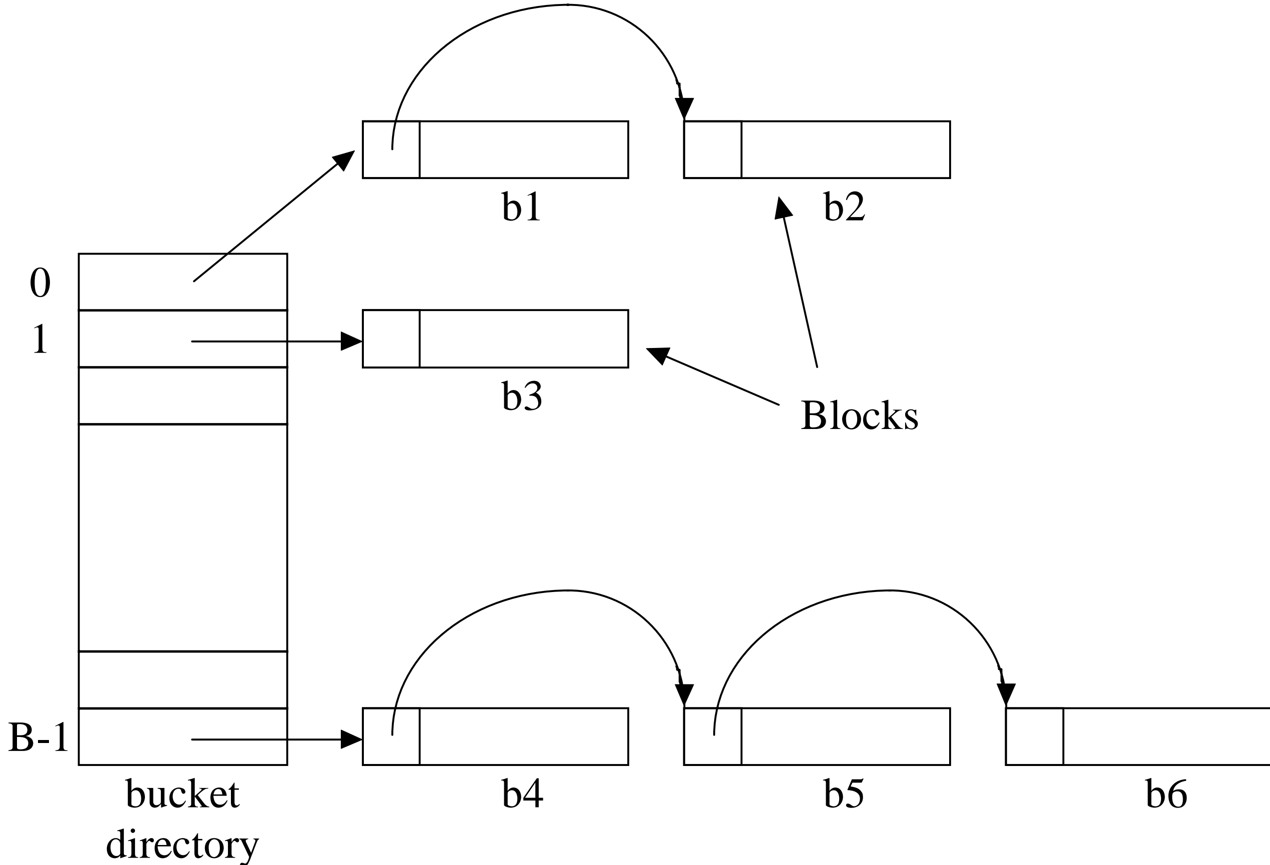
HASHORGANISATION

Literatur:
Jeffrey D. Ullman: Principles of Database Systems, 2nd Edition 1982, Kapitel 2.2

- Die Sätze der Datei werden auf eine Menge von **Buckets** aufgeteilt.
- Jedes Bucket besteht aus einem oder mehreren **Blöcken**.
- Die Aufteilung geschieht über eine **Hashfunktion h**:
  ein Satz mit dem Schlüsselwert v wird im Bucket mit der Nummer **h(v)** eingetragen.
- Die **Hashtabelle** (Bucket directory) enthält die Adressen des ersten Blocks in jedem Bucket.
- Im **Header** jedes Blocks befindet sich ein Zeiger auf den nächsten Block des Buckets (sofern vorhanden).


b1
b2
0
1
b3
Blocks
B-1
bucket
directory
b4
b5
b6

Die Hashfunktion:

Eine wünschenswerte Eigenschaft der Hashfunktion ist, dass sie „gut hasht“. Das heißt, h(v) ist über alle möglichen Werte von v gleich gut verteilt.

