UNIVERSITÄT DES SAARLANDES
Fachbereich 14– Informatik
Prof. Dr.–Ing. G. Weikum
Dipl.–Inform. R. Schenkel

Datenbanksysteme WS 1999/2000

Übung 7 Beispiellösung zu den Aufgaben 1,2,3 und 5

Aufgabe 1: Zugriffsstrukturen

Gegeben sei die Musikdatenbank aus Übung 1. Die Schemadefinition, die angenommenen Kardinalitäten der einzelnen Relationen sowie die mittleren Tupellängen in Bytes sind in der folgenden Tabelle aufgeführt.

Tabelle	Schema		# Tupel	Mittlere Tupellänge (Bytes)
Disk	DiskID DiskTitel	NUMBER PRIMARY KEY NOT NULL VARCHAR(255)	1000	100
Musik-stueck	DiskID StueckID Titel	NUMBER NUMBER VARCHAR(255)	10000	50
Person	PID Name Nationalitaet	NUMBER PRIMARY KEY NOT NULL VARCHAR(40) NOT NULL CHAR(5)	6000	40
Autor	PID DiskID StueckID Funktion	NUMBER NOT NULL NUMBER NOT NULL NUMBER NOT NULL VARCHAR(20) NOT NULL	30000	40

Zur Effizienzsteigerung von Anfragen sind unter anderem die folgenden mittelbaren Indizes angelegt, die im Datenbanksystem jeweils als B*–Baum implementiert seien:

```
CREATE UNIQUE INDEX DiskIdIndex ON Disk(DiskID);
CREATE INDEX DiskTitelIndex ON Disk(DiskTitel);
CREATE INDEX PidIndex ON Autor (PID);
```

Außerdem sind folgende Parameter/Einstellungen bekannt:

- Jede Relation und jeder Index belegen jeweils einen eigenen Tablespace.
- Die Seitengröße sei 4 KBytes, die Länge des Seiten–Headers sei 96 Bytes.
- Der Füllgrad von Datenseiten sei 90%, von Indexseiten 70%.
- Der Datentyp NUMBER belege 10 Bytes.
- *Disktitel* sei Schlüsselkandidat der Relation *Disk* und habe durchschnittl. Länge von 90 Bytes.
- Ein TID oder Zeiger im B*–Baum habe die Länge 6 Bytes.
- Alle Werte eines Fremdschlüsselattributes kommen jeweils gleichhäufig vor. (Gleichverteilungsannahme).

Analysieren Sie die Speicherplatz– und Indexzugriffskosten dieser Datenbank. Vernachlässigen Sie dabei den Platzbedarf für die Slotlisten und für die Längenfelder des Datentyps VARCHAR.

Lösung zu Aufgabe 1:

1 Speicherkosten

a) Daten:
Datenseiten sind zu 90 % gefüllt, so daß pro Seite *0.9 * (4096– 96) = 3600* Bytes zur Verfügung stehen. Bei Betrachtung der Kardinalitäten und der Tupellängen ergeben sich somit im Mittel folgende Werte.

Relation	Anzahl Tupel pro Seite	Anzahl Seiten
Disk	Ein Tupel hat eine durchschnittliche Länge von 100 Bytes. Daher passen im Mittel $\lfloor 3600/100 \rfloor = 36$ Tupel auf eine Seite.	Da die Relation Disk 1000 Tupel enthält benötigen wir $\lceil 1000 / 36 \rceil = 28$ Seiten.
Musikstück	analog: $\lfloor 3600/50 \rfloor = 72$	analog: $\lceil 10000 / 72 \rceil = 139$
Person	analog: $\lfloor 3600/40 \rfloor = 90$	analog: $\lceil 6000 / 90 \rceil = 67$
Autor	analog: $\lfloor 3600/40 \rfloor = 90$	analog: $\lceil 30000 / 90 \rceil = 334$

b) Indizes:
Indexseiten sind zu 70 % gefüllt, so daß pro Seite *0.7 * (4096 – 96) = 2800* Bytes zur Verfügung stehen.

- Ein Eintrag (NUMMER und TID bzw. Sohnzeiger) des Index DiskIdIndex braucht 10 + 6 = 16 Bytes. Daher passen $\lfloor 2800 / 16 \rfloor = 175$ Einträge auf eine Seite. Da Diskid ein Primärschlüssel ist, und da die Relation Disk 1000 Tupel enthält, brauchen wir $\lceil 1000 / 175 \rceil = 6$ Blätter. Die inneren Knoten haben ebenfalls einen Fanout von 175 und somit reicht eine weitere 'Ebene' mit der Wurzel aus.
  Insgesamt brauchen wir 6 + 1 Blätter.
- Analog erhalten wir für den Index DiskTitelIndex $\lfloor 2800 / (90 + 6) \rfloor = 29$ Einträge pro Seite, und da Disktitel ein Schlüsselkandidat sein soll erhalten wir auf Blattebene 1000 Einträge und benötigen somit $\lceil 1000 / 29 \rceil = 35$ Seiten. Auf der Ebene über den Blättern brauchen wir 2 Seiten und diese werden wiederum in der Wurzel "zusammengefaßt". Insgesamt erhalten wir somit 38 Seiten.
- Bei dem Index PidIndex muß berücksichtigt werden, daß nur 6000 verschiedene PID's existieren (Primärschlüssel der Relation Person), jedoch 30000 Tupel der Relation Autor. Da eine Gleichverteilung angenommen wird, enthält jedes Blatt im Mittel eine TID–Liste der Länge 5. Somit ist die Größe eines Blatteintrags 10 + 5 * 6 = 40 Bytes und pro Blatt können wir $\lfloor 2800 / 40 \rfloor = 70$ Einträge speichern. Daraus folgt, daß wir $\lceil 6000 / 70 \rceil = 86$ Blätter brauchen. Die Größe eines Eintrags in einem inneren Knoten ist 10 + 6 Bytes und somit ist der Fanout $\lfloor 2800 / 16 \rfloor = 175$ und daher reicht die Wurzel als einziger Nicht–Blatt–Knoten aus.
  Insgesamt brauchen wir also 87 Seiten.

2 Zugriffskosten

- Die Höhe des B*–Baumes des Index DiskIdIndex ist 2 und jeder Eintrag eines Blattes enthält nur einen TID. Somit haben wir nach spätestens 3 Blockzugriffen den gewünschten Datensatz gefunden.

- Analog brauchen wir 3 + 1 Zugriffe um eine Disk über den Index DiskTitelIndex zu finden.
- Bei dem Index PidIndex haben wir im Mittel 5 TID's pro Blatteintrag, und damit brauchen wir 2 + 5 Blockzugriffe, um alle Tupel der Relation Autor mit einer bestimmten PID zu finden.

Aufgabe 2: B*–Baum

a) Fügen Sie in einen zu Beginn leeren B*–Baum vom Typ *m=2, m*=2* der Reihe nach die folgenden Schlüssel ein: *13, 97, 45, 23, 103, 84, 100, 99, 50, 70, 55, 47, 52, 54*.
Welches Aussehen hat der Baum nach Ausführung dieser Einfügunge–Operationen ?
Löschen Sie anschließend die Schlüssel *70* und *84*. Überlegen Sie sich dazu eine geeignete Reorganisation beim Löschen.

b) Die Speicherplatzausnutzung eines B*–Baums beträgt – von der Wurzel abgesehen – mindestens 50%. Bei zufälligen Einfügungen beträgt sie im Mittel ln(2) $\approx$ 69%.

   – Unter welchen Bedingungen wird – auch bei einer großen Anzahl von Einfügungen (also auch asymtotisch) – die schlechteste Speicherplatzausnutzung von 50% erreicht?

   – Wie kann man die Mindestspeicherplatzauslastung (und damit auch die mittlere Speicherplatzauslastung) generell, d.h. unabhängig von pathologischen Fällen, verbessern?

c) Bei der Entwicklung eines neuen Datenbanksystems hat der Chef des Designteams (, der leider leider in Saarbrücken studiert hat) die Idee, zur Unterstützung von Operationen auf B*–Bäumen für jeden Knoten einen Verweis zum Vater einzuführen.

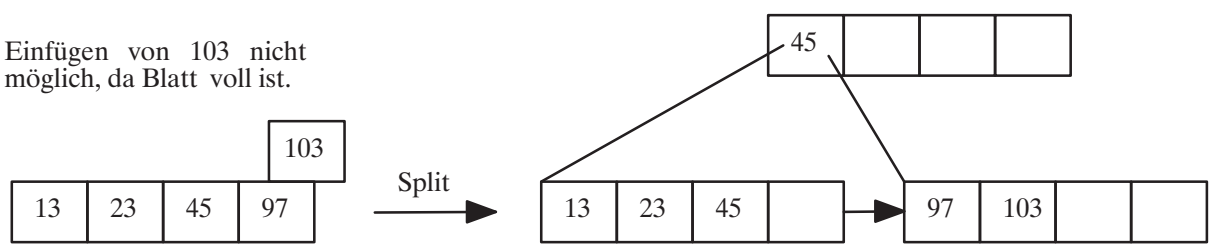
   – Bei welchen Operationen auf B*–Bäumen scheint dieser zusätzliche Verweis sinnvoll zu sein?

   – Warum ist die Idee des Systementwicklers letztlich aber doch alles andere als genial?

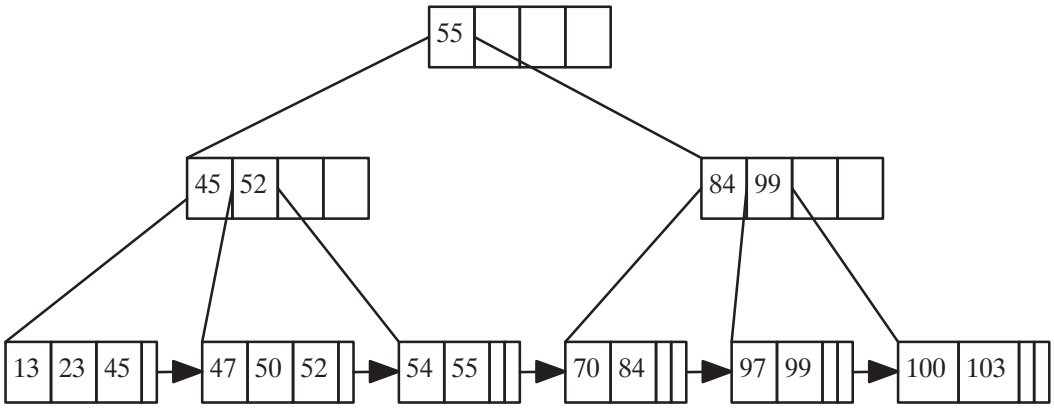
d) Die inneren Knoten eines B*–Baumes sind darauf ausgelegt, möglichst viele Einträge zu enthalten, um den Verzweigungsgrad zu maximieren. Sie erreichen oft Größenordnungen von mehreren Hundert Einträgen pro Knoten und damit pro Diskblock. Der CPU–Aufwand, der nötig ist um den Zeiger zu finden, dem der Suchalgorithmus bei gegebenem Suchschlüssel folgen muß, ist deshalb nicht unbedingt vernachlässigbar. Welche Verfahren würden Sie vorschlagen, um diese Suche zu implementieren? Unterscheiden Sie dabei zwischen Schlüsseln fester Länge (z.B. Typ NUMBER) und Schlüsseln variabler Länge (z.B. Typ VARCHAR(n)).

Lösung zu Aufgabe 2a):

Die ersten 4 Einfügungen können in ein Blatt erfolgen. Da ein Blatt jedoch höchstens 4 Einträge enthalten kann, muß bei der 5. Einfügeoperation ein 'Split' durchgeführt werden. Die folgende Abbildung verdeutlicht, wie eine solche Split–Operation vorgeht:



Analog erfolgen die weiteren Einfügungen bis wir zum Schluß den folgenden Baum erhalten (Zu beachten ist, daß beim Split eines Nicht–Blattknotens der Median in den Vaterknoten wandert):



Durch das Löschen von 70 würde ein Blatt den Mindestfüllgrad unterschreiten. Um die Invariante der Definition des B*–Baumes aufrecht zu erhalten, müßten wir also das Blatt mit seinem Bruder verschmelzen. Wenn die Summe der beiden Knoten, die verschmolzen werden, größer als die maximale Anzahl von Einträgen ist, so muß anschließend noch ein erneuter Split durchgeführt werden.
Da jedoch in fast allen praktischen Anwendungen von Datenbank die 'INSERTS' bei weitem überwiegen, und da die Verschmelzungen und die Splits einen zusätzlichen Aufwand darstellen, läßt man solche Unterschreitungen des Mindestfüllgrades zu und hofft, daß spätere INSERTS dies wieder beheben.

Lösung zu Aufgabe 2b):

– Unter welchen Bedingungen wird – auch bei einer großen Anzahl von Einfügungen (also auch asymtotisch) – die schlechteste Speicherplatzausnutzung von 50% erreicht?

Fügen wir eine sortierte Folge von Schlüsseln in einen B*–Baum ein, so erhalten wir ein einziges Blatt mit einer Anzahl von Schlüsseln zwischen m* und $2 \times$ m*. Alle anderen Blätter enthalten m* viele Schlüssel. Da die maximale Anzahl von Einträgen in den Blättern $2 \times$ m* ist, erhalten wir einen Füllgrad von ungefähr 50%.

– Wie kann man die Mindestspeicherplatzauslastung (und damit auch die mittlere Speicherplatzauslastung) generell, d.h. unabhängig von pathologischen Fällen, verbessern?

- Für den speziellen Fall des Einfügens von sortierten Folgen existiert in den meisten Systemen die Möglichkeit durch Setzen eines entsprechenden Parameters die spezielle Eigenschaft der Folge für ein effizientes Aufbauen (bzgl. Füllgrad und auch Laufzeit) des B*–Baumes auszunutzen. Dies ist vor allem vorteilhaft, wenn ganze Datenbanken geladen werden.
- Im laufenden Betrieb kann man den Füllgrad erhöhen, indem man einen Ausgleich zwischen mehreren Geschwisterknoten durchführt. Beim Überlauf eines Knotens wird versucht, Einträge in den Geschwisterknoten einzufügen. So könnte beispielsweise erst ein Split erfolgen, wenn zwei benachbarte Knoten voll sind. Aus den zwei gefüllten Knoten würde dann nur ein neuer Knoten abgesplittet, was in diesem Falle zu einem Mindestfüllgrad von 2/3 führen würde. Im allgemeinen Fall könnte man bei Betrachtung von m–1 um den Einfügeknoten liegenden Geschwisterknoten immer mindestens einen Füllgrad von m/(m+1) erreichen. Da dieses Vorgehen jedoch einen zusätzlichen Aufwand (insbesondere an I/O Aktivität) bewirkt, wird es in der Praxis meistens nicht angewendet.

Lösung zu Aufgabe 2c):

– Bei welchen Operationen auf B*–Bäumen scheint dieser zusätzliche Verweis sinnvoll zu sein?

Splits können sich im Baum nach oben propagieren, d.h. der Split eines Blattes kann dazu führen, daß der Vaterknoten auch einen Split durchführen muß, usw. Dazu müssen wir den Vaterknoten jedoch kennen. Dies können wir z.B. durch einen solchen Vaterverweis erreichen. Der analoge Fall kann auftreten, wenn wir beim Löschen Knoten zusammenfügen.

– Warum ist die Idee des Systementwicklers letztlich aber doch alles andere als genial?

Wird ein innerer Knoten gesplittet, so müssen wir die Vaterverweise aller seiner Kinder ändern und dazu alle entsprechenden Blöcke lesen. Durch den hohen Fanout der Knoten ist dieser Aufwand aber nicht gerechtfertigt. Der hohe Verzweigungsgrad bewirkt außerdem, daß die H"ohe des Baumes sehr gering ist (selbst für sehr große Datenbanken kleiner als 6). Daher ist es besser, wenn wir uns den Vater jeweils explizit merken kann, oder durch eine rekursive Implementierung der Einfügeoperation dies implizit tun.

Lösung zu Aufgabe 2d):

1. **Konstante Schlüssellänge**
   Bei Schlüsseln konstanter Länge kann ein Eintrag direkt über seine Position innerhalb einer Seite adressiert werden. Es sind keine zusätzlichen 'Inhaltsverzeichnisse' wie TID–Slots nötig.
   Eine offensichtliche Vorgehensweise ist die *lineare Suche*, wo die Seite sequentiell von vorne nach hinten durchlaufen wird. Im Mittel sind hier bei m Einträgen m/2 Vergleichsoperationen nötig. Hier braucht keine Sortierung innerhalb der Seite vorzuliegen.
   Bei einer geordneten Seite kann die *Sprungsuche* sowie eine *binäre Suche* durchgeführt werden. Bei der Sprungsuche wird die Seite in n Intervalle fester Länge eingeteilt. Beim ersten Durchlauf wird durch Vergleich mit dem größten Schlüssel jedes Intervalles das Intervall identifiziert, wo der gesuchte Schlüssel liegen muß. Danach wird linear innerhalb des Intervalles gesucht. Bei m Einträgen innerhalb einer Seite sind n/2 + m/(2*n) Vergleiche nötig. Wegen der Optimalität für $n = \sqrt{m}$ wird die Sprungsuche oft auch als Quadratwurzel–Suche bezeichnet.
   Beim binären Suchen sind bei m Einträgen im Mittel $\log_2(m) - 1$ Vergleiche nötig.

2. **Variable Schlüssellänge**
   Variable Schlüssellänge kann durch Schlüsseltypen variabler Länge (z.B. VARCHAR(n)) entstehen sowie durch die Kompression von Schlüsselfeldern fester Länge (siehe dazu auch Aufgabe 3). Hier entsteht das Problem, daß die Position eines Eintrags innerhalb einer Seite nicht bekannt ist. Wir können ohne Vorhandensein zusätzlicher Strukturen nur die lineare Suche durchführend. Sind jedoch TID–Slots am Ende der Seite vorhanden, so können wir die Sprungsuche und binäre Suche auf diesem Verzeichnis ausführen. Durch die TID–Slots wird jedoch zusätzlicher Platz innerhalb der Seite belegt, der den Verzweigungsgrad des Baumes reduziert, was den I/O–Aufwand wiederum erhöht. Da I/O–Zugriffe in der Regel wesentlich teurer sind als Vergleichsoperationen, muß hier also sorgfältig abgewogen werden, ob sich die binäre Suche auf Kosten der Speicherreduktion auszahlt.

Aufgabe 3: Kostenschätzung für Indizes

Betrachten Sie einen als B*–Baum realisierten Textindex über 10 Mio. Dokumente. Pro Dokument gebe es 100 Suchwörter, die zu indizieren sind; insgesamt gebe es 1 Mio. verschiedene Suchwörter, die alle gleichhäufig vorkommen.

(a) Wie groß (Anzahl der Blattseiten und der Blattknoten, Höhe) wird der B*–Baum? Wieviele Indexseitenzugriffe sind notwendig, um alle Dokumente zu bestimmen, die ein bestimmtes Suchwort enthalten? (2 Punkte)
Gehen Sie von einer Nettoseitengröße von 4000 Bytes aus, und nehmen Sie einen Seitenfüllgrad von 70 Prozent an. Die Länge eines Suchworts sei konstant 30 Bytes, die Länge eines TIDs (bzw. Zeigers auf ein Dokument, z.B. auch eine URL) sei 10 Bytes, und die Länge eines Sohnzeigers innerhalb des Baums sei 4 Bytes.
Hinweise:
– Ein Blattknoten kann aus mehreren miteinander verketteten Blattseiten bestehen.
– Ein Baum der Höhe 1 besteht nur aus der Wurzel.

Beispiellösung zu a):

Anzahl der Referenzen:	1 Mrd (10^9)
Anzahl verschiedener Suchwörter:	1 Mio (10^6)
Mittlere Anzahl von Referenzen pro Suchwort:	1000 (wg. Gleichverteilung)
Länge eines Blatteintrages:	30 Bytes + 1000 * 10 Bytes= 10030 Bytes
Anzahl von Einträgen pro Seite:	$\frac{0.7 \times 4000}{10030} < 1 \Rightarrow$ Mehrere Seiten/Blatt
Anzahl s der Seiten pro Blatt:	$s = \left\lceil \frac{10030}{2800} \right\rceil = 4\ Seiten$
Anzahl der Blätter:	1 Mio.
Anzahl der Blattseiten:	1 Mio * 4 Seiten = 4 Mio. Seiten
Länge eines Eintrages in inneren Knoten:	30 Bytes + 4 Bytes = 34 Bytes (Suchwort+Sohnzeiger)
Für Knotenfanout f muß nun gelten:	$34\ Bytes \times (f-1) + 4\ Bytes \leq 0.7 \times 4000$
Daraus ergibt sich für den Fanout f:	$f \leq 83$
Die Höhe h des Index ist damit:	$h = \lceil log_{83}(1\ Mio.) \rceil + 1 = \mathbf{5}$
Bestimmung aller TIDs zu einem Suchwort durch	$h + s - 1 = 8$ Seitenzugriffe

(b) Wie verändert sich die Lösung zu a), wenn Sie in den Nichtblattseiten des Index eine Schlüsselkomprimierung durchführen, bei der die Suchwörter innerhalb der Wegweiser (konstant) um den Faktor 2 verkürzt werden?

Beispiellösung zu b):

Länge eines Eintrages in inneren Knoten:	**15 Bytes** + 4 Bytes = 19 Bytes (Suchwort+Sohnzeiger)

Für Knotenfanout f muß nun gelten: $19\,Bytes \times (f-1) + 4\,Bytes \leq 0.7 \times 4000$

Daraus ergibt sich für den Fanout f: $f \leq \mathbf{148}$

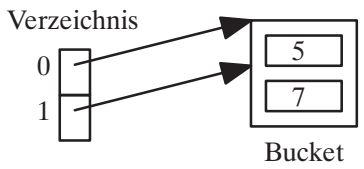
Die Höhe h des Index ist damit: $h = \lceil log_{148}(1\,Mio.) \rceil + 1 = \mathbf{4}$

Bestimmung aller TIDs zu einem Suchwort durch $h + s - 1 = \mathbf{7}$ Seitenzugriffe

Aufgabe 5: Erweiterbares Hashing

In der Vorlesung wurde das Erweiterbare Hashing (Extendible Hashing) als eine Möglichkeit des dynamischen Hashings vorgestellt. Ziel dynamischer Hash–Verfahren ist die dynamische Anpassung des Bildbereiches an wechselndes Datenvolumen, um die gute Suchperformance zu erhalten. Das Rehashing von Einträgen darf dabei jedoch den laufenden Betrieb nicht zu lange aufhalten.

Gegeben sei der folgende erweiterbare Hash–Index der globalen Tiefe *d=1*. Die Hash–Funktion sei *h(k)= k*. Die Anzahl der Buckets sei *N=1* und ein Bucket beinhalte maximal *2 Tupel*, die hier nur durch ihren Schlüsselwert angedeutet sind.



a) Welches ist die lokale Tiefe des Buckets im Hash–Index?

b) Fügen Sie die Werte *1, 10, 5 und (wieder) 5* in den Hash–Index ein.

c) Warum ist es im Allgemeinen sinnvoller, für die Zuweisung eines Buckets die hinteren Bits des Hash–Ergebnisses *h(k)* zu verwenden, und nicht die vorderen Bits?

Lösung zu Aufgabe 5 a)

Das Verzeichnis des Index enthält 2^1 Einträge, was eine globale Tiefe von d=1 ergibt. Da 2^{1-0} Einträge auf das Bucket verweisen, ergibt sich eine lokale Tiefe von 0. Es werden also für die Zuweisung eines Eintrages in dieses Bucket noch keine Bits betrachtet. Jeder Eintrag wird diesem Bucket zugeordnet.

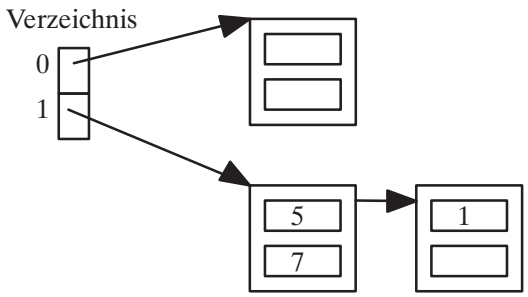
Lösung zu Aufgabe 5 b)

Einfügen von 1:

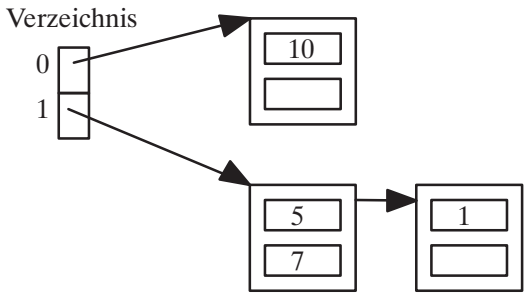
Der Wert 1 wird dem (einzigen) Bucket zugeordnet, was einen Überlauf zur Folge hat. Da die lokale Tiefe i=0 des Buckets kleiner als die globale Tiefe d=1 des Index ist, wird ein neues Bucket erzeugt und im Verzeichnis eingetragen. Die lokale Tiefe beider Buckets wird damit i+1. Danach werden die Einträge im gesplitteten Bucket neu verteilt (rehashing). Es ergibt sich in diesem Fall allerdings das Problem, daß die Verteilung keinen Nutzen bringt, da sich die Schlüsselwerte im letzten Bit nicht unterscheiden. Das Rehashing erzeugt erneut einen Überlauf im zu splittenden Bucket. Es gibt prinzipiell nun zwei Möglichkeiten: Man könnte einerseits die Tiefe des Verzeichnisses erweitern und damit die Anzahl der zu betrachtenden Bits erhöhen. In unserem Beispiel würde das zu einer Verteilung der Einträge führen, da sich die Schlüssel im nächsten Bit unterscheiden und damit auf die neuen Buckets verteilt würden. Der Nachteil ist allerdings, daß das Rehashing evtl. mehrfach gestartet wird, bis es eine

Verteilung erzielt. Dies kann den Normalbetrieb negativ in der Performance beeinflußen. Im Extremfall würde diese Vorgehensweise außerdem nie zu einer Verteilung führen, nämlich wenn die Schlüsselwerte mehrfach vorkommen können. Der Vorteil ist, daß eine schließlich doch erfolgreiche Verteilung die gute Suchperformance erhält.

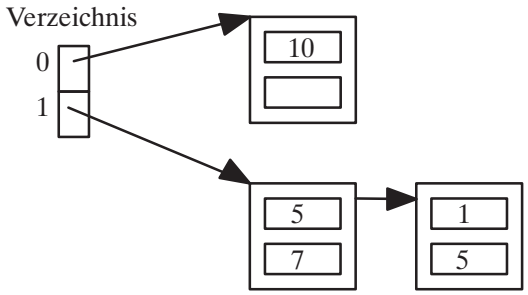
Die alternative Möglichkeit ist, beim Überlauf eines Buckets während des Rehashings desselben Buckets einen Überlaufbereich einzurichten und diesen mit dem Bucket zu verketten. Hier wird zunächst kein neues Rehashing initiiert. Erst wenn dieser Überlaufbereich überläuft, wird ein neuer Split des Buckets gestartet. Der Vorteil liegt darin, daß die Kosten für das Rehashing niedrig sind. Dagegen wird die Suchperformance schlechter, da beim Suchen in dem übergelaufenen Bereich potentiell auf mehrere Seiten sequentiell zugegriffen werden muß. Dies kann aber ein temporärer Effekt sein, der beim nächsten Überlauf eines solchen Bereiches, wo die globale Verzeichnistiefe evtl. sowieso schon gewachsen ist und eine insgesamt gleichmäßigere Verteilung der Schlüsselwerte eingetreten ist, kompensiert werden. Wir richten deshalb einen Überlaufbereich ein, wodurch sich nun folgender Index ergibt:



Einfügen von 10:

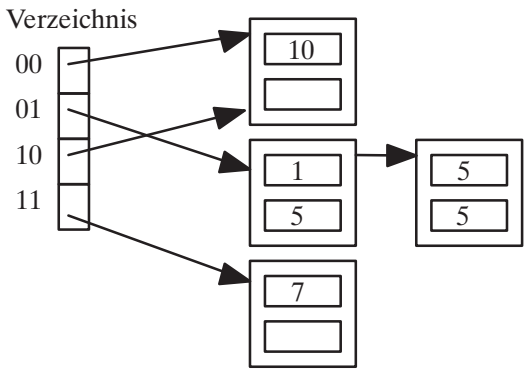


Einfügen von 5:



Erneutes Einfügen von 5:

Das erneute Einfügen von 5 erzeugt nun einen Überlauf. Da die lokale Tiefe des Buckets gleich der globalen Tiefe des Verzeichnisses ist, wird die Verzeichnistiefe inkrementiert, das Verzeichnis kopiert und das Bucket schließlich gesplittet. Das Bucket für die Bits ’01’ läuft während seines nicht über. Der Überlaufbereich jedoch bleibt erhalten. Es ergibt sich schließlich folgende Struktur:



Lösung zu Aufgabe 5 c)

Ziel des Hash–Index ist die möglichst gleichmäßige Verteilung der Einträge in die Buckets. Die betrachteten Bits des Verzeichnisses sollten also möglichst gleich häufig vorkommen. Bei vielen Anwendungen wird beim erweiterbaren Hash–Index oft die einfache Hash–Funktion h(k)=k verwendet, wobei k der Wert eines numerischen Primärschlüssels einer Relation ist (z.B. Kundennummer, Artikelnummer, Rechnungsnummer..). Diese Werte werden oft in irgendeiner Form fortlaufend durchnummeriert. Bei einer Bitweite von 32 Bits bleiben deshalb die höherwertigen Bits über sehr lange Zeiträume konstant, wodurch keine gleichmäßige Verteilung der Einträge erfolgen würde.