$ ein.

Robin-Hood-Hashing ändert also nichts an der durchschnittlichen Länge von Sondierungsfolgen, sondern gleicht nur die Längen der verschiedenen Sondierungsfolgen einander an — wie Robin Hood den Bestand an Gütern nicht geändert hat, sondern nur deren Verteilung. Erstaunlicherweise sinkt mit Robin-Hood-Hashing die Varianz der Länge von Sondierungsfolgen von einem Wert von fast $2m$ für Double Hashing auf einen Wert von weniger als 2, also eine sehr kleine Konstante. Die Varianz bleibt sogar konstant, wenn die Hashtabelle voll ist. Der Erwartungswert für die Länge der längsten Sondierungsfolge ist bei n gespeicherten Schlüsseln höchstens um $\lceil \log_2 n \rceil$ höher als der Erwartungswert aller Längen. Für eine volle Hashtabelle ergibt sich als Erwartungswert für die Länge der längsten Sondierungsfolge $\Theta(\ln m)$. Diesen Wert kann man nur um einen konstanten Faktor verbessern, wenn man die Schlüssel in der Hashtabelle so unterbringt, daß die Länge der längsten Sondierungsfolge minimiert wird [69, 148, 161]; um dies tun zu können, muß man aber das entsprechende Zuordnungsproblem [92] lösen, das selbst $O(n^2 \log n)$ Zeit [60] kosten kann.

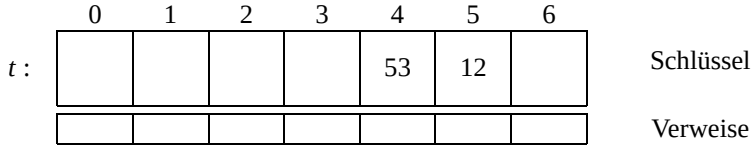
Kennt man zu einer Hashtabelle die Länge l der längsten auftretenden Sondierungsfolge, so kann man dies für eine Beschleunigung der erfolglosen Suche ausnutzen [114]: Jede Suche, auch eine erfolglose, wird nach dem Betrachten von l Hashtabelleneinträgen abgebrochen. Diese Länge l kann man bei Robin-Hood-Hashing (ohne Entferne-Operationen) ohne Zusatzaufwand mitführen, weil man beim Einfügen eines (verdrängten) Schlüssels dessen relative Position in seiner Sondierungsfolge ohnehin kennen muß. Trifft man beim Einfügen eines Schlüssels k auf einen mit k' belegten

Platz, so berechnet man die aktuelle Position von k' in seiner Sondierungsfolge durch eine Suche nach k'; die entsprechende Information für k kennt man bereits. Bei dieser Realisierung ist das Einfügen eines Schlüssels bei Robin-Hood-Hashing ineffizienter als etwa bei Double Hashing, weil ja die durchschnittliche Länge von Sondierungsfolgen nicht verkürzt worden ist und beim Einfügen für jeden betrachteten, belegten Platz eine erfolgreiche Suche durchgeführt werden muß. Mit einem schlauen Algorithmus für die Suche (*smart searching* [27]) läßt sich sowohl die Effizienz der Suche als auch die des Einfügens deutlich verbessern. Dabei benutzen wir die Kenntnis des auf die nächstgelegene ganze Zahl gerundeten Erwartungswerts s der Länge von Sondierungsfolgen und beginnen bei der Suche nach einem Schlüssel k nicht an Position 1 seiner Sondierungsfolge, sondern an Position s. Die zu s gehörende Adresse für Schlüssel k kann leicht berechnet werden; sie ist bei Double Hashing $h(k) - (s-1)h'(k)$. Finden wir Schlüssel k nicht an dem zu s gehörenden Platz, so sondieren wir der Reihe nach die Plätze zu $s+1, s-1, s+2, s-2, \ldots$ nach unten bis 1 und nach oben bis l, falls s durch Abrunden entstand, und sonst $s+1, s-1, s+2, \ldots$ Wenn k dabei nicht gefunden wird, endet die Suche erfolglos. Die Effizienz der erfolglosen Suche verbessert sich bei diesem Verfahren natürlich nicht, aber der Erwartungswert für die erfolgreiche Suche ist eine Konstante. Selbst bei einer vollen Hashtabelle werden stets weniger als 2.8 Einträge inspiziert. Die höchste Effizienz bei der Suche erzielt man, wenn man Hashtabelleneinträge in genau derjenigen Reihenfolge betrachtet, die sich durch die Anordnung aller in der Sondierungsfolge zum gesuchten Schlüssel vorkommenden Plätze nach absteigenden Erfolgswahrscheinlichkeiten ergibt. In diesem Fall inspiziert man bei einer Suche stets weniger als 2.6 Einträge. Damit ist nicht nur die Effizienz der Suche, sondern auch die Effizienz des Einfügens fast dieselbe wie bei Double Hashing (bis auf einen kleinen konstanten Faktor). Außerdem kann man mit Robin-Hood-Hashing auch eine Folge von Entferne- und Einfügeoperationen durchführen, ohne daß die Suchzeit degeneriert. Experimente hierzu und zum Vergleich von Robin-Hood-Hashing mit anderen offenen Hashverfahren sind in [26] ausführlich beschrieben.

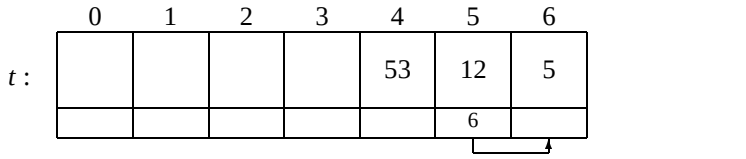
4.3.7 Coalesced Hashing

Vergleichen wir rückblickend die Effizienz aller bisher betrachteten Verfahren, so zeigt sich, daß sowohl die erfolglose als auch die erfolgreiche Suche bei Verkettung der Überläufer am schnellsten ist. Das ist auch intuitiv plausibel, denn bei offenen Hashverfahren war es ja stets möglich, daß wir beim Inspizieren der Plätze gemäß der Sondierungsfolge für einen Schlüssel k andere Schlüssel k' angetroffen haben, die keine Synonyme von k waren. Andererseits haben die Verfahren der Verkettung der Überläufer (vgl. Abschnitt 4.2) den Nachteil, daß selbst dann neuer Speicherplatz außerhalb der Hashtabelle dynamisch bereitgestellt und belegt werden muß, wenn in der Hashtabelle noch Plätze frei sind. Das Verfahren des *Coalesced Hashing* (englisch: to coalesce = verschmelzen) verbindet das Prinzip des offenen Hashing mit dem der Verkettung der Überläufer. Alle Überläufer befinden sich in einer Überlaufkette, die in der Hashtabelle abgespeichert ist. Jeder Eintrag der Hashtabelle besteht aus dem Schlüssel samt dem zugehörigen Datensatz und einem Zeiger (realisiert als Hashadresse) auf den nächsten Eintrag in der Überlaufkette. Ein einzufügender Schlüssel wird ans Ende der Überlaufkette angehängt.

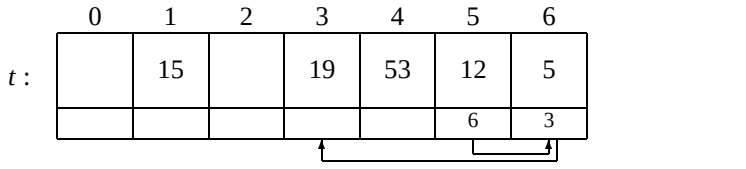
Betrachten wir das Beispiel, das uns bisher begleitet hat. Die Hashtabellengröße sei 7, die Hashfunktion sei k mod 7 für Schlüssel k, und die Schlüssel 12, 53, 5, 15, 19, 43 seien in dieser Reihenfolge in die anfangs leere Hashtabelle einzufügen. Nach Einfügen von 12, 53 erhalten wir folgende Situation:



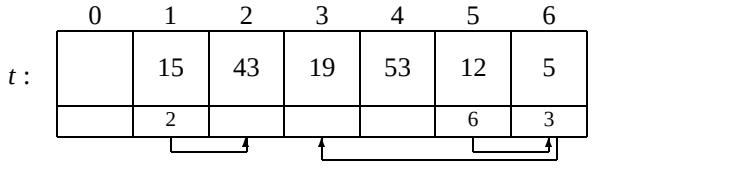
Beim Einfügen von Schlüssel 5 stellen wir fest, daß $h(5) = 5$ bereits mit Schlüssel 12 belegt ist; es gibt aber noch keine Überläufer, Schlüssel 12 ist das Ende der Überlaufkette. Statt nun dynamisch einen neuen Speicherplatz zu allokieren (wie beim Verketten der Überläufer außerhalb der Hashtabelle), müssen wir uns hier für einen freien Platz in der Hashtabelle entscheiden, den wir mit Schlüssel 5 belegen wollen. Wir legen uns darauf fest, den von rechts her ersten freien Platz in der Hashtabelle zu nehmen (also denjenigen mit höchster Hashadresse). Schlüssel 5 wird also bei $t[6]$ eingetragen und mit $t[5]$ verkettet:



Nach Einfügen von 15 und 19 ergibt sich



und nach Einfügen von 43 schließlich



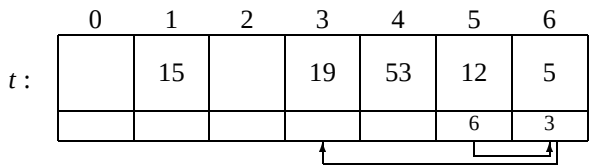
Methode: *Coalesced Hashing*
Jeder Eintrag der Hashtabelle besteht aus dem Datensatz mit Schlüssel und einem Verweis (Hashadresse) auf den Nachfolger in der Überlaufkette.

Suchen nach dem Schlüssel k: Beginne bei $t[h(k)]$ und folge den Verweisen der Überlaufkette, bis entweder k gefunden wurde (erfolgreiche Suche) oder das Ende der Überlaufkette erreicht ist (erfolglose Suche).

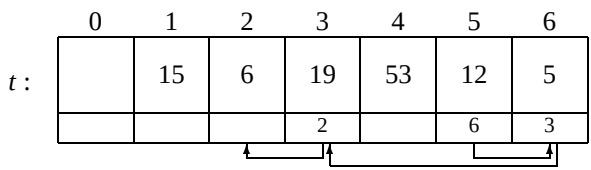
Einfügen eines Schlüssels k: Suche nach k; die Suche verläuft erfolglos (sonst wird k nicht eingefügt) und endet am Ende einer Überlaufkette oder bei $t[h(k)]$. Im letzteren Fall trage k in $t[h(k)]$ ein; sonst wähle das freie Hashtabellenelement mit größter Hashadresse, hänge es an die Überlaufkette an und trage k dort ein.

Entfernen eines Schlüssels k: Suche nach k; die Suche verläuft erfolgreich (sonst kann k nicht entfernt werden). Steht k in $t[h(k)]$, so lösche k dort; verweist $t[h(k)]$ auf ein Element der Überlaufkette, so übertrage dieses nach $t[h(k)]$. Lösche k an seiner alten Position und adjustiere den Verweis auf das nächste Element in der Überlaufkette, falls k in der Überlaufkette auftritt.

Bis auf das Auswählen eines freien Eintrags in der Hashtabelle gleicht also diese Methode völlig dem Hashing mit separater Verkettung der Überläufer. Es gibt aber einen wichtigen Unterschied bei den entstehenden Situationen. Fügen wir gemäß obiger Regel in der nach Einfügen von 12, 53, 5, 15, 19 entstandenen Situation



statt des Schlüssel 43 (wie im obigen Beispiel) jetzt den Schlüssel 6 ein, so stellen wir fest, daß $t[h(6)] = t[6]$ bereits belegt ist, und hängen Schlüssel 6 an das Ende der Überlaufkette ab $t[6]$ an:



Die Überlaufkette ab $t[5]$ enthält somit auch den Schlüssel 6, obwohl 6 kein Synonym von 5 ist; entsprechend enthält die Überlaufkette ab $t[6]$ auch die Schlüssel 5 und 19, obwohl beide keine Synonyme von Schlüssel 6 sind. Die beiden Überlaufketten von Schlüsseln 5 und 6 sind verschmolzen. Die Korrektheit der angegebenen Methode wird hiervon nicht beeinträchtigt, wohl aber die Effizienz. Da Überlaufketten etwas länger werden als beim separaten Verketten, dauert die Suche etwas länger; im Durchschnitt ist das aber sehr wenig, wie eine Analyse zeigt:

$$C'_n \approx 1 + \frac{1}{4}\left(e^{2\alpha} - 1 - 2\alpha\right)$$

$$C_n \approx 1 + \frac{1}{8\alpha}\left(e^{2\alpha} - 1 - 2\alpha\right) + \frac{1}{4}\alpha$$

Tabelle 4.5 vermittelt durch einige in diese Formeln eingesetzte Werte von α einen Eindruck von der Effizienz des Coalesced Hashing.

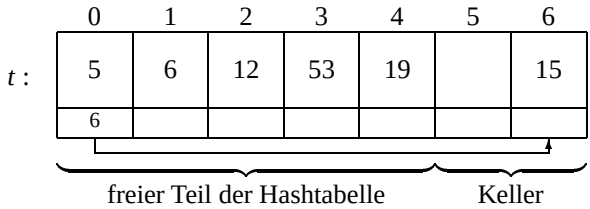
Anzahl betrachteter Einträge	Coalesced Hashing	
	erfolgreich	erfolglos
$\alpha = 0.50$	1.30	1.18
0.90	1.68	1.81
0.95	1.74	1.95
1.00	1.80	2.10

Tabelle 4.5

Diese beachtliche Effizienz der Suche beim Coalesced Hashing wird erzielt um den Preis eines etwas höheren Speicherplatzbedarfs für die Verweise und eines etwas höheren Zeitbedarfs für das Einfügen, da ja nach einem freien Platz in der Hashtabelle gesucht werden muß. Verzichtet man auf das Wiederbelegen der Plätze als *entfernt* markierter Einträge, so kann man einen einzigen Verweis (Hashadresse), ausgehend von Hashadresse $m-1$, schrittweise durch die Hashtabelle bewegen (lineares Suchen, vgl. Kapitel 3) und an einem gefundenen freien Platz bis zur nächsten Einfügeoperation ruhen lassen; alle Plätze mit höherer Hashadresse sind ja schon belegt. Dann durchläuft dieser Verweis höchstens soviele Plätze, wie Schlüssel in die Hashtabelle eingefügt worden sind. Im Durchschnitt benötigt eine Einfüge-Operation also gerade einen Versuch, um einen leeren Platz in der Tabelle zu finden. Man kann zeigen, daß für eine zufällige Einfügung etwa $\alpha \cdot e^{\alpha}$ Plätze auf der Suche nach einem freien Platz inspiziert werden müssen [89].