Für kleine B kann das Bucketdirectory im Hauptspeicher gehalten werden. Sonst wird es über mehrere Blöcke verteilt.

Beispiel 1:

Gegeben sei eine Folge von Datensätzen mit dem Integerwert v als Schlüssel, sowie ein Bucketdirectory mit 5 Buckets.

Wir verwenden die Hashfunktion

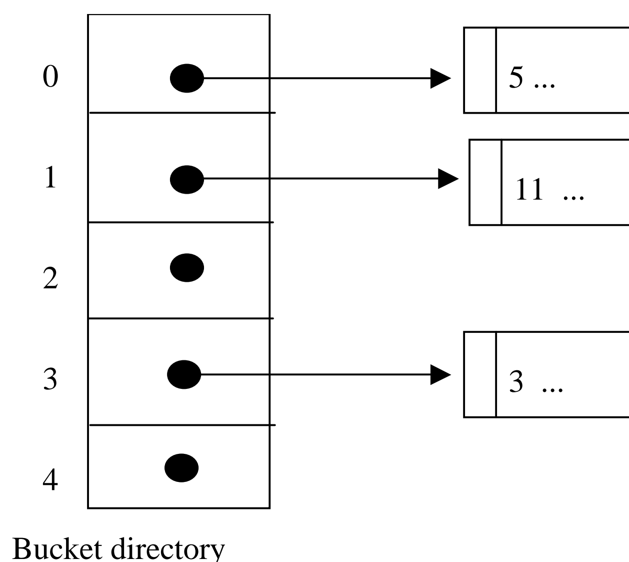
$h(v) = v \bmod 5$,

die alle Datensätze auf die 5 Buckets verteilt:

$v_1 = 3$ → $h(v_1) = 3 \bmod 5 = 3$ → Datensatz kommt in Bucket 3

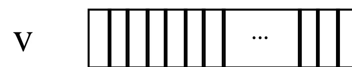
$v_2 = 5$ → $h(v_2) = 5 \bmod 5 = 0$ → Datensatz kommt in Bucket 0

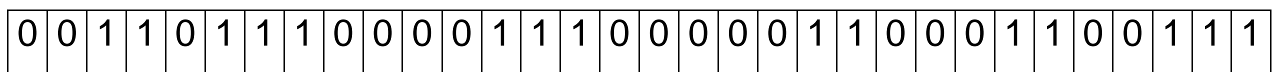
$v_3 = 11$ → $h(v_3) = 11 \bmod 5 = 1$ → Datensatz kommt in Bucket 1



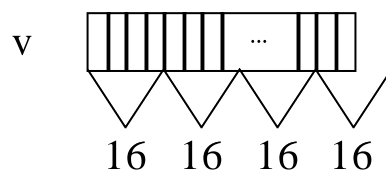
Beispiel 2:

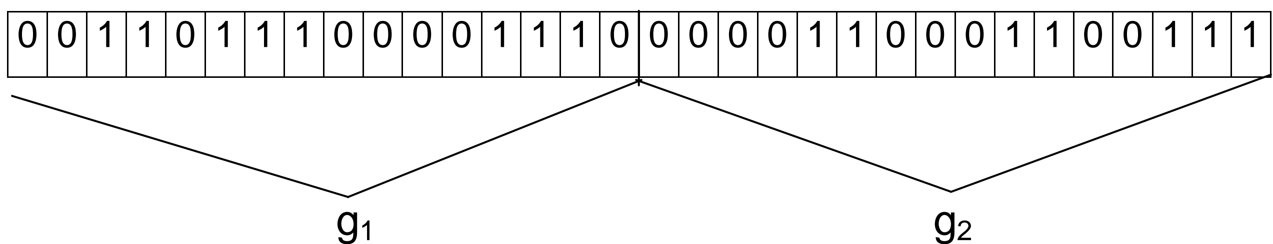
1. Wir nehmen an der Schlüssel v liegt in Form von n Bits vor.





2. Die Bits werden in Gruppen g_i zu jeweils 16 Bit zusammengefasst. Die letzte Gruppe wird, falls nötig, mit 0-Bits aufgefüllt.





3. Die Gruppen g_i werden jetzt als Integer-Zahlen behandelt und addiert.

$$S = \sum g_i$$

$$S = g_1 + g_2 = 14094 + 3175 = 17269$$

4. Die Summe wird durch die Anzahl der Buckets dividiert und der Rest wird als Bucket-Nummer des Satzes genommen.

$$r = S \mathit{mod} \#\text{Buckets}$$

Für eine Bucketzahl von 100 wären das zum Beispiel:

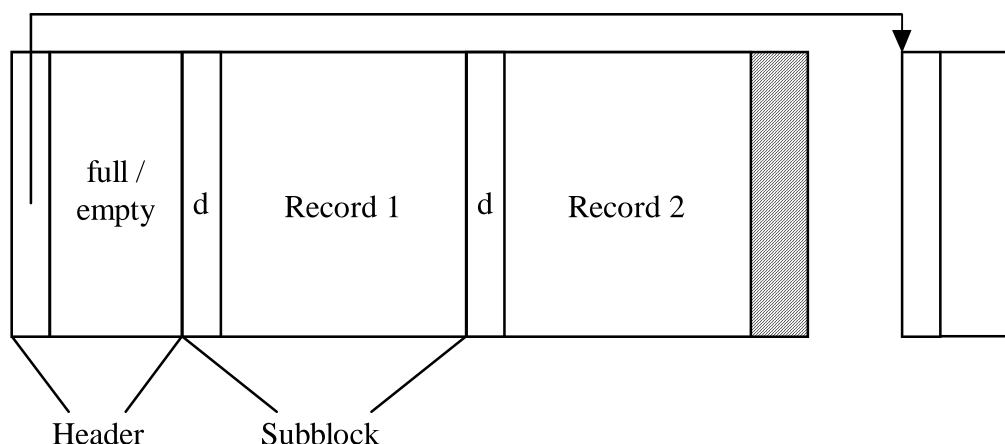
$$r = 17269 \mathit{mod} 100 = 69$$

Full/Empty Bit:

Um leere Subblöcke von gefüllten unterscheiden zu können, führt man im Header jedes Blocks, für jeden Subblock ein full/empty Bit ein.

Deletion Bit:

Falls pinned Records auftreten können, reicht die Unterscheidung zwischen gefüllten und leeren Subblöcken nicht aus. Es muss dann ein zusätzliches Bit eingeführt werden, dass verhindert das Subblöcke von gelöschten Records wiederverwendet werden.



Lookup (Suchen):

Um einen bestimmten Satz finden zu können muss man folgendermaßen vorgehen:

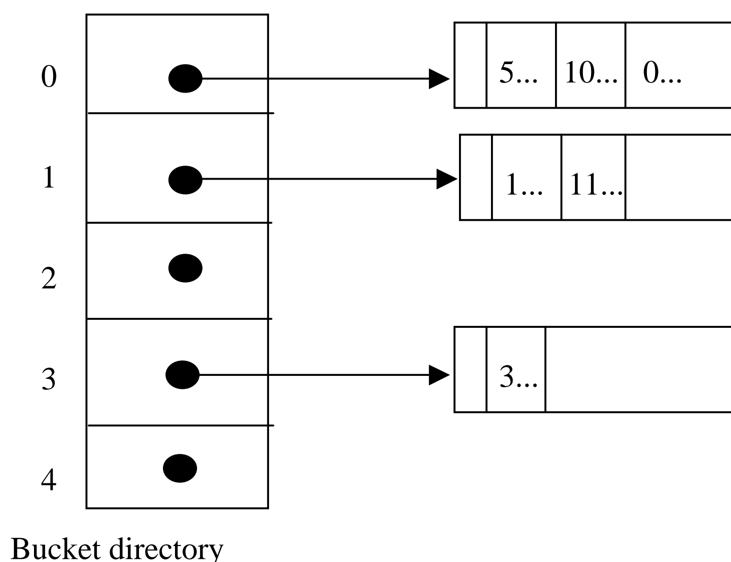
1. Zunächst benötigt man den Schlüssel *v* des gesuchten Satzes.

2. Mit der Hashfunktion h berechnet man die Position $i = h(v)$ des entsprechenden Buckets in der Hashtabelle.

3. Sequentielles Durchsuchen der Blöcke nach dem gewünschten Satz.

Beispiel:

Gegeben sei die unten stehende Hashtabelle mit der Hashfunktion h(v) = v mod 5.

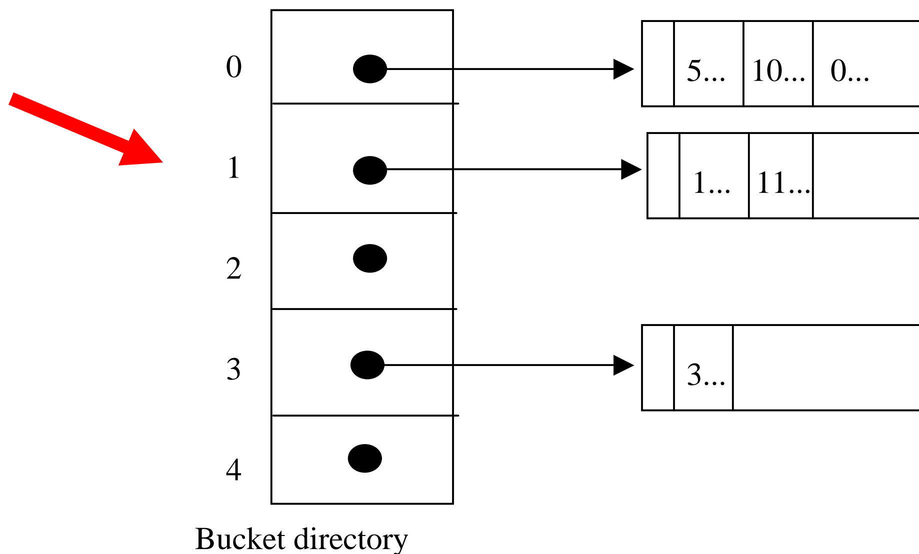


Beispiel (Fortsetzung):

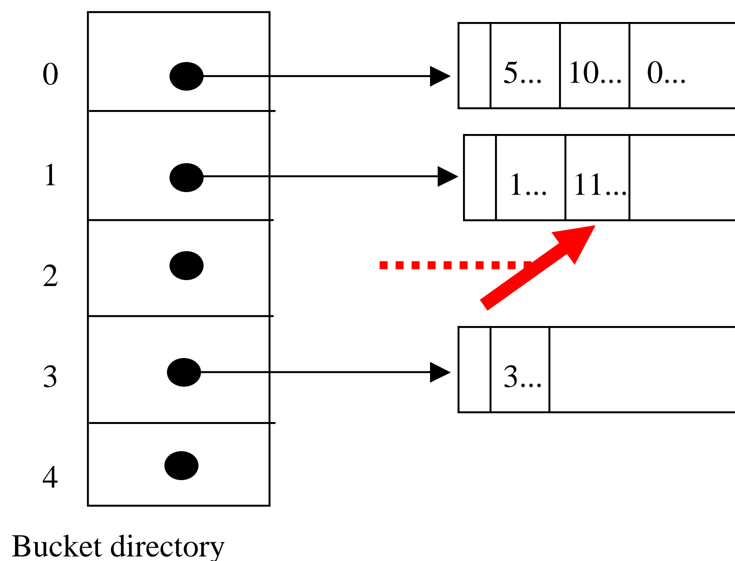
1. Gesucht werde der Datensatz mit dem Schlüssel 11.

2. Berechne den Hashwert h(11) = 11 mod 5 = 1

   → Der Datensatz befindet sich also im Bucket 1.



3. Sequentielles Durchsuchen der Blöcke im Bucket 1 nach dem Datensatz mit Schlüssel 11.



Insertion (Einfügen):

Bei dem Einfügen eines Satzes mit dem Schlüssel *v* wird folgendes Verfahren angewandt:

1. Wende das Lookup-Verfahren an, um den Satz zu suchen.

   Wenn der Satz gefunden werden konnte, kann er nicht ein weiteres mal eingefügt werden → Fehler!
   (Das Einfügen eines Datensatzes ist natürlich nur möglich, wenn noch kein Datensatz mit diesem Schlüssel existiert!)

2. Konnte der Satz nicht gefunden werden, suche einen freien Subblock in den Blöcken für Bucket h(v).

   Wenn kein freier Subblock verfügbar ist, dann fordere einen neuen Block vom Dateisystem an und setze einen Zeiger auf den neuen Block im Header des bisher letzen Blocks im Bucket.

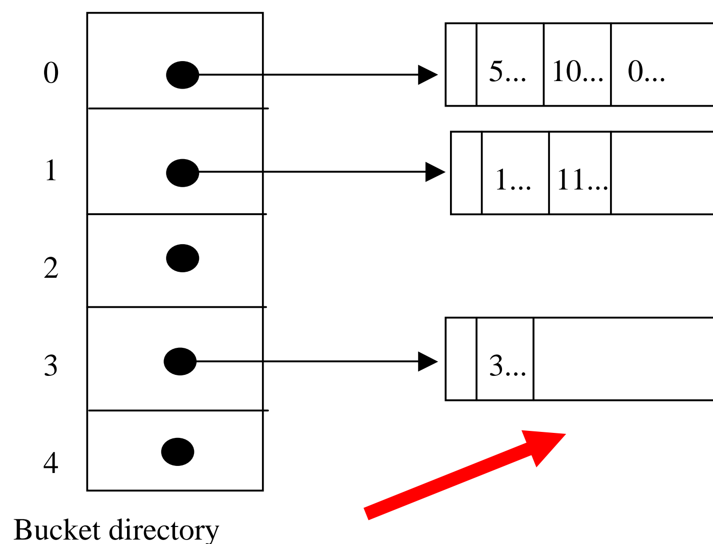
3. Füge den Satz in den freien Subblock ein.

Beispiel:

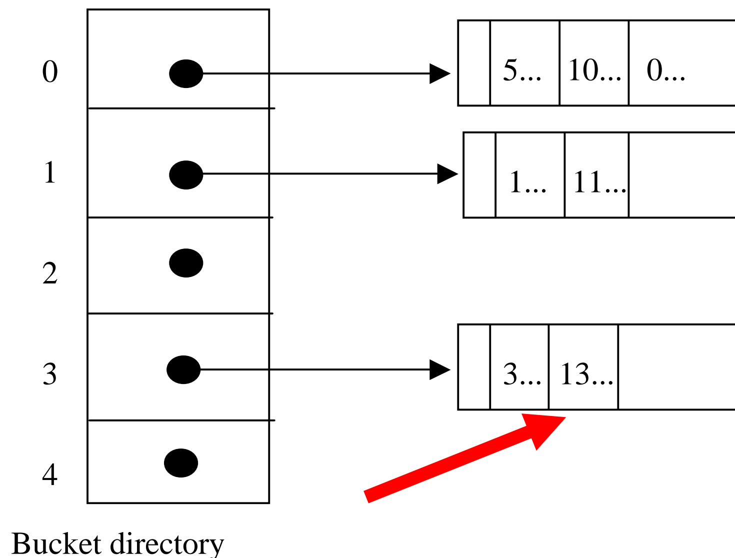
Es soll ein Datensatz mit dem Schlüssel 13 eingefügt werden.

1. Wende das Lookup-Verfahren an, um den Satz mit dem Schlüssel 13 zu suchen.

2. Konnte der Satz nicht gefunden werden, suche einen freien Subblock in den Blöcken für Bucket h(13) = 13 mod 5 = 3.



3. Füge den Satz in den freien Subblock ein.



Deletion (Löschen):

Um einen Satz zu löschen, geht man folgendermaßen vor:

1. Finden das Satzes nach dem Lookup-Verfahren.
2. Setzen des entsprechen full/empty-Bits auf 0 damit der Subblock wieder verwendet werden kann.

Bemerkungen:

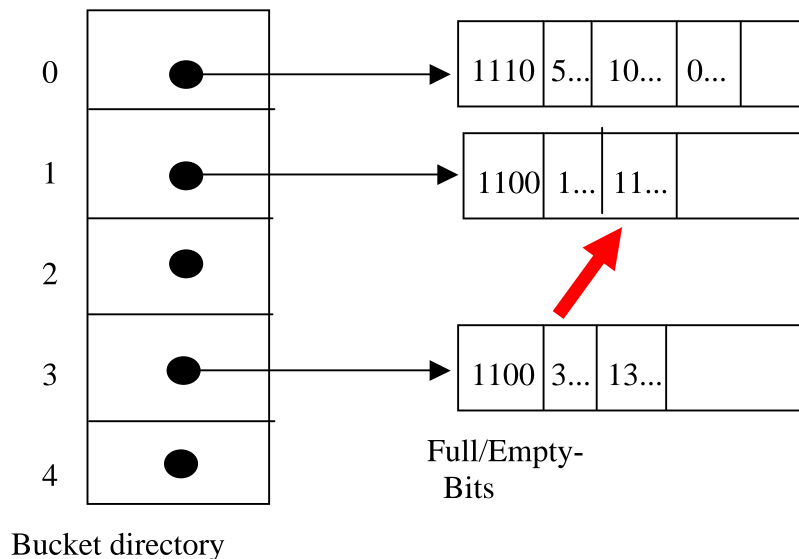
- Bei pinned Records muss man anders vorgehen:
  in Schritt 2 wird das full/empty-Bit auf 1 gelassen und statt dessen das deleted-Bit des Subblocks auf 1 gesetzt.
  Wenn jetzt ein Zeiger auf den gelöschten Satz zeigt, kann erkannt werden, dass dieser Satz bereits gelöscht ist, und der Zeiger wird ebenfalls gelöscht oder auf NULL gesetzt.

- Bei unpinned Records kann der Löschalgorithmus so konstruiert werden, dass der letzte Satz aus dem letzten Block des Buckets in den freigewordenen Subblock verlegt wird. Wenn dabei der letzte Block keinen weiteren Satz mehr beinhaltet, kann er dem Dateisystem wieder zurückgegeben werden.

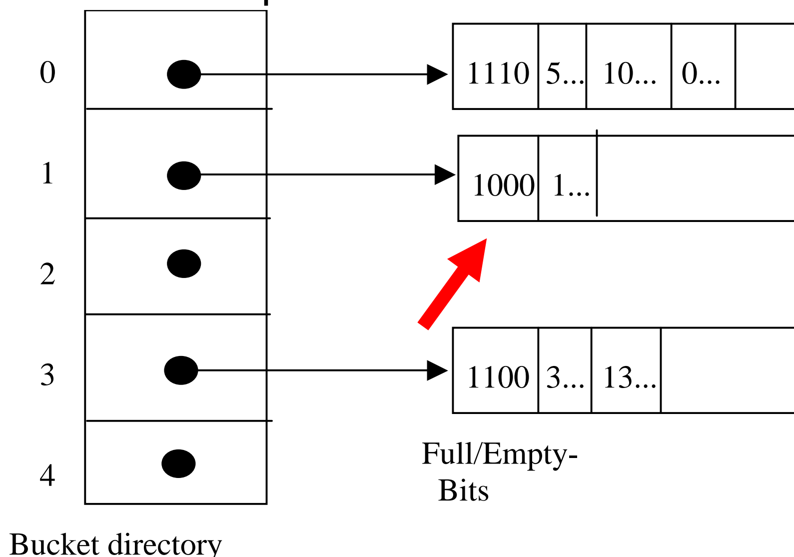
Beispiel:

Es soll der Datensatz mit dem Schlüssel 11 gelöscht werden.

1. Finden den Satz mit dem Schlüssel 11 nach dem Lookup-Verfahren.



2. Setzen des entsprechen full/empty-Bits auf 0 damit der Subblock wieder verwendet werden kann.
   Außerdem kann man noch den entsprechenden Eintrag löschen oder ihn später einfach überschreiben lassen.



Modification (Verändern):

Wenn eines oder mehrere Felder eines Satzes verändert werden sollen, wird nach folgendem Algorithmus vorgegangen:

wenn eines der Felder der Schlüssel ist

dann	lösche den Satz; füge den Satz erneut (an der richtigen Stelle) ein.
sonst	suche den Satz; verändere den Satz.

Bemerkung:

Im Falle von pinned Records kann der Schlüssel eines Satzes nicht verändert werden.

Beispiel: Dinosaurierdatei (Ullman):

Ein Satz, der Informationen über einen Dinosaurier enthält, hat folgende Struktur:

```
NAME:     CHAR(20)   /* key */
PERIOD:   CHAR(10)
HABITAT:  CHAR(5)
DIET:     CHAR(5)
LENGTH:   INT  /* in feet */
WEIGHT:   INT  /* in tons */
```

Annahmen:

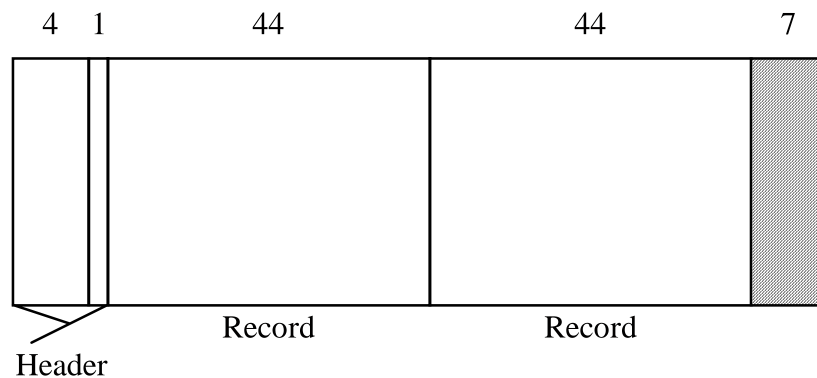
- Integer haben eine Länge von 2 Byte.

  → Ein Satz ist 44 Bytes groß.

- Blöcke haben eine Länge von 100 Byte.

- Jeder Block hat einen Header mit einem 4 Byte großem Zeiger auf den nächsten Block und einem Byte, das die full/empty Bits beinhaltet.

Ein Block in unserer Datei hat also folgenden Aufbau:



Als Hashfunktion wird die

Länge des Schlüssels *v* modulo 5

gewählt. Daher besteht die Hashtabelle aus fünf Einträgen.

Aufgabe:

Im Folgenden werden jetzt

1. Ein Satz für den Elasmosaurus eingefügt: (Elasmosaurus, Cretaceous, sea, carn., 40, 5).
2. Der Satz für den Brontosaurus wird modifiziert, denn der richtige Name für den Brontosaurus ist Apatosaurus: (**Apatosaurus**, Jurassic, lake, herb., 70, 25).

Die Dino-Datenbank vor den Transaktionen:

pointer (null)

full/empty information

one record

one record

waste space

bucket directory

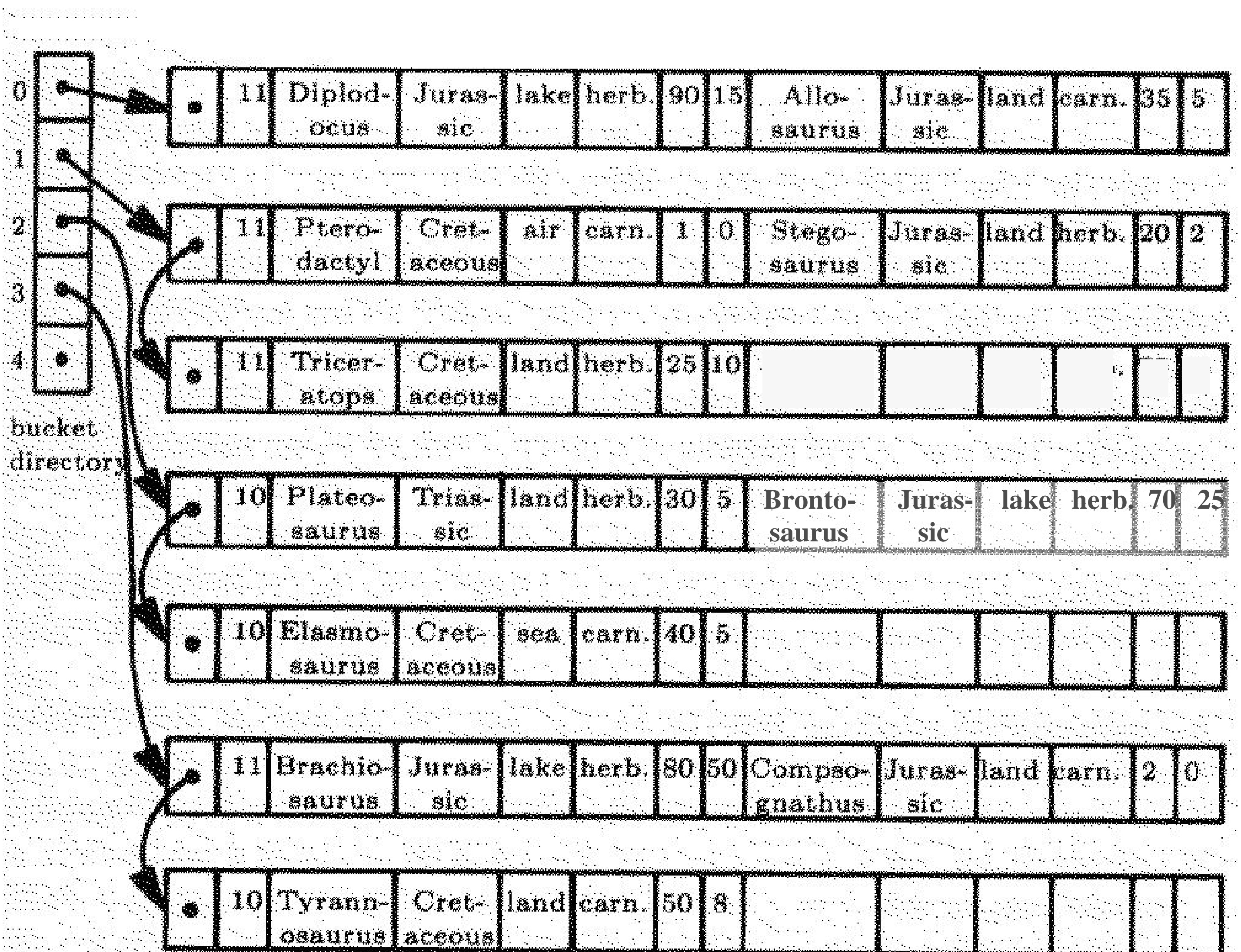
Bucket	Pointer	Full/empty	Name	Period	Habitat	Diet			Name	Period	Habitat	Diet		
0	•	11	Diplod-ocus	Juras-sic	lake	herb.	90	15	Allo-saurus	Juras-sic	land	carn.	35	5
1	→	11	Ptero-dactyl	Cret-aceous	air	carn.	1	0	Stego-saurus	Juras-sic	land	herb.	20	2
	•	10	Tricer-atops	Cret-aceous	land	herb.	25	10						
2	•	11	Plateo