Das Coalesced Hashing in der beschriebenen Form geht zurück auf Williams [193]. Inzwischen sind viele Varianten des Verfahrens untersucht worden. Eine Wesentliche sperrt einen Teil der Hashtabelle, den *Keller*, für die normale Benutzung und weist diesem Teil nur Überläufer zu. Sobald der Keller voll ist, wird auch der Platz im Rest der Hashtabelle verwendet.

In unserem Beispiel einer Hashtabelle der Größe 7 wählen wir $t[0]$ bis $t[4]$ als frei verfügbaren Tei Hashtabelle; $t[5]$ bis $t[6]$ ist der Keller. Die Hashfunktion ändert sich damit zu $h(k) = k \bmod 5$, die Algorithmen für das Suchen, Einfügen und Entf bleiben unverändert. Fügen wir nun die Schlüssel 12, 53, 5, 15, 19, 6 in die Hashtabelle ein, so entsteht folgende Situation:



Man erwartet, daß sich durch die Verwendung des Kellers die Verschmelzung von Überlaufketten reduziert; solange Überläufer im Keller abgelegt werden, gibt es keine Verschmelzung. Im Beispiel ist dies der Fall. Die Verschmelzung von Überlaufketten ist also umso geringer, je größer der Keller ist. Andererseits reduziert sich, wenn ein Speicherplatz fester Größe insgesamt zur Verfügung steht, bei großem Keller der freie Teil der Hashtabelle; dadurch werden Kollisionen wahrscheinlicher, und damit gibt es mehr Überläufer. Im Extremfall ist nur $t[0]$ frei, und $t[1]$ bis $t[m-1]$ bilden den Keller; dann sind alle Schlüssel in einer einzigen Überlaufkette gespeichert. Die Anzahl der Überläufer sinkt also mit kleinerem Keller. Nennen wir m_h die Anzahl der frei verfügbaren Plätze der Hashtabelle und m_k die Anzahl der Plätze im Keller $(m_h + m_k = m)$, dann ist das Verhältnis m_h/m für die Effizienz der Suche entscheidend. In einer vollen Hashtabelle ist der Erwartungswert für die erfolgreiche Suche bei $m_h/m = 0.853\ldots$ minimal, für die erfolglose Suche bei $m_h/m = 0.782\ldots$ [67]; der Wert $m_h/m = 0.86$ scheint ein guter Kompromiß für beide Fälle und einen großen Bereich von Belegungsfaktoren zu sein.

4.4 Dynamische Hashverfahren

Wenngleich alle der von uns bisher vorgestellten Hashverfahren die Operationen Einfügen und Entfernen unterstützen, so ist die tatsächlich realisierte Dynamik der Verfahren nur begrenzt. Bei offenen Hashverfahren ist das Einfügen von Schlüsseln über den vorgesehenen Speicherplatz hinaus unmöglich; bei Hashverfahren mit Verkettung der Überläufer ist es 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. Wir wollen in diesem Abschnitt vier Hashverfahren für stark wachsende oder schrumpfende Datenbestände vorstellen. Solche *dynamischen Hashverfahren* (vgl. Übersichtsartikel [47], [102]) sind insbesondere für Daten von Bedeutung, die auf Externspeichern verwaltet werden, wie etwa die Datensätze einer Datenbank (man kann sie aber auch als Hauptspeicherstrukturen einsetzen [103]). Die kleinste Einheit des Zugriffs auf den Externspeicher ist der *Datenblock*; eine Lese- bzw. Schreiboperation auf den Externspeicher überträgt einen Block vom Externspeicher in den Hauptspeicher des Rechners bzw. vom Hauptspeicher auf den Externspeicher. Bei den meisten Rechnern haben Blöcke eine feste Größe, typi-

scherweise 512 Byte oder ein Vielfaches davon. Damit können in der Regel mehrere Datensätze in einem Block gespeichert werden. Sei *b* (Blockkapazität) die Anzahl der Datensätze, die neben einigen Verwaltungsinformationen in einem Block gespeichert werden. Dann können wir einen Block wie folgt beschreiben:

```
const
  b = 30; {Beispiel für Blockkapazität}
type
  block = record
            verwaltung : {z.B. Anzahl belegter Einträge, etc.};
            eintrag : array[1 .. b] of datensatz
          end
```

Anstelle einer Hashtabelle verwenden wir dann eine Datei, bestehend aus Blöcken:

```
type
  hashdatei = file of block
```

Wir setzen voraus, daß ein Block in der Datei durch seine relative Adresse, beginnend bei Adresse 0, direkt angesprochen werden kann. Eine Datei von *m* Blöcken mit Adressen 0 bis $m-1$ wächst durch Anhängen des Blocks mit Adresse *m* und schrumpft durch Abhängen des Blocks mit Adresse $m-1$. Eine Hashfunktion ordnet einem Schlüssel *k* die relative Adresse $h(k)$ mit $0 \leq h(k) \leq m-1$ des Blocks zu, in dem der Datensatz mit Schlüssel *k* zu speichern ist. Adreßkollisionen sind also hier kein Problem, solange es nicht mehr als *b* Synonyme gibt, denn diese können ja gemeinsam in einem Datenblock gespeichert werden. Wir wollen uns nicht darum kümmern, wie Schlüssel *innerhalb* eines Datenblocks eingefügt, entfernt und wiedergefunden werden können, weil dies wegen der kleinen Größe von *b* nur geringen Rechenaufwand bedeutet. Um ein Vielfaches teurer sind dagegen Blockzugriffe, also das Lesen oder Schreiben eines ganzen Blocks. Als Maß für die Effizienz von externen, dynamischen Hashverfahren verwendet man daher üblicherweise die Anzahl erforderlicher Blockzugriffe.

Die besondere Problematik dynamischer Hashverfahren liegt nun darin, daß man nicht einfach ein und dieselbe Hashfunktion bei sich änderndem *m* verwenden kann, weil man sonst gespeicherte Schlüssel nicht unbedingt wiederfindet, und daß man eine globale Reorganisation, also das Umspeichern sämtlicher Datensätze gemäß einem geänderten *m*, aus Effizienzgründen vermeiden möchte. Man kann beide Probleme lösen, indem man nur Teilbereiche des gesamten Speichers — meist einzelne Datenblöcke — reorganisiert und sich merkt, für welche Teilbereiche eine Reorganisation erfolgt ist und welche neue Hashfunktion dabei verwendet wurde. Bei den Vereinbarungen

```
var
  hd : hashdatei;
  m {aktuelle Anzahl der Blöcke in hd},
  n {aktuelle Anzahl in hd gespeicherter Datensätze} : integer
```

kann für viele dynamische Hashverfahren ein Rahmen für das Einfügen eines Datensatzes *ds* in eine Hashdatei *hd* mit *m* Blöcken und *n* aktuell gespeicherten Datensätzen wie folgt beschrieben werden:

```
while hd mit m Blöcken und n Datensätzen ist für ds zu klein do
  begin {erweitere hd um einen Block}
    füge neuen, leeren Block mit Adresse m an hd an;
    wähle Blockadresse i im Bereich 0 bis m − 1;
    adaptiere Hashfunktion h;
    verteile Datensätze aus Block i gemäß h auf Blöcke i und m;
    m := m + 1
  end;
trage ds in Block mit Adresse h(ds.k) in hd ein
```

Die Suche nach einem Datensatz mit Schlüssel k besteht dann einfach im Prüfen des Inhalts des Blocks mit Adresse $h(k)$. Beim Entfernen von Einträgen wird — analog zum Einfügen — überprüft, ob die Hashdatei zu groß ist; sie wird gegebenenfalls um einen Block verkleinert. Innerhalb des gegebenen Rahmens unterscheiden sich dynamische Hashverfahren im Kriterium für das Erweitern oder Schrumpfen der Hashdatei um einen Block und in der Wahl der adaptierten Hashfunktion und ihrer Speicherung und damit der Wahl des Blocks der zu verteilenden Datensätze.

Im nächsten Abschnitt werden wir ein dynamisches Hashverfahren, das *lineare Hashing*, mit einer sehr einfachen Hashfunktion vorstellen; genauer an die zu speichernden Daten angepaßte Verfahren, bei denen das Speichern der Hashfunktion selbst zu einem Datenverwaltungsproblem wird, präsentieren wir dann in den folgenden Abschnitten.

4.4.1 Lineares Hashing

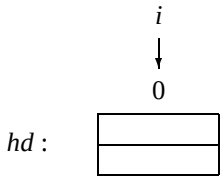
Bei linearem Hashing [111, 112] besteht die Hashfunktion h zu jedem Zeitpunkt aus höchstens zwei einfachen Hashfunktionen h_1 und h_2, die jeweils die gesamte Hashdatei adressieren. Für eine anfängliche Dateigröße von m_0 Blöcken der Hashdatei und eine aktuelle Größe von m Blöcken, wobei $m_0 \cdot 2^l \leq m < m_0 \cdot 2^{l+1}$ für eine natürliche Zahl l gilt, adressiert h_1 den Adreßbereich $0 \,..\, m_0 \cdot 2^l - 1$ und h_2 den Adreßbereich $0 \,..\, m_0 \cdot 2^{l+1} - 1$. Dabei ist je nach Schlüssel k entweder $h_2(k) = h_1(k)$ oder $h_2(k) = h_1(k) + m_0 \cdot 2^l$. Der *Dateilevel* l gibt dabei die Anzahl der kompletten Dateiverdoppelungen an. Wenn also $h_1(k)$ die Datensätze des Blocks $h_1(k)$ verteilt, ergibt sich nach Einfügen eines leeren Blocks mit Adresse m die Adresse i des Blocks, der die zu verteilenden Datensätze enthält, als $i = m - m_0 \cdot 2^l$. Damit durchläuft i der Reihe nach die Adressen 0 bis $m_0 \cdot 2^l - 1$; anfangs ist $i = 0$ und $l = 0$. Hashfunktion h_1 ist dann für diejenigen Schlüssel k anzuwenden, für die $i \leq h_1(k) \leq m_0 \cdot 2^l - 1$ gilt. Für alle anderen Schlüssel k, das sind diejenigen mit $0 \leq h_1(k) < i$, ist h_2 anzuwenden. Der gesamte dynamische Hashdateizustand ist also durch den Dateilevel l und die Adresse i der nächsten Seite mit zu verteilenden Datensätzen (der nächsten zu *splittenden* Seite) charakterisiert, wenn h_1 und h_2 festliegen. Eine geeignete Wahl für h_1 und h_2 ergibt sich beispielsweise durch Anwendung der Divisions-Rest-Methode, mit $h_1(k) = k \bmod (m_0 \cdot 2^l)$ und $h_2(k) = k \bmod (m_0 \cdot 2^{l+1})$.

Weil die nächste zu splittende Seite unabhängig von einem einzufügenden Datensatz festliegt, kann bei linearem Hashing keine Rücksicht darauf genommen werden, ob ein Datensatz noch in dem ihm zugeordneten Datenblock Platz findet. Man entscheidet sich hier dafür, gemäß der aktuellen Hashfunktion h bei mehr als b Synonymen Ketten von Blöcken für diese Synonyme zu bilden, ganz ähnlich wie bei Hashverfahren mit Verkettung der Überläufer. Überlaufblöcke werden dabei in einem eigenen Speicherbereich untergebracht. Wir werden der Einfachheit halber einen durch die Hashfunktion adressierten Block (*Primärblock*) und evtl. ihm zugeordnete Überlaufblöcke (*Sekundärblöcke*) nicht unterscheiden und logisch wie einen Block behandeln. Es sollte aber klar sein, daß die Verwendung von Überlaufblöcken zusätzliche Externspeicherzugriffe und damit Effizienzeinbußen nach sich zieht.

Das Kriterium für das Erweitern der Hashdatei um einen Block ist bei linearem Hashing üblicherweise der Belegungsfaktor $n/(b \cdot m)$ der Hashdatei. Würde er als Folge einer Einfügeoperation einen festgesetzten Schwellenwert überschreiten, so wird die Hashdatei erweitert; würde er als Folge einer Entferne-Operation einen (anderen) Schwellenwert unterschreiten, so wird die Hashdatei um einen Block verkleinert. Das Verkleinern erfolgt hier völlig symmetrisch zum Erweitern der Datei. Die Einträge im Block mit Adresse m und im Block mit Adresse $m - m_0 \cdot 2^l$ werden zusammengefaßt und im Block mit Adresse $m - m_0 \cdot 2^l$ abgelegt; i und l werden wiederum entsprechend angepaßt.

Aber auch andere Hashfunktionen als nach der Divisions-Rest-Methode sind vorstellbar. Für manche Operationen ist es wünschenswert, in einem Datenblock möglichst Datensätze mit nahe beieinander liegenden Schlüsseln zu speichern — etwa beim Finden des einem Suchschlüssel nächstgelegenen Schlüssels (best match query, nearest neighbour query) oder beim Finden aller Schlüssel in einem gewissen Bereich (range query). Die Divisions-Rest-Methode leistet diese *ordnungserhaltende* Abbildung von Schlüsseln auf Adressen offenbar nicht. So erhält man beispielsweise eine ordnungserhaltende Abbildung ganzzahliger Schlüssel, indem man diese durch Bitstrings fester Länge darstellt, und die von links her ersten (also in der Zahl höchstwertigen) l Bits eines jeden Schlüssels in umgekehrter Reihenfolge, also von rechts nach links, als Dualzahl liest und als Hashadresse im Bereich von 0 bis 2^l ansieht [136]. Um Häufungen von Schlüsseln zu vermeiden, betrachtet man manchmal auch Bitstrings, die sich aus Schlüsseln durch Anwenden einer Hashfunktion und entsprechende Interpretation des Hashfunktionswertes ergeben, sogenannte *Pseudoschlüssel*. Wir wollen dies hier nicht mehr explizit berücksichtigen, weil dieser Unterschied keinen Effekt auf die vorgestellten Verfahren hat, und stattdessen stets Schlüssel direkt als Bitstrings ansehen.

Beispiel: Betrachten wir die mit linearem Hashing und der beschriebenen ordnungserhaltenden Hashfunktion organisierte Hashdatei, die sich durch Einfügen der Schlüssel 12, 53, 5, 15, 2, 19, 43 in dieser Reihenfolge in die Hashdatei ergibt, die anfangs aus einem leeren Datenblock besteht ($m_0 = 1$). In jedem Datenblock können bis zu zwei Datensätze gespeichert werden; wir zeigen im folgenden nur deren Schlüssel. Wählen wir 0.9 als Schwellenwert des Belegungsfaktors zum Erweitern der Datei und die feste Darstellungslänge von 6 Bits für jeden Schlüssel, so ergibt sich bei der in Abbildung 4.3 gezeigten Ausgangssituation vor dem Einfügen des zweiten Schlüssels ein



dezimal	dual
12	001100
53	110101
5	000101
15	001111
2	000010
19	010011
43	101011

Abbildung 4.3

Split des Blocks 0 in Blöcke 0 und 1, und nach dem Eintragen dieses Schlüssels die in Abbildung 4.4 gezeigte Situation.

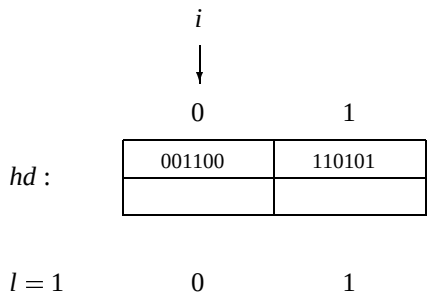


Abbildung 4.4

Schlüssel 5 kann auf dem freien Platz in Block 0 gespeichert werden; der Schwellenwert für den Belegungsfaktor wird nicht überschritten. Dies geschieht erst bei der Einfügung von Schlüssel 15. Hierbei wird ein neuer Block, nämlich mit Adresse 2, an die Hashdatei angehängt. Die in Block 0 gespeicherten Schlüssel werden gemäß ihrem zweiten Bit auf Blöcke 0 und 2 verteilt: Schlüssel mit führenden Bits 00 bleiben im Block 0, Schlüssel mit führenden Bits 01 (solche treten bisher nicht auf) werden in Block 2 gespeichert (01 rückwärts gelesen ergibt 10, also die duale Darstellung der Hashadresse 2). Dann wird der einzufügende Schlüssel 15 gemäß seiner beiden führenden Bits in Block 0 eingetragen. Hierbei muß für Block 0 ein Überlaufblock angelegt werden. Die Adresse des Überlaufblocks entstammt einem anderen Adreßbereich und sei hier nicht von Bedeutung. Damit ergibt sich die in Abbildung 4.5 dargestellte Situation.

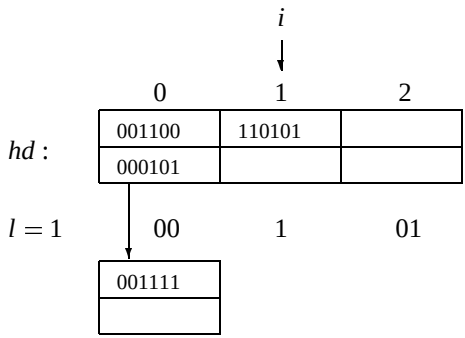


Abbildung 4.5

Schlüssel 2 kann ohne weitere Reorganisation der Datei in Block 0 (genauer: dessen Überlaufblock) eingefügt werden. Erst Schlüssel 19 führt wieder zu einem Überschreiten des Schwellenwerts des Belegungsfaktors und damit zum Anhängen eines neuen Datenblocks an die Hashdatei. Damit ist eine weitere Dateiverdoppelung beendet, und wir erhalten die in Abbildung 4.6 gezeigte Situation.

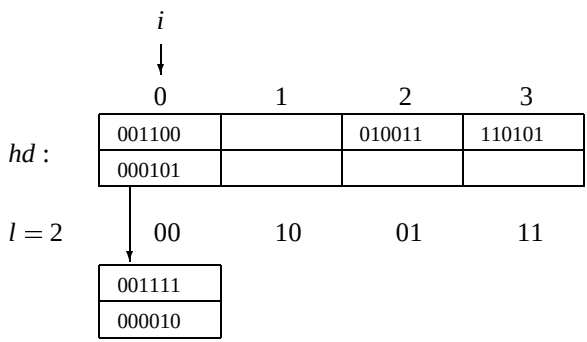


Abbildung 4.6

Schließlich kann Schlüssel 43 in Datenblock 1 eingetragen werden und die Folge der Einfügungen ist beendet.

Beziehen wir die Anzahl l bereits erfolgter Dateiverdoppelungen in die Hashfunktion ein, so adressiert bei aktueller Dateigröße m und Anfangsgröße m_0 mit nächstem zu splittendem Datenblock i offenbar die Hashfunktion h_l die Datenblöcke mit Adressen i bis $m_0 \cdot 2^l - 1$, und h_{l+1} adressiert Blöcke 0 bis $i-1$ und $m_0 \cdot 2^l$ bis m, wie in Abbildung 4.7 gezeigt.

Da bei linearem Hashing nach dem Erweitern der Datei um einen Block das Kriterium für das Erweitern der Datei sicher nicht mehr erfüllt ist, realisiert schon die folgende Spezialisierung des allgemeinen Prinzips dynamischer Hashverfahren die Einfügeoperation:

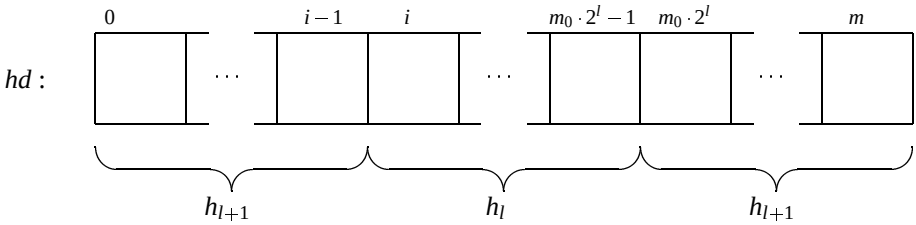


Abbildung 4.7

```
procedure Einfügen (ds: datensatz; var hd: hashdatei;
          var m,n,i,l : integer; schwelle: real);
{fügt Datensatz ds in Hashdatei hd mit Dateilevel l ein}
begin
  if (n+1)/(b·m) > schwelle
    then {erweitere hd um einen Block}
      begin
        reserviere Block mit Adresse m für hd;
        verteile Datensätze aus Block i gemäß h_{l+1} auf Blöcke i und m;
        m := m+1;
        if i < m0·2^l − 1
          then i := i+1
          else {Dateiverdoppelung ist erfolgt}
            begin
              i := 0;
              l := l+1
            end
      end;
  n := n+1;
  {bestimme den ds.k zugeordneten Block}
  if (i ≤ h_l(ds.k)) and (h_l(ds.k) ≤ m0·2^l − 1)
    then trage ds im Block h_l(ds.k) ein
    else trage ds im Block h_{l+1}(ds.k) ein
end
```

Wir sparen uns die genaue algorithmische Beschreibung der Operationen Suchen und Entfernen, weil das Suchen nach einem Datensatz mit Schlüssel k lediglich das Bestimmen des k zugeordneten Blocks (wie am Ende der Einfügeprozedur) und das Inspizieren dieses Blocks ist, und weil das Entfernen mit einem eventuellen Verschmelzen von Blöcken völlig symmetrisch zum Einfügen operiert. Das bedeutet auch, daß die Entferneoperation ebenso wie die Einfügeoperation beim Reorganisieren von Teilen der Hashdatei keinerlei Rücksicht auf die aktuelle Verteilung der Datensätze nimmt. So wurde etwa in unserem Beispiel ein neuer Block (derjenige mit Adresse 2) angelegt, ohne daß er Datensätze des übergelaufenen Blocks 0 aufnahm. Bei Schlüsseln, die über

dem Universum K aller möglichen Schlüssel einigermaßen gleichverteilt sind, ist dies nicht unbedingt ein gravierender Nachteil. Man kann zeigen, daß bei Gleichverteilung der Datensätze die erwartete Speicherplatzausnutzung in einem dynamischen Hashverfahren, das mit rekursiver Halbierung (Verteilung von Datensätzen aus einem Block auf zwei Blöcke mit gleich großem Hashadreßbereich) arbeitet, ohne Berücksichtigung von Überläufern bei $\ln 2$, also etwa 69 %, liegt [51, 100, 124]. Strebt man jedoch einen konstant hohen Belegungsfaktor an, so ergibt sich zwischen zwei aufeinander folgenden Dateiverdoppelungen (man sagt auch: während einer *Expansion*) eine gewisse Diskontinuität bei der erwarteten Länge von Überlaufketten. Zu Beginn der Expansion werden alle Überlaufketten etwa gleich lang sein, aber gegen Ende der Expansion werden die Überlaufketten bereits gesplitteter Blöcke wesentlich kürzer sein als diejenigen noch nicht gesplitteter Blöcke. Dieser spürbare Effekt läßt sich mit Hilfe *partieller Expansionen* abschwächen [101]. Dazu verteilt man etwa in der ersten partiellen Expansion den Inhalt von jeweils zwei Datenblöcken auf drei Datenblöcke, von denen einer die Datei vergrößert, und in einer zweiten partiellen Expansion entsprechend von drei Datenblöcken auf vier Datenblöcke.

Trotzdem wird häufig die Speicherplatzausnutzung der Überlaufblöcke deutlich hinter derjenigen der Primärblöcke zurückbleiben, insbesondere dann, wenn die Kapazität der Überlaufblöcke groß ist. Man kann nun versuchen, mehreren Primärblöcken gemeinsam wenige Überlaufblöcke zuzuordnen (*overflow bucket sharing*). Dann muß man sich fragen, wie diese verwaltet werden sollen. Da eine statische Struktur für Überlaufdatensätze der Dynamik des linearen Hashing entgegensteht, kann man das Problem der Verwaltung von Überlaufdatensätzen als das ursprüngliche Problem ansehen, beschränkt auf eine kleinere Anzahl von Datensätzen. Es ist demnach natürlich, diese rekursiv mittels linearem Hashing zu verwalten [157]. So erhält man mehrere Rekursionsebenen von linearem Hashing, bis schließlich keine Überläufer mehr auftreten. Die resultierende bessere Speicherplatzausnutzung erkauft man sich dabei durch Operationen, die sich über die rekursiven Ebenen der Daten fortsetzen können.

Es ist klar, daß bei stark ungleich verteilten Schlüsseln lineares Hashing degenerieren kann, im Extremfall zur Verwaltung einer einzigen linearen Kette von Überlaufblöcken. Eine Garantie für die Anzahl der zur Suche benötigten Externzugriffe läßt sich also nicht geben. Wir wollen in den nächsten Abschnitten andere dynamische Hashverfahren vorstellen, bei denen solch eine Garantie gegeben werden kann.

4.4.2 Virtuelles Hashing

Bei virtuellem Hashing [111, 110] werden — im Unterschied zu linearem Hashing — Überlaufblöcke vollständig vermieden. Anstatt — wie bei linearem Hashing — die Hashdatei nur um jeweils einen Block zu vergrößern, verdoppelt man die Größe der Hashdatei bei virtuellem Hashing in einem Schritt, wenn eine Einfügeoperation in einen bereits vollen Datenblock durchgeführt werden soll und nicht schon beide Blöcke, auf welche die Datensätze verteilt werden müssen, zur Hashdatei gehören (Verfahren VH1 in [111] und [110]). Natürlich sollen nach einer Dateiverdoppelung nur die Sätze des überlaufenden Blocks verteilt werden, und nicht etwa die Sätze anderer Blöcke. Das kann dazu führen, daß Sätze nicht gemäß der Hashfunktion gespeichert sind, die der

aktuellen Hashdateigröße entspricht, sondern gemäß einer für eine kleinere Hashdateigröße verwendeten Hashfunktion. Es genügt also nicht, zu jedem Zeitpunkt mit nur zwei Hashfunktionen alle Datenseiten zu adressieren. Wir müssen vielmehr für alle im Zeitablauf eingesetzten Hashfunktionen vermerken, für welche Datenblöcke sie aktuell relevant sind. Bei virtuellem Hashing geschieht dies mit je einer Bittabelle für jede erfolgte Dateiverdoppelung und damit für jede im Zeitablauf verwendete Hashfunktion, außer der letzten. Sei wieder l die Anzahl der erfolgten Dateiverdoppelungen, und m_0 die Anfangsgröße der Hashdatei. Dann speichert für $0 \leq j \leq l-1$ die j-te Bittabelle bit_j gerade $m_0 \cdot 2^j$ Bits, eines für jeden der Datenblöcke 0 bis $m_0 \cdot 2^j - 1$. Ein Bit hat genau dann den Wert 1, wenn die Hashfunktion h_j nicht ausreicht, um die gemäß der aktuellen Hashfunktion h auf diesen Block gehörenden Datensätze in der aktuellen Hashdatei zu adressieren. Dann muß also eine der Hashfunktionen $h_{j+1}, h_{j+2}, \ldots$ verwendet werden; h_j ist für diesen Block veraltet.

Beispiel: Betrachten wir wieder das Einfügen der Schlüssel 12, 53, 5, 15, 2, 19, 43 in dieser Reihenfolge in die Hashdatei, die anfangs aus einem leeren Datenblock besteht $(m_0 = 1)$. In jedem Datenblock finden zwei Datensätze Platz; wieder sei $h_j(k)$ mit $0 \leq j \leq l$ der Wert der Dualzahl der ersten j Bits von k, rückwärts gelesen. Schlüssel 12 und 53 (vgl. deren Dualdarstellung in Abbildung 4.3) werden mit $h_0(k) \equiv 0$ in Datenblock 0 eingefügt. Das Einfügen von Schlüssel 5 bringt Block 0 zum Überlaufen; er muß gesplittet werden. Die Hashdateigröße wird von einem auf zwei Blöcke verdoppelt, und in Bittabelle bit_0 wird vermerkt, daß h_0 zur Adressierung nicht ausreicht, wie in Abbildung 4.8 dargestellt.

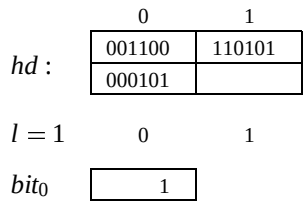


Abbildung 4.8

Für 15, den nächsten einzufügenden Schlüssel, wird nun die aktuelle Hashadresse berechnet. Hierfür wird zunächst h_0 auf Schlüssel 15 angewandt, mit Resultat 0. Dann wird $bit_0[0]$ überprüft; weil dieses Bit den Wert 1 hat, wird nun $h_1(15)$ berechnet, wieder mit Resultat 0. Weil erst eine Dateiverdoppelung erfolgt ist, gibt es keine weitere Bittabelle, und h_1 ist die auf Schlüssel 15 anzuwendende Hashfunktion. Also läuft Block 0 erneut über. Da h_1 die auf den einzufügenden Schlüssel anzuwendende Hashfunktion war, muß Block 0 mittels h_2 gesplittet werden, also seinen Inhalt und den einzufügenden Schlüssel auf Blöcke 0 und 2 verteilen. Nachdem aber Block 2 nicht schon zur Hashdatei gehört, ist eine Dateiverdoppelung erforderlich. Sie führt zu der in Abbildung 4.9 gezeigten Situation.

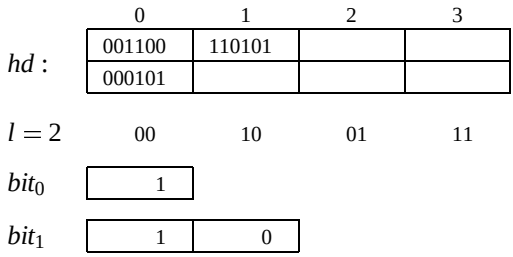


Abbildung 4.9

Auch hier ist Schlüssel 15 in den vollen Block $h_2(15)$ einzufügen, und wieder ist eine Dateiverdoppelung erforderlich. Allgemein ist eine Dateiverdoppelung dann erforderlich, wenn ein Datensatz in einen vollen Block eingefügt werden soll, dessen höchstes vermerktes Bit, also bit_{l-1}, eine 1 ist. Wir erhalten somit die in Abbildung 4.10 gezeigte Situation, in der die Datensätze aus Block 0 und der einzufügende Datensatz auf Blöcke 0 und 4 verteilt sind. Mit Ausnahme von Block 0 ist bisher kein Block gesplittet worden.

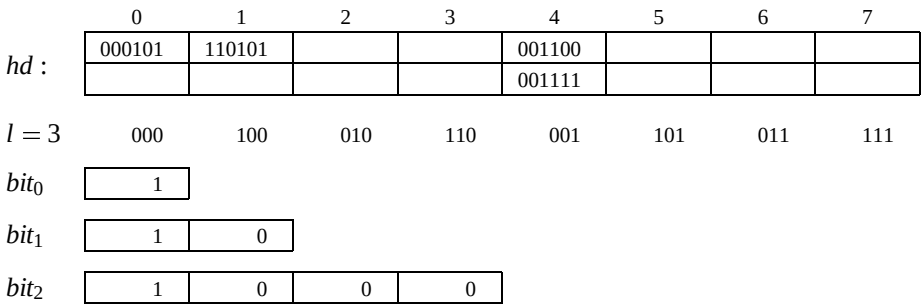


Abbildung 4.10

Das Einfügen der restlichen drei Schlüssel verläuft ohne weitere Dateiverdoppelungen. Für Schlüssel 2 erhalten wir $bit_0[h_0(2)] = 1$, $bit_1[h_1(2)] = 1$, $bit_2[h_2(2)] = 1$, und wegen $l = 3$ wird Schlüssel 2 schließlich gemäß $h_3(2)$ in Block 0 eingefügt. Entsprechend landet Schlüssel 19 mit $bit_0[h_0(19)] = 1$, $bit_1[h_1(19)] = 1$ und $bit_2[h_2(19)] = 0$ gemäß $h_2(19)$ im Block 2. Schließlich wird Schlüssel 43 gemäß $h_1(43)$ in Block 1 eingefügt.

Allgemein läßt sich also für eine Hashdatei der Anfangsgröße m_0 mit l erfolgten Verdoppelungen (Dateilevel l) mit Hilfe von l Bittabellen bit_j, $0 \le j \le l-1$, der Typen

type bit_j = **array** $[0 \,..\, m_0 \cdot 2^j - 1]$ **of** *bit*

die aktuelle Hashadresse $h(k)$ eines Schlüssels k wie folgt ermitteln:

```
j := 0;
while (j < l) and (bit_j[h_j(k)] = 1) do j := j + 1;
h(k) := h_j(k)
```

Adressiert Hashfunktion h_j die Sätze eines Datenblocks, so bezeichnen wir j als den *Level* dieses Blocks. Wie wir am Beispiel gesehen haben, bewirkt das Einfügen eines Datensatzes in einen vollen Block mit Level l eine Dateiverdoppelung. Der Dateilevel ändert sich auf $l+1$, und eine neue Bittabelle bit_l mit einer 1 für den betroffenen Datenblock und sonst lauter Nullen wird angelegt. Außerdem werden die Datensätze des betroffenen Blocks verteilt. Beim Einfügen in einen vollen Block mit kleinerem Level entfallen die Dateiverdoppelung und das Anlegen einer Bittabelle; es müssen lediglich ein vorhandener Bittabelleneintrag von 0 auf 1 verändert und die Datensätze verteilt werden. Damit kann das Einfügen ohne Rücksicht auf Implementierungsdetails wie folgt beschrieben werden:

```
procedure Einfügen(ds: datensatz; var hd: hashdatei; var l: integer;
                   var bit: sequence of bittabelle);
{fügt Datensatz ds in Hashdatei hd mit Dateilevel l und Bittabellen
 bit_0 bis bit_{l-1} ein}
var j : integer;
begin
  ermittle Hashadresse h_j(ds.k) und Level j des Blocks, in den k
    einzufügen ist;
  while Block h_j(ds.k) ist voll do
    begin
      if j = l
        then {Block hat Dateilevel l}
          begin
            verdopple hd;
            l := l + 1;
            kreiere bit_l = (0,0,...,0)
          end;
      bit_j[h_j(ds.k)] := 1;
      verteile Sätze von Block h_j(ds.k) auf Blöcke h_j(ds.k) und
        h_{j+1}(ds.k) gemäß h_{j+1};
      ermittle erneut h_j(ds.k) und Level j;
    end;
  trage ds in Block h_j(ds.k) ein
end
```

Nimmt man hierbei an, daß alle Bittabellen im Hauptspeicher gehalten werden können, so genügt für das Wiederfinden eines Datensatzes bei gegebenem Schlüssel offenbar ein Externzugriff. Diesen garantiert extrem schnellen Zugriff erkauft man sich aber mit einer sehr schlechten Speicherplatzausnutzung. Für n Datensätze in der Datei und b Sätze pro Datenseite kann man zeigen, daß die Speicherplatzausnutzung von

der Größenordnung $O(n^{-(1/b)})$ ist, also mit wachsender Dateigröße abnimmt. Außerdem ist klar, daß die Speicherplatzausnutzung stark schwankt: Unmittelbar nach einer Dateiverdoppelung sinkt sie schlagartig auf die Hälfte. Diesen Effekt kann man vermeiden, wenn man die Hashfunktion nur zur Adressierung *virtueller* und nicht tatsächlicher Datenblöcke verwendet. Zu diesem Zweck übernimmt eine Adreßtabelle die Rolle der Hashdatei: Die jeweils aktuelle Hashfunktion adressiert einen Eintrag der Adreßtabelle. Ein Adreßtabelleneintrag ist dann lediglich die Adresse eines Blocks der Hashdatei. Statt einer Dateiverdoppelung findet also hier eine Adreßtabellenverdoppelung statt; die Datei wächst nur um einzelne Blöcke (Verfahren VH0 in [110, 111]). Für das in Abbildung 4.10 gezeigte Beispiel ergibt sich dann die in Abbildung 4.11 dargestellte Situation.

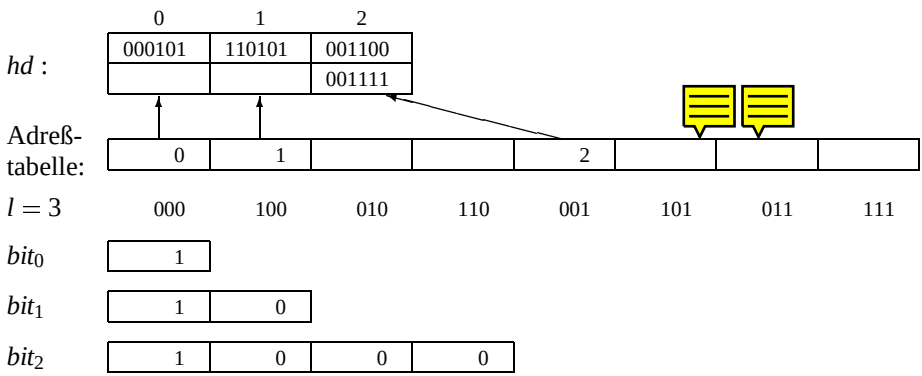


Abbildung 4.11

Man kann zeigen, daß die mittlere Speicherplatzausnutzung der Hashdatei hier um den Mittelwert $\ln 2 \approx 0.69$ pendelt. Soll auch hier noch mit einem einzigen Externspeicherzugriff ein Datensatz wiedergefunden werden können, so muß die Adreßtabelle neben den Bittabellen im Hauptspeicher Platz finden. Weil die Adreßtabelle Platz für mehr Einträge vorsieht als alle Bittabellen zusammen, und weil ein Eintrag der Adreßtabelle mehr Platz benötigt als ein Bit, kann dies unter Umständen eine unrealistische Annahme sein. Möglicherweise muß man dann die Adreßtabelle (und vielleicht sogar die Bittabellen) auf dem Externspeicher verwalten; dann können für das Wiederfinden eines Datensatzes zwei oder mehr Externzugriffe nötig werden. Im nächsten Abschnitt werden wir ein Verfahren vorstellen, bei dem man einen Schlüssel stets mit höchstens zwei Externzugriffen wiederfindet.

4.4.3 Erweiterbares Hashing

Erweiterbares Hashing (vorgestellt in [51], mit Ordnungserhaltung in [178]) hat eine starke Ähnlichkeit mit virtuellem Hashing mit Adreßtabelle. Wie dort wird bei erweiterbarem Hashing die Adreßtabelle bei Bedarf verdoppelt. Dieser Bedarf tritt ein, wenn

durch das Einfügen eines Datensatzes ein Datenblock geteilt werden muß und die beiden Adressen der beiden resultierenden Datenblöcke nicht in der bereits vorhandenen Adreßtabelle zu speichern sind. Während bei virtuellem Hashing mit Adreßtabelle die Adresse eines Datenblocks nur einmal in der Adreßtabelle auftritt und die unbenutzten Adreßtabellenfelder über die Bittabellen erkennbar sind, wird bei erweiterbarem Hashing jedes Adreßtabellenfeld benutzt. Damit spart man sich die Bittabellen; die bisher nicht benutzten Adreßtabelleneinträge müssen jetzt sinnvoll angegeben werden. Das ist aber leicht möglich, weil es wenigstens einen Adreßtabelleneintrag für jeden Datenblock gibt und damit die Adreßtabelle Schlüssel nach den ersten l Bits wenigstens so fein unterscheidet wie für die Verteilung auf Datenblöcke erforderlich. So gibt es beispielsweise in der in Abbildung 4.11 dargestellten Situation nur einen mit Bit 1 beginnenden Schlüssel (nämlich 110101), aber vier mit Bit 1 beginnende Nummern von Adreßtabelleneinträgen (nämlich 100, 110, 101 und 111). Wir können also einfach mit allen vier Adreßtabelleneinträgen auf denselben Datenblock verweisen, wie in Abbildung 4.12 gezeigt. Eintrag * in der Adreßtabelle repräsentiert eine fiktive Adresse, nämlich die eines leeren Datenblocks, den wir nicht explizit speichern.

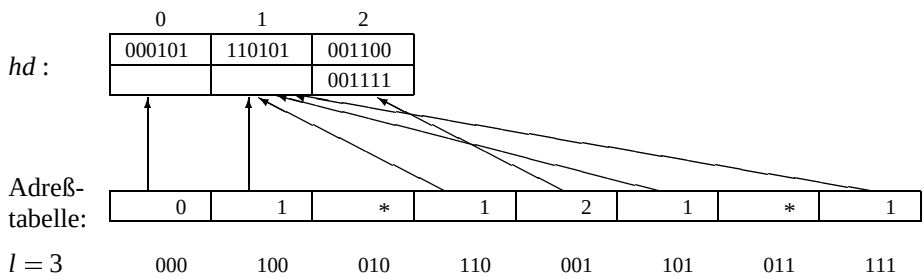


Abbildung 4.12

Betrachten wir zum genaueren Verständnis wieder das sukzessive Einfügen der Schlüssel 12, 53, 5 und 15, so ergibt sich nach Einfügen der ersten drei dieser Schlüssel die in Abbildung 4.13 gezeigte Situation.

Das Einfügen von Schlüssel 15 führt zu einem Split des Datenblocks 0. Dazu wird zunächst die Adreßtabelle verdoppelt und der Adreßtabellenlevel, also die Anzahl der zur Bestimmung der Nummer eines Adreßtabelleneintrags herangezogenen Bits, um 1 erhöht, wie in Abbildung 4.14 gezeigt.

Die Verdoppelung der Adreßtabelle ist das Anhängen einer identischen Kopie der bisherigen Adreßtabelle an sich selbst. Dann wird der überlaufende Datenblock gesplittet, indem ein neuer Block kreiert wird und der Inhalt des überlaufenden Blocks und der einzufügende Eintrag verteilt werden. In der gezeigten Situation ist jedoch ein Split des Datenblocks 0 nicht erfolgreich: Beide gespeicherten Einträge und der einzufügende Eintrag beginnen mit Bits 00, lassen sich also in den ersten $l = 2$ Bits nicht unterscheiden. Der neu angelegte Block, auf den der Adreßtabelleneintrag 01 verweist, bleibt leer. In diesem Fall wollen wir uns, in einer kleinen Modifikation des Vorschlags in [51], das

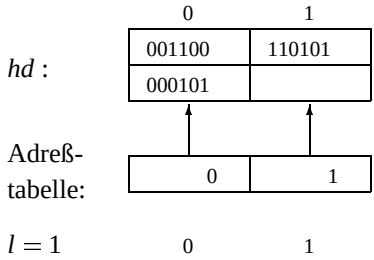


Abbildung 4.13

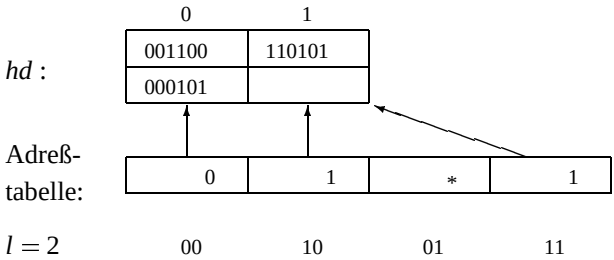


Abbildung 4.14

explizite Speichern eines leeren Blocks sparen und stattdessen den Adreßverweis als Verweis auf einen leeren Block kenntlich machen. Die Adreßverweise für verschiedene leere Blöcke werden verschieden gewählt. Dann wird eine weitere Adreßtabellenverdoppelung durchgeführt, die mit der in Abbildung 4.15 gezeigten Situation endet.

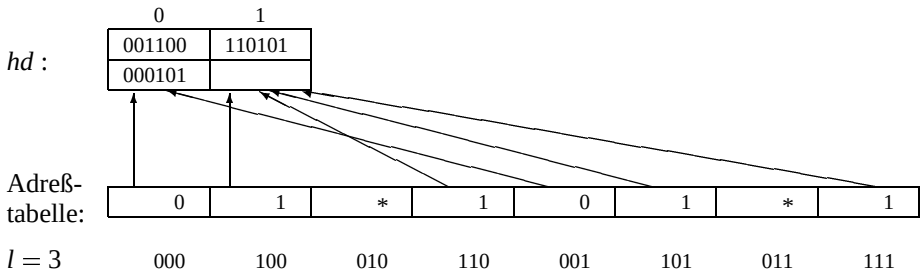


Abbildung 4.15

Weil die drei fraglichen Schlüssel nicht auch noch im dritten Bit übereinstimmen, ist ein Split des Datenblocks 0 jetzt erfolgreich. Block 0 wird in Blöcke 0 und 2 (die

nächste freie Datenblockadresse) aufgeteilt. Die beiden vor der Aufteilung auf Block 0 verweisenden Adreßtabelleneinträge werden gemäß dem dritten Bit angepaßt, wie in Abbildung 4.12 gezeigt.

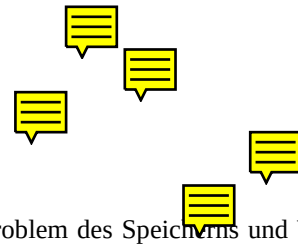
Unter der Annahme, daß nicht nur die Datenblöcke, sondern auch die Adreßtabelle auf Externspeicher verwaltet werden, kommt man bei der Suche nach einem Datensatz mit gegebenem Schlüssel bei erweiterbarem Hashing stets mit höchstens zwei Externzugriffen aus (das *Zwei-Zugriffs-Prinzip* des erweiterbaren Hashing): Für Level l der Adreßtabelle wird zunächst gemäß den ersten l Bits des Schlüssels auf einen Adreßtabelleneintrag zugegriffen. Wird dort auf einen leeren Block verwiesen, so endet die Suche erfolglos; sonst wird der dort referenzierte Datenblock gelesen und inspiziert. Um beim Versuch des Einfügens in einen vollen Datenblock ohne weitere Externzugriffe entscheiden zu können, ob die Adreßtabelle verdoppelt werden muß, merken wir uns neben dem Level der Adreßtabelle (auch *globale Tiefe* genannt) für jeden Datenblock einen Level, die *lokale Tiefe*. Die lokale Tiefe eines Datenblocks i ist die Länge des kürzesten Anfangsstücks eines Schlüssels, das die Schlüssel im Block i von allen anderen unterscheidet. Beispielsweise haben in der in Abbildung 4.15 gezeigten Situation beide Datenblöcke die lokale Tiefe 1; in der Situation in Abbildung 4.12 dagegen haben Blöcke 0 und 2 die lokale Tiefe 3. Die Schlüssel aller Sätze in einem Block mit lokaler Tiefe t stimmen also mindestens in den ersten t Bits überein, und alle Sätze mit solchen Schlüsseln befinden sich in diesem Block. Auf einen Block mit lokaler Tiefe t verweisen bei globaler Tiefe l genau 2^{l-t} Einträge der Adreßtabelle. Das sind natürlich genau diejenigen Einträge, deren Hashadressen (relative Nummern in der Adreßtabelle) in den ersten t Bits mit den Schlüsseln im Block übereinstimmen.

Beim Einfügen eines Satzes in einen Block wird zunächst durch eine Suche der Block identifiziert, in den der Satz einzufügen ist. Verweist der durch die ersten l Bits des Schlüssels identifizierte Adreßtabelleneintrag auf einen leeren Block, so wird ein Block erzeugt und der einzufügende Schlüssel dort eingetragen. Verweist dagegen der Adreßtabelleneintrag auf einen nicht leeren und nicht vollen Datenblock, so wird der einzufügende Schlüssel dort eingetragen. Interessant ist also der Fall, daß ein Datensatz in einen bereits vollen Block eingefügt werden müßte. In diesem Fall wird der betreffende Block zunächst gesplittet. Damit dies gelingen kann, muß wenigstens ein weiteres Bit der Schlüssel als Unterscheidungsmerkmal verwendet werden. Aus dem Block mit lokaler Tiefe t vor dem Split werden zwei Blöcke mit jeweils lokaler Tiefe $t+1$. Falls vor dem Split bereits $t = l$ gilt, muß zunächst die globale Tiefe l erhöht werden. Hierzu wird die Größe der Adreßtabelle verdoppelt. Wie wir bereits in unserem Beispiel gesehen haben, muß der Split eines Blocks nicht notwendigerweise dazu führen, daß der einzufügende Schlüssel auch gespeichert werden kann. In diesem Fall unterscheiden sich die Schlüssel der $b+1$ zu speichernden Sätze in den ersten $t+1$ Bits nicht. Dann werden Blocksplit und womöglich sogar Adreßtabellenverdoppelung wiederholt durchgeführt, bis schließlich eine Aufteilung gelingt.

Das Verfahren zum Entfernen eines Datensatzes ist auch bei erweiterbarem Hashing gerade die Umkehrung des Einfügens. Zunächst wird der zu entfernende Datensatz aus dem entsprechenden Block gelöscht. Dann wird überprüft, ob ein Blocksplit rückgängig gemacht werden kann, indem zwei Blöcke zu einem verschmolzen werden. Zwei Blöcke können dann verschmolzen werden, wenn die dort gespeicherten Sätze gemeinsam in einen Block passen und die Blöcke durch einen Split aus einem Block entstehen können — solche Blöcke heißen auch *Brüder*. Zwei Blöcke sind Brüder, wenn sie die

gleiche lokale Tiefe t haben und die Schlüssel aller in beiden Blöcken gespeicherten Datensätze in den ersten $t-1$ Bits übereinstimmen. Dann stimmen die Hashadressen von Verweisen auf diese Blöcke in der Adreßtabelle ebenfalls in den ersten $t-1$ Bits überein. Das Verschmelzen geschieht dann durch Zusammenlegen der Sätze auf einen der beiden Brüder und das Anpassen der Adreßtabelleneinträge. Wenn nach einer Verschmelzung für jeden Block die lokale Tiefe echt kleiner ist als die globale Tiefe der Adreßtabelle, so verweisen auf jeden Datenblock mindestens zwei Einträge der Adreßtabelle, und die Adreßtabelle kann halbiert werden. Diese Operation ist völlig symmetrisch zur Verdoppelung der Adreßtabelle.

Ein wichtiges Argument für den Einsatz von erweiterbarem Hashing für die Organisation externer Dateien ist neben der garantierten Effizienz der Suchoperation und der im Mittel akzeptablen Effizienz des Einfügens und Entfernens eine gute Speicherplatzausnutzung. Bei gleichverteilten Schlüsseln ergibt sich nach dem zufälligen Einfügen von n Datensätzen in die anfangs leere Hashdatei eine mittlere Anzahl von $(n/b)\ln 2$ Blöcken. Damit sind Blöcke durchschnittlich zu etwa 69 % belegt, wie dies auch für viele andere Strukturen gilt, die mit rekursivem Halbieren arbeiten [51, 100, 124]. Im Unterschied dazu wächst die Größe der Adreßtabelle überlinear in n, mit $O((1/b)n^{1+1/b})$ [197]. Die Blockkapazität b spielt offensichtlich auch hier eine gewichtige Rolle. Eine genauere, aber kompliziertere Analyse der Größe der Adreßtabelle findet man in [53].



4.5 Das Gridfile

In den vorangehenden Abschnitten haben wir das Problem des Speicherns und Wiederfindens von Schlüsseln mit genau einer Komponente, sogenannte *eindimensionale* Schlüssel, betrachtet. Bei vielen Datenverwaltungsproblemen hat man es aber mit *mehrdimensionalen* Schlüsseln, also Schlüsseln mit mehreren Komponenten, zu tun. So kann man beispielsweise einen Eintrag in einem Telefonbuch als aus zwei Komponenten bestehend ansehen: Die erste Komponente ist der Teilnehmername samt Adresse, die zweite Komponente die Telefonnummer. Organisiert man nun die Einträge in einer Datenstruktur gemäß der ersten Komponente, so wird die Suche nach Datensätzen mit gegebener zweiter Komponente im allgemeinen nicht unterstützt. Beispielsweise ist es nicht leicht, im Telefonbuch einen Teilnehmer mit gegebener Telefonnummer zu finden. Mehrdimensionale Hashverfahren versuchen hier, Abhilfe zu schaffen, indem mehrdimensionale Schlüssel so verwaltet werden, daß die Suche nach Datensätzen mit einigen vorgegebenen Schlüsselkomponentenwerten für alle Komponenten gleich gut unterstützt wird. Außerdem sollen natürlich das Einfügen und Entfernen von Datensätzen effizient möglich sein. Da es sich bei mehrdimensionalen Schlüsseln manchmal um geometrische Daten handelt, wie etwa Koordinaten von Punkten in der Ebene, ist es darüber hinaus wünschenswert, räumlich orientierte Anfragen zu unterstützen. So möchte man beispielsweise einen rechteckigen Ausschnitt aus einer Landkarte (mit Städten als Punkten) auf dem Bildschirm anzeigen. Um diese Punkte zu finden, führt man eine Bereichsanfrage aus. Bei der *Bereichsanfrage* fragt man nach allen Schlüsseln, deren sämtliche Komponenten in einen jeweils vorgegebenen Bereich (ein Schlüsselintervall)

fallen. Ist ein Schlüssel ein Paar kartesischer Koordinaten der Ebene, so ist der Bereich einer Bereichsanfrage ein achsenparalleles Rechteck. Auch im eindimensionalen Fall spielt die räumliche Nähe von Schlüsseln bereits eine gewisse Rolle, nämlich bei ordnungserhaltenden dynamischen Hashverfahren. Die mehrdimensionale Bereichsanfrage kann man als Verallgemeinerung der Suche nach einem Schlüssel mit anschließendem sequentiellen Inspizieren der benachbarten Schlüssel ansehen, wie sie durch Ordnungserhaltung unterstützt wird.

Sei d die Dimension der Schlüssel, und sei K_i das Universum der i-ten Schlüsselkomponente, $1 \leq i \leq d$. Dann ist $K = K_1 \times K_2 \times \ldots \times K_d$ das Universum aller möglichen d-dimensionalen Schlüssel. Für die Menge der Dimensionen $D = \{1, \ldots, d\}$ und $I \subseteq D$ betrachten wir genauer die folgenden Operationen:

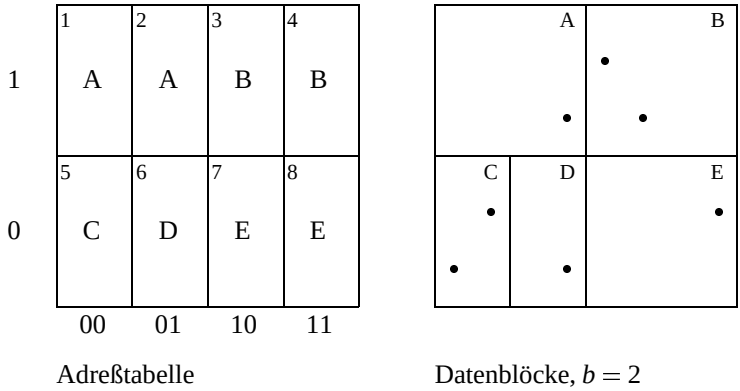
- *Suchen* nach Schlüssel $k = (k_1, \ldots, k_d) \in K$ mit vorgegebenem k_i für $i \in I$;
- *Bereichsanfrage* nach allen Schlüsseln $k = (k_1, \ldots, k_d) \in K$ mit $k_i^u \leq k_i \leq k_i^o$ für alle $i \in I$, wobei k_i^u die untere und k_i^o die obere Bereichsgrenze in Dimension i ist;
- *Einfügen* eines Schlüssels k;
- *Entfernen* eines Schlüssels k.

Falls $I \subset D$, so sprechen wir von *partieller Suche* (*partial match query*) und *partieller Bereichsanfrage* (*partial range query*).

Bei mehrdimensionalen Hashverfahren versucht man nun, in Verallgemeinerung von eindimensionalen Verfahren den mehrdimensionalen Raum in mehrdimensionale Rechtecke einzuteilen, die gerade das Produkt eindimensionaler Intervalle sind. Für die einzelnen Dimensionen versucht man dann, übliche eindimensionale dynamische Hashverfahren zu verwenden. Jedem Teilraum des Datenraums wird genau ein Datenblock zugeordnet, wie wir dies schon von dynamischen Hashverfahren kennen; derselbe Block kann mehreren Teilräumen zugeordnet sein. Zur klaren Unterscheidung nennen wir einen einzelnen Teilraum *Gitterzelle*; die Vereinigung der Gitterzellen, denen derselbe Block zugeordnet ist, heißt *Blockregion*. Wir beschränken uns in diesem Abschnitt wegen der einfacheren Darstellung auf zweidimensionale Daten; die Verallgemeinerung für höhere Dimensionen sollte klar sein. Mit der Einteilung des Datenraums in rechteckige Gitterzellen erreicht man, daß alle in einem Block gespeicherten Punkte räumlich dicht beieinander liegen, eine günstige Voraussetzung für Bereichsanfragen.

Betrachten wir zunächst das in Abbildung 4.16 gezeigte Beispiel mit zweidimensionalen Schlüsseln, die als Punkte in der Ebene gezeichnet sind und mit zweidimensionalem erweiterbarem Hashing mit ordnungserhaltender Hashfunktion verwaltet werden. Wegen der besseren geometrischen Zuordnung geben wir die Hashadressen in Abbildung 4.16 in jeder Dimension in aufsteigender Sortierung an; die Speicherung der Adreßtabelle bleibt davon unberührt.

Wir verwalten also eine zweidimensionale Adreßtabelle, deren Spalten mit der einen und deren Zeilen mit der anderen eindimensionalen Hashfunktion adressiert werden, gemäß erweiterbarem Hashing (EXCELL in [179]). Ein Eintrag in der Adreßtabelle ist die Adresse desjenigen Datenblocks, in dem die Punkte der entsprechenden Gitterzelle gespeichert sind. Bei einer Blockkapazität b von zwei Datensätzen können wir die in



Adreßtabelle Datenblöcke, $b = 2$

Abbildung 4.16

Abbildung 4.16 gezeigte Aufteilung der Daten auf fünf Blöcke wählen. Als Folge der feinen Unterteilung des Datenraums in Teilräume durch die bei erweiterbarem Hashing gewählte Adreßtabelle gibt es auch in unserem Beispiel Blockregionen, die durch Vereinigung mehrerer Gitterzellen entstehen. In diesen Fällen gibt es mehrere Verweise von der Adreßtabelle auf den entsprechenden Datenblock. In unserem Beispiel ist dies so für die Datenblöcke *A*, *B* und *E*. Der Nachteil mehrerer Verweise auf Datenblöcke ist im mehrdimensionalen Fall aber leichter behebbar als im eindimensionalen: Schon wenige Hashadressen genügen, um viele Adreßtabelleneinträge zu verwalten, weil die Anzahl der Adreßtabelleneinträge das Produkt der Anzahlen der Hashadressen in den verschiedenen Dimensionen ist, und nicht — wie es im eindimensionalen der Fall wäre — deren Summe. Damit wird es attraktiv, die Hashadressen in allen Dimensionen explizit zu verwalten; für realistische Anwendungsfälle kann dies leicht im Hauptspeicher geschehen. Somit entfällt die Notwendigkeit zur Adreßtabellenverdoppelung, und damit läßt sich die Adreßtabelle zur Situation von Abbildung 4.16 wie in Abbildung 4.17 gezeigt angeben. In [158] findet man eine Analyse der Größe der Adreßtabelle für beide Verfahren.

Ein sehr bekanntes und bewährtes mehrdimensionales Hashverfahren, das man als mehrdimensionales erweiterbares Hashing mit den angegebenen Modifikationen ansehen kann, ist das *Gridfile* [129], das wir im folgenden genauer erläutern werden. Die Einteilung des Datenraums für jede Dimension geben wir hierbei in Koordinatenwerten statt in führenden Bits von Schlüsseln an, weil damit der geometrische Bezug einfacher erkennbar ist. Die Einteilung des Datenraums für jede Dimension heißt *Scale*; die Adreßtabelle heißt *Directory*. Die Scales werden im Hauptspeicher verwaltet, die Directory-Matrix wird dagegen extern gespeichert. Dies geschieht mit dem Ziel der Zwei-Zugriffs-Garantie für die exakte Suche, wie bei erweiterbarem Hashing. Überdies erreicht man beim Gridfile, daß die partielle Suche für jede spezifizierte Schlüsselkomponente gleichermaßen effizient ist.

Wir werden ein Gridfile im folgenden kompakter graphisch darstellen, indem wir die Aufteilung des Datenraums in Gitterzellen gemäß der Adreßtabelle und die Aufteilung

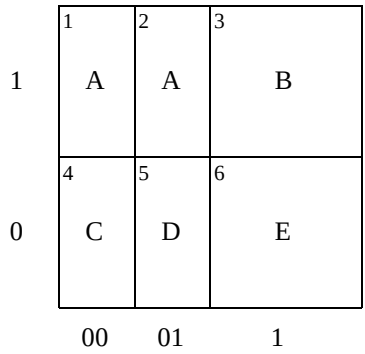


Abbildung 4.17

des Datenraums in Blockregionen übereinander zeichnen. Gestrichelte Linien trennen dabei Gitterzellen, die zur selben Blockregion gehören; durchgezogene Linien trennen Regionen. Außerdem vermerken wir die Scales in jeder Dimension, die Directoryadressen und die Datenblockadressen (siehe Abbildung 4.18).

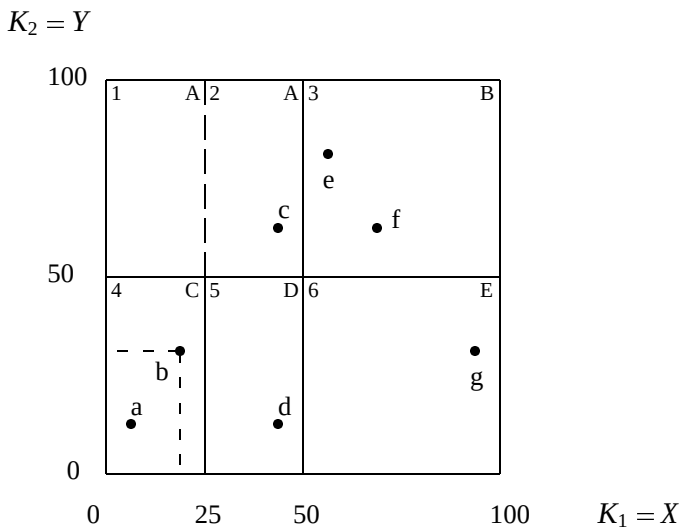


Abbildung 4.18

Bezeichnen wir für $d = 2$ K_1 mit X, K_2 mit Y, k_1 mit x und k_2 mit y, so kann die *exakte Suche* nach $k = (x,y)$ wie folgt durchgeführt werden:

1. Bestimme anhand der X-Scales die Spalte s der Directory-Matrix, in die x fällt; bestimme anhand der Y-Scales die Zeile z der Directory-Matrix, in die y fällt.
2. Berechne die Externspeicheradresse a_1 des Directory-Elements in Zeile z und Spalte s.
3. Lies den Directory-Block *dir* mit Adresse a_1 in den Hauptspeicher.
4. Bestimme die Externspeicheradresse a_2 des Datenblocks zu derjenigen Gitterzelle in *dir*, in die (x,y) fällt.
5. Lies den Datenblock *dat* mit Adresse a_2 in den Hauptspeicher.
6. Durchsuche *dat* nach (x,y) und berichte das Ergebnis.

In dem in Abbildung 4.18 gezeigten Beispiel führt die Suche nach Punkt $b = (20,38)$ zur Bestimmung der zweiten Zeile (von oben) und der ersten Spalte (von links) der Directory-Matrix und damit zum Directory-Element mit Adresse 4. Dieses enthält den Verweis auf Datenblock C, in dem die Punkte a und b gespeichert sind. Die Suche nach b endet also erfolgreich.

Lediglich in Schritten 3 und 5 des Algorithmus zur exakten Suche findet je ein Externzugriff statt; die exakte Suche benötigt also stets genau zwei Zugriffe, wenn die Scales im Hauptspeicher verwaltet werden — das *Zwei-Zugriffs-Prinzip* des Gridfiles.

Damit sollte auch klar sein, wie die partielle Suche und die Bereichsanfrage beantwortet werden können. Bei der Bereichsanfrage etwa sucht man zunächst nach dem linken unteren Punkt des rechteckigen Anfragebereichs und überprüft für alle Punkte im gefundenen Datenblock, ob sie im Anfragebereich liegen. Dann setzt man die Suche nach rechts und nach oben über benachbarte Zeilen und Spalten und daraus berechenbare Directoryelemente fort. Das bedeutet, daß auch auf der Directory-Matrix eine Bereichsanfrage durchgeführt wird: Gesucht sind alle Gitterzellen, die den Anfragebereich schneiden. Als Folge davon muß die Directory-Matrix, die ja wegen ihrer Größe im allgemeinen auch auf Externspeicher verwaltet wird, dieselben Operationen unterstützen wie die Datenstruktur für die ursprünglich gegebenen Datenpunkte. Es ist also vernünftig, Gitterzellen ebenso wie Datenpunkte in einem Gridfile zu organisieren. Dies führt zum Mehr-Ebenen-Gridfile [94], das für die meisten realen Anwendungsfälle mit nur zwei Ebenen auskommt, wenn ein großes Wurzel-Directory im Hauptspeicher gehalten werden kann [78].

Nehmen wir für das Beispiel der in Abbildung 4.16 gezeigten Datenpunkte an, daß jeder Datenblock $b = 2$ Punkte, jeder Directory-Block $b' = 2$ Adressen von Datenblöcken und das Wurzel-Directory $b'' = 4$ Adressen von Directoryblöcken speichern kann. Dann ergibt sich für die gezeigten Datenblöcke A,B,C,D und E das in Abbildung 4.19 gezeigte 2-Ebenen-Directory mit Directoryblöcken A',B' und C' und einem Wurzeldirectory. Eine Bereichsanfrage mit dem Anfragebereich $[40\,..\,60] \times [40\,..\,60]$ führt in der gezeigten Situation im Wurzeldirectory auf die Directoryblockadressen A',B' und C', und für diese Directoryblöcke auf die Datenblockadressen A,B,D,E. Die Effizienz einer Bereichsanfrage ist also nach unten beschränkt durch die Effizienz der exakten

Suche; mit größer werdenden Anfragebereichen steigt in der Tendenz auch die Anzahl der als Antwort gefundenen Datensätze und die Anzahl der benötigten Externzugriffe. Die Effizienz von Bereichsanfragen mit großen Anfragebereichen ist eng an die Speicherplatzausnutzung gekoppelt, weil Externzugriffe, die wenig zur Antwort beitragen, nur für Gitterzellen und Datenblockregionen am Rand des Anfragebereiches ausgeführt werden müssen. Eine genaue Analyse [54] ergibt, daß im Mittel $O(n^{1-|I|/d})$ Externzugriffe für die partielle Suche nach $|I|$ von d Schlüsseln in einem Gridfile mit n Datensätzen ausreichen. Diese Effizienz wird für optimal gehalten [160]. Dabei ist es natürlich stets wichtig, daß sich das Gridfile an dynamisch veränderliche Datenmengen anpaßt. Wir werden im folgenden genauer betrachten, wie dies bei Einfüge- und Entferneoperationen geschieht.

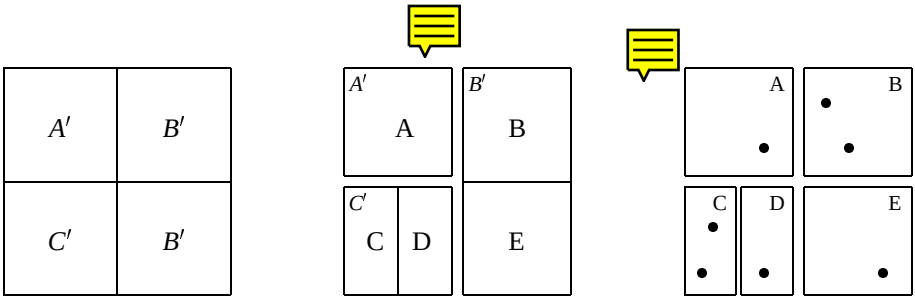


Abbildung 4.19

Beim *Einfügen* eines Datensatzes wird zunächst durch eine exakte Suche der Datenblock ermittelt, in den der Datensatz einzufügen ist. Sofern der Datensatz in diesem Block noch Platz findet, wird er dort eingefügt, der Block auf den Externspeicher zurückgeschrieben, und die Einfügeoperation ist beendet. Andernfalls muß ein neuer Datenblock kreiert werden. Zu diesem Zweck wird der fragliche Block in zwei Blöcke geteilt, indem seine Region entlang einer Koordinatenachse in der Mitte *zerschnitten* (*gesplittet*, englisch: *split*) wird — ein *Datenblocksplit*. Die Datensätze werden gemäß der beiden neuen Datenblockregionen auf die beiden neuen Datenblöcke aufgeteilt. Die neue Situation muß im Directory vermerkt werden. Weil das Directory (als Ganzes beim Ein-Ebenen-Directory und als lokaler Directoryblock im Mehr-Ebenen-Directory) als Matrix organisiert bleiben muß, durchtrennt die Splitlinie in allen von der Splitdimension verschiedenen Dimensionen den gesamten (zum Directoryblock lokalen) Datenraum. Wie schon bei erweiterbarem Hashing kann hier natürlich der Fall auftreten, daß ein Blocksplit nicht zum wirklichen Verteilen von Datensätzen führt, daß einer der neugeschaffenen Blöcke also leer bleibt; in diesem Fall wird der Blocksplit rekursiv für den noch immer übervollen Block fortgesetzt. Somit ist nur noch die Wahl der Splitdimension bei einem Blocksplit offen. Betrachten wir dazu das in Abbildung 4.18 gezeigte Beispiel und nehmen wir an, daß ein Directoryblock $b' = 6$ Datenblockadressen verwalten kann; Abbildung 4.18 zeigt gerade einen Directoryblock mit Verweisen auf die Datenblöcke A, B, C, D und E. Im folgenden geben wir drei Regeln an, von denen die

erste in dieser Reihenfolge angewendet wird, die eine eindeutige Splitentscheidung liefert:

(1) Teile die längste Seite einer Datenblockregion.
Soll im Beispiel der Abbildung 4.18 Datenblockregion C geteilt werden, so findet ein *waagerechter Split* statt, also eine Aufteilung der Region $[0\,..\,25] \times [0\,..\,50]$ in Regionen $[0\,..\,25] \times [0\,..\,25]$ und $[0\,..\,25] \times [25\,..\,50]$. Wegen der Matrixeigenschaft des Directoryblocks sind damit zwei Verweise auf Datenblock D und zwei Verweise auf Datenblock E erforderlich; die Anzahl der Verweise kann sich durch einen Split also mehr als eigentlich nötig erhöhen.

(2) Teile eine Datenblockregion gemäß einer vorhandenen Einteilung in Gitterzellen.
Soll im Beispiel der Abbildung 4.18 die Datenblockregion A geteilt werden, so erfolgt ein vertikaler Split, weil Regel 1 keine eindeutige Entscheidung liefert und gemäß Regel 2 die bereits vorhandene vertikale Splitlinie verwendet werden muß. Im Directory wird lediglich ein Teil der Verweise geändert; die Gitterzelleneinteilung ändert sich nicht.

(3) Teile eine Datenblockregion in derjenigen Dimension, in der die kleinste Anzahl von Teilungen vermerkt ist.
Im Beispiel der Abbildung 4.18 hat demnach ein Split der Datenblockregion B in waagerechter Richtung zu erfolgen.

Liefert keine dieser Regeln eine eindeutige Entscheidung, so wird die Blockregion entlang einer beliebigen Dimension geteilt, etwa abwechselnd nach X und Y. Die vorgestellte Splitstrategie präferiert keine der Dimensionen vor einer anderen, führt also in der Tendenz zu Blockregionen, deren Verhältnis von Länge zu Breite möglichst nahe bei 1 liegt. Im Unterschied zu directorylosen Strukturen ist es beim Gridfile (wie schon bei erweiterbarem Hashing) nicht erforderlich, leere Datenblöcke explizit zu speichern. Statt dessen genügt es, entsprechend markierte Verweise im Directory zu verwalten.

Die Teilung eines Datenblocks führt im entsprechenden Directoryblock im allgemeinen zur Erhöhung der Anzahl der zu verwaltenden Verweise. Läuft der Directoryblock über, so wird auch dieser geteilt. Man kann hier im wesentlichen dieselben Regeln verwenden wie beim Teilen einer Datenblockregion. Beim Teilen einer Directoryblockregion muß die Einteilung der beiden resultierenden Blöcke in Gitterzellen überprüft werden, weil diese als Folge der Teilung günstiger werden kann. Betrachten wir dazu als Beispiel Abbildung 4.18 mit einer Directoryblockkapazität von $b' = 5$ und Datenblockkapaziät $b = 2$ und nehmen wir an, daß der gezeigte Directoryblock soeben durch Einfügen des Punktes b und damit durch Einziehen der Splitlinie $x = 25$ mit der Verfeinerung der Einteilung von vier auf sechs Gitterzellen übervoll geworden ist. Teilen wir nun die Directoryblockregion (willkürlich) waagerecht, so entfällt die Notwendigkeit, Datenblockregion A in zwei Gitterzellen aufzuteilen; wir kommen also mit fünf Verweisen auf die fünf Datenblöcke aus, die allerdings nicht in einem Directoryblock untergebracht werden können (vgl. Abbildung 4.20). Denselben Effekt können wir bereits in Abbildung 4.19 gegenüber Abbildung 4.18 beobachten.

Das *Löschen* eines Datensatzes aus einem Gridfile wird realisiert durch eine exakte Suche nach dem zu löschenden Datensatz, gefolgt vom anschließenden Entfernen des Datensatzes im entsprechenden Datenblock und Zurückschreiben dieses Blocks. Im

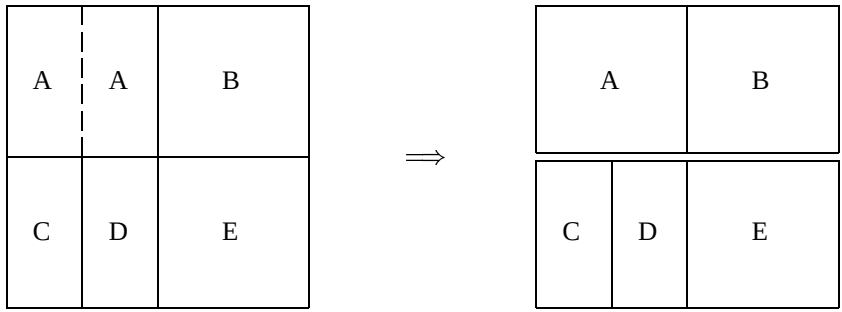


Abbildung 4.20

Unterschied zum Einfügen sind nach dem Löschen keine weiteren Aktionen zwingend erforderlich; im Interesse einer guten Speicherplatzausnutzung, die ja auch für die Effizienz von Anfragen wichtig ist, sind solche Aktionen dennoch geboten. Symmetrisch zum Aufteilen (Split) einer Region bei einem Blocküberlauf nach einer Einfügeoperation kann man nach einer Entferneoperation zwei Blöcke *verschmelzen* (englisch: *merge*), um die Speicherplatzausnutzung nicht unter ein gewisses Mindestmaß absinken zu lassen. Damit sich in einem dynamischen Anwendungsfall, mit weiteren noch zu erwartenden Einfüge- und Entferneoperationen, nicht ständig Teile- und Verschmelzeoperationen abwechseln, wird eine Verschmelzeoperation nur nach schrittweiser Überprüfung zweier Bedingungen durchgeführt. Zunächst muß die Speicherplatzausnutzung für den Datenblock, aus dem ein Datensatz soeben gelöscht wurde, eine vorgegebene Schranke unterschreiten, damit eine Verschmelzeoperation überhaupt erwogen und die dafür notwendigen Externzugriffe ausgeführt werden. Liegt eine solche *Schranke für die Überprüfung der Verschmelzung* etwa bei 30 %, so ist einerseits sichergestellt, daß Verschmelzeoperationen nicht allzu häufig unternommen werden, und andererseits sind die Aussichten auf einen genügend schwach gefüllten Partnerblock für die Verschmelzung nicht allzu schlecht. Liegt die Füllung eines Datenblocks unterhalb dieser Schranke, so wird unter allen gemäß der Gitterzelleneinteilung und der Verschmelzestrategie möglichen Partnern für eine Verschmelzung derjenige mit der schwächsten Füllung ermittelt. Eine obere *Schranke für das Durchführen der Verschmelzung* — typischerweise bei etwa 70 % — gibt die höchste nach der Verschmelzung beider Blöcke akzeptable Speicherplatzausnutzung an, bei der die Verschmelzung noch durchgeführt wird.

Die *Verschmelzestrategie* legt fest, welche Regionen überhaupt als Partner für eine Verschmelzung in Frage kommen. Dabei wird stets gefordert, daß die durch die Verschmelzung entstehende Blockregion rechteckig ist. In dem in Abbildung 4.18 gezeigten Beispiel ist damit ein Verschmelzen der Blockregionen A und C nicht zulässig. Die *Nachbarstrategie* läßt nun alle Verschmelzungen zu, bei denen ein rechteckiger Bereich entsteht. So können etwa gemäß der Nachbarstrategie die Regionen D und E in Abbildung 4.18 verschmolzen werden; bei Datenblockkapazität $b = 2$ passen auch tatsächlich die Inhalte beider Blöcke zusammen in einen Block. Im Hinblick auf eine hohe Speicherplatzausnutzung scheint diese am wenigsten restriktive Verschmelzestrategie

ganz besonders günstig zu sein. Daß dies nicht unbedingt so ist, zeigt das Beispiel in Abbildung 4.21. Dort sieht man, daß nach der gezeigten und gemäß Nachbarstrategie zulässigen Verschmelzung von Block *A* mit Block *E*, *B* mit *C* und *D*, *F* mit *G*, *H* mit *L* und *I* mit *J* und *K* keine weitere Verschmelzung mehr möglich ist, selbst wenn fast alle Datensätze entfernt werden — eine *Verklemmung* (*deadlock*). Die Speicherplatzausnutzung kann also hier beliebig absinken. Da man dies auf alle Fälle vermeiden möchte, muß man bei Anwendung der Nachbarstrategie Verklemmungen durch entsprechende Prüfung beim Verschmelzen verhindern. Es sollte klar sein, daß dies nicht immer ganz einfach und effizient möglich ist.

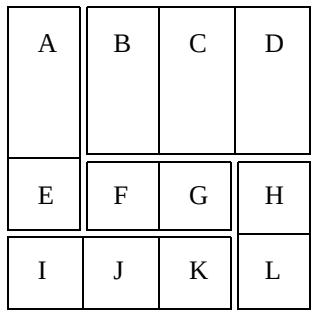


Abbildung 4.21

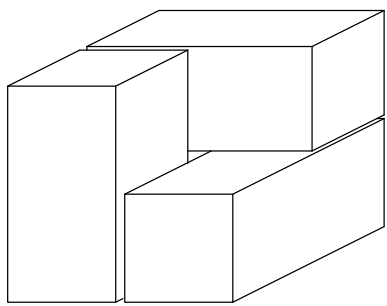


Abbildung 4.22

Die *Bruderstrategie* (*buddy merge*) erlaubt nur das Verschmelzen solcher Blöcke, die durch eine Teilung aus einem gemeinsamen Block hervorgegangen sein können. In diesem Fall macht eine Verschmelzung gerade eine Teilung rückgängig. Während eine Region höchstens einen Bruder in jeder Dimension hat, besitzt sie in jeder Dimension bis zu zwei Nachbarn; im zweidimensionalen Fall kann man also bei der Nachbarstrategie unter bis zu vier Partnern wählen, bei der Bruderstrategie aber höchstens unter

zweien. In dem in Abbildung 4.21 gezeigten Beispiel etwa hat Region G die beiden Brüder H und K und zusätzlich die beiden Nachbarn F und C. F ist kein Bruder von G, weil F und G nicht durch einen Split aus einer Region hervorgegangen sein können; die Gitterzellengrenze, die F von G trennt, muß zeitlich vor einer anderen G begrenzenden Linie eingeführt worden sein. Dagegen ist G entweder durch Abtrennen von H oder durch Abtrennen von K entstanden; jede dieser beiden Regionen kann als Partner beim Verschmelzen dienen. Die Bruderstrategie stellt im zweidimensionalen Fall sicher, daß keine Verklemmung auftritt; bereits im dreidimensionalen sind aber Verklemmungen möglich, wie Abbildung 4.22 zeigt.

Weil eine Region in jeder Dimension einen Bruder haben kann, ist es manchmal sinnvoll, unmittelbar nach der Aufteilung einer Region in zwei neue Regionen die Möglichkeit der Verschmelzung, gewissermaßen mit dem anderen Bruder, zu überprüfen. So führt etwa in dem in Abbildung 4.23 gezeigten Beispiel bei einer Datenblockkapazität von $b = 3$ Datensätzen das Einfügen des Datensatzes k zunächst zu einem Aufteilen des Blocks A auf die Blöcke A und D; bei einer oberen Schranke von 35 % für das Überprüfen und von 70 % für das Durchführen der Verschmelzung kann aber dann D mit C verschmolzen und somit die Speicherplatzausnutzung verbessert werden.

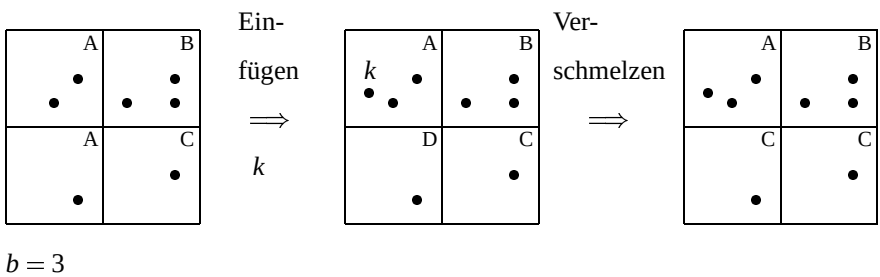


Abbildung 4.23

Das Verschmelzen von Directoryblöcken unterscheidet sich vom Verschmelzen von Datenblöcken durch die Notwendigkeit der Anpassung der Gitterzelleneinteilungen der beiden zu verschmelzenden Blöcke. Während beim Teilen von Directoryblöcken Splitlinien entfallen können, kann das Verschmelzen eine Verfeinerung der Einteilung bewirken. Zur Illustration dieses Phänomens können wir Abbildung 4.20 von rechts nach links lesen. Nehmen wir an, daß Blockregion B durch Verschmelzen zweier Blockregionen nach dem Entfernen eines Datensatzes entstanden ist, und daß die Schranken für das Prüfen und Durchführen einer Verschmelzung vorschreiben, die beiden rechts in Abbildung 4.20 dargestellten Directoryblöcke zu verschmelzen. Das Resultat der Verschmelzung ist der links in Abbildung 4.20 dargestellte Directoryblock, der aber nicht nur fünf, sondern sechs Regionen verwalten muß. Dieser Effekt muß vor der Durchführung der Verschmelzung zweier Directoryblöcke bedacht werden, weil sonst im Extremfall der resultierende Directoryblock bereits wieder übervoll sein kann (in unserem Beispiel wäre dies der Fall für Directoryblockkapazität $b' = 5$).

Eine Analyse des durchschnittlichen Verhaltens des Gridfiles hat sich als schwierig herausgestellt. Simulationen haben gezeigt, daß die durchschnittliche Auslastung von Datenblöcken in vielen Situationen bei etwa 70 % (ungefähr $\ln 2$) liegt, ein Wert, der sich für viele Strukturen ergibt, die mit rekursivem Halbieren arbeiten ([51], [53], [124]). Analytische Überlegungen zum Verhalten von Gridfiles findet man in [53] und [158]. Bei Datenblockkapazität b wächst das Directory des Gridfiles bei n gleichverteilten Datensätzen mit $O(n^{(1+1/b)})$, wie dies auch schon bei erweiterbarem Hashing der Fall war. Bei einer ungünstigen Verteilung der Datensätze, im zweidimensionalen Fall etwa entlang einer Diagonalen, wächst das Directory sogar mit $O(n^d)$ für ein d-dimensionales Gridfile. Trotz dieses relativ schlechten schlimmsten Falles ist das Gridfile eine für viele Anwendungen geeignete mehrdimensionale Datenstruktur.

4.6 Aufgaben

Aufgabe 4.1

Wieviele Schritte werden im schlechtesten Fall benötigt, um in eine anfangs leere Hashtabelle n Schlüssel einzufügen, wenn zur Überlaufbehandlung die Methode der separaten Verkettung mit unsortierten bzw. sortierten Listen verwendet wird? Wieviele Schritte benötigt man in diesen beiden Fällen, um nach jedem der n eingefügten Schlüssel einmal zu suchen?

Aufgabe 4.2

Zeigen Sie, daß die mittlere Anzahl von Hashtabellenplätzen, die bei einer erfolgreichen Suche (mit gleicher Wahrscheinlichkeit für alle Schlüssel) inspiziert werden, bei Hashing mit linearem Sondieren nicht von der Reihenfolge abhängt, in der die Schlüssel in die anfangs leere Hashtabelle eingefügt worden sind.

Gilt die entsprechende Aussage auch für quadratisches Sondieren?

Aufgabe 4.3

Geben Sie die Belegung einer Hashtabelle der Größe 13 an, wenn die Schlüssel

$$5, 1, 19, 23, 14, 17, 32, 30, 2$$

in die anfangs leere Tabelle eingefügt werden und offenes Hashing mit Hashfunktion $h(k) = k \bmod 13$ und

a) linearem Sondieren;

b) linearem Sondieren mit Sondierungsfunktion $s(j,k) = -j$;

c) quadratischem Sondieren

verwendet wird.

Vergleichen Sie die Anzahlen der beim Einfügen betrachteten Hashtabellenplätze für diese drei Sondierungsverfahren. Welche Kosten sind für eine erfolgreiche Suche zu erwarten, wenn nach jedem vorhandenen Schlüssel mit gleicher Wahrscheinlichkeit gesucht wird?

Aufgabe 4.4

Gegeben seien eine Hashtabelle der Größe 7 mit der Belegung

	0	1	2	3	4	5	6
t :	1	164	8	21	73	22	89

und die Hashfunktion $h(k) = (\text{Quersumme } (k)) \bmod 7$. Als Kollisionsstrategie wird quadratisches Sondieren angewandt.

a) Geben Sie alle Reihenfolgen an, in denen die Schlüssel in die anfangs leere Hashtabelle eingefügt worden sein können.

b) Gibt es eine andere Reihenfolge, die zu einer geringeren durchschnittlichen Anzahl zu inspizierender Hashtabellenplätze bei der erfolgreichen Suche führt, wenn die Suche nach jedem Schlüssel gleich wahrscheinlich ist?

Aufgabe 4.5

Gegeben sei eine anfangs leere Hashtabelle mit 13 Elementen, in die der Reihe nach die Schlüssel $14, 21, 27, 28, 8, 18, 15, 36, 5, 2$ mit Double Hashing eingefügt werden sollen. Die zu verwendenden Hashfunktionen seien $h(k) = k \bmod 13$ und $h'(k) = 1 + k \bmod 11$. Geben Sie die Belegung der Hashtabelle an, wenn die Schlüssel

a) in der gegebenen Reihenfolge;

b) in sortierter Reihenfolge;

c) in der gegebenen Reihenfolge mit Brents Algorithmus;

d) in sortierter Reihenfolge mit Brents Algorithmus;

e) in der gegebenen Reihenfolge mit Binärbaum-Sondieren;

f) in sortierter Reihenfolge mit Binärbaum-Sondieren;

g) in der gegebenen Reihenfolge mit Ordered Hashing;

h) in sortierter Reihenfolge mit Ordered Hashing

eingefügt werden.

Wieviele Hashtabellenplätze müssen beim Einfügen eines der Schlüssel, bei der erfolgreichen und bei der erfolglosen Suche jeweils höchstens inspiziert werden?

Aufgabe 4.6

a) Sind die beiden bei Double Hashing verwendeten Hashfunktionen $h(k) = k \bmod 7$ und $h'(k) = 1 + k \bmod 5$ unabhängig?

b) Ist $h'(k) = k^2 \bmod 7$ eine für h geeignete zweite Hashfunktion?

Aufgabe 4.7

Lösen Sie Aufgabe 4.5 für Robin-Hood-Hashing. Vergleichen Sie die erwartete Anzahl inspizierter Hashtabelleneinträge für die erfolgreiche Suche (bei gleicher Suchwahrscheinlichkeit für jeden Schlüssel) bei den Fällen a) bis h) der Aufgabe 4.5 mit Robin-Hood-Hashing mit dem Standard-Suchalgorithmus und mit smart searching. Dabei soll für smart searching als Erwartungswert der Länge von Sondierungsfolgen gerade deren Mittelwert für die gespeicherten Schlüssel verwendet werden.

Aufgabe 4.8

Lösen Sie Aufgabe 4.5 für Coalesced Hashing ohne Keller. Vergleichen Sie auch die Effizienz der erfolgreichen Suche (vgl. Aufgabe 4.7). Bei welcher Kellergröße ist im Beispiel die erfolgreiche Suche am schnellsten, wenn die Hashfunktion weiterhin nach der Divisions-Rest-Methode gewählt wird? Wie lang ist dann die längste Überlaufkette? Bei welcher Kellergröße ist im Beispiel die längste Überlaufkette am kürzesten, und wie schnell ist dann die erfolgreiche Suche?

Aufgabe 4.9

Verfolgen Sie die Entwicklung einer nach linearem Hashing organisierten Hashdatei mit Datenblockkapazität $b = 2$, wenn in die anfangs aus drei leeren Blöcken bestehende Datei die Schlüssel 5, 12, 43, 16, 19, 1990, 53 in dieser Reihenfolge eingefügt werden. Verwenden Sie dazu Hashfunktionen nach der Divisions-Rest-Methode und den Schwellenwert 0.8 für den Belegungsfaktor als Auslöser einer Block-Split-Operation.

a) Wieviele Blöcke werden für die ersten vier, wieviele für die ersten fünf und wieviele für alle sieben Schlüssel verwendet?

b) Kommt es im Verlauf des Einfügens vor, daß sich die Anzahl der im Mittel für die erfolgreiche Suche benötigten Externzugriffe verringert, obwohl sich die Anzahl gespeicherter Schlüssel erhöht hat? Welches ist der beste Wert, welches der schlechteste?

c) Gelangt man für die gegebene Schlüsselfolge zu einer besseren Speicherplatzausnutzung oder einer besseren mittleren Anzahl von Externzugriffen für die erfolgreiche Suche, wenn man mit einer anderen anfänglichen Dateigröße beginnt oder einen anderen Schwellenwert für den Belegungsfaktor wählt? Welches sind die besten Werte?

d) Wie ändert sich die Situation bei Verwendung einer ordnungserhaltenden Hashfunktion?

Aufgabe 4.10

Geben Sie eine genaue algorithmische Beschreibung für das Entfernen eines Datensatzes einschließlich des Verschmelzens von Blöcken an. Betrachten Sie die in der Aufgabenstellung der Aufgabe 4.9 beschriebene Situation, und verfolgen Sie die Entwicklung der Hashdatei, wenn alle Schlüssel in *derselben* Reihenfolge wieder entfernt werden, in der sie eingefügt wurden. Beantworten Sie die Fragen a) bis d) von Aufgabe 4.9 entsprechend.

Aufgabe 4.11

Betrachten Sie eine anfangs aus einem leeren Block bestehende Hashdatei mit Blockkapazität $b = 2$, die mit linearem Hashing organisiert ist, wobei die Hashfunktionen nach der Divisions-Rest-Methode gebildet werden und ein Block-Split stattfindet, wenn der Belegungsfaktor den Wert 1 erreicht. Geben Sie je eine Folge von n Schlüsseln an, die in der gegebenen Reihenfolge in die leere Hashdatei eingefügt werden, so daß

a) die mittlere Anzahl von Externzugriffen für die erfolgreiche Suche linear von n abhängt und sich nach jeder Dateiverdopplung für jeden Schlüssel die Hashadresse ändert;

b) zu keinem Zeitpunkt Überlaufblöcke erforderlich sind;

c) unter den während einer Expansion noch nicht gesplitteten Blöcken stets soviele Blöcke überlaufen, wie in dieser Expansion bereits gesplittet worden sind (aber höchstens alle noch nicht gesplitteten Blöcke).

Aufgabe 4.12

Geben Sie für virtuelles Hashing ohne und mit Adreßtabelle eine genaue algorithmische Beschreibung für das Entfernen eines Datensatzes einschließlich dem Verschmelzen von Blöcken an.

Aufgabe 4.13

Geben Sie für erweiterbares Hashing genaue algorithmische Beschreibungen an für Suchen, Einfügen und Entfernen von Datensätzen einschließlich Aufteilen und Verschmelzen von Blöcken und Verdoppeln und Halbieren der Adreßtabelle. Ein leerer Block kann explizit gespeichert, durch einen ihm eigenen Verweis dargestellt, oder durch einen für alle Blöcke gleichen Verweis dargestellt werden; wie unterscheiden sich die Algorithmen?

Aufgabe 4.14

In einem zweidimensionalen Gridfile reicht das Universum der ganzzahligen Schlüssel beider Dimensionen von 0 bis 20. Ein Datenblock kann höchstens vier Punkte, ein Directory-Block höchstens vier Verweise speichern. Beim Split wird eine Region im Zweifel senkrecht geteilt. Fügen Sie in das anfangs leere Gridfile die Punkte (4,6), (8,10), (18,4), (3,16), (14,18), (16,13), (11,2), (18,8), (12,9), (13,7), (20,7) und (16,2) ein.

a) Wieviele Externzugriffe verursacht die teuerste der Einfügeoperationen, wenn von einer Operation zur nächsten kein Block im Hauptspeicher gepuffert wird? Wie lautet die Anwort, wenn ein Directory-Block jeder Ebene und ein Datenblock gepuffert werden?

b) Wie hoch ist die Speicherplatzausnutzung von Datenblöcken, wie hoch die von Directory-Blöcken, in der nach dem Einfügen aller Punkte entstandenen Situation im Mittel und für den am besten und den am schlechtesten ausgenutzten Block?

c) Geben Sie eine Bereichsanfrage an, bei der die Anzahl gelesener Punkte, die nicht zur Antwort gehören, am höchsten ist. Wieviele Blöcke können dabei höchstens gelesen werden?

d) Geben Sie eine erfolgreiche und eine erfolglose partielle Suchanfrage an, bei der die Anzahl gelesener Blöcke am höchsten ist. Wieviele Punkte werden dabei höchstens gelesen, wieviele mindestens?

Aufgabe 4.15

Entwerfen Sie einen Algorithmus zur Beantwortung einer Anfrage nach einem nächsten Nachbarn (nearest neighbor, best match) eines gegebenen Anfragepunktes in einem zweidimensionalen Gridfile. Der nächste Nachbar eines Anfragepunktes ist derjenige Punkt in der betrachteten Menge, der zum Anfragepunkt die geringste Distanz hat.

Beziehen Sie neben der euklidischen Metrik (L_2) auch die Manhattan-Metrik (L_1) und die Maximums-Metrik (L_∞) in Ihre Überlegungen ein. Zur Erinnerung: Die Distanz d_i in Metrik L_i zwischen zwei Punkten (x,y) und (x',y') ist definiert als $d_i((x,y),(x',y')) = (|x-x'|^i + |y-y'|^i)^{1/i}$.

Aufgabe 4.16

Entwerfen Sie einen Algorithmus, der für ein zweidimensionales Gridfile mit Nachbar-Verschmelze-Strategie das Entstehen von Verklemmungen verhindert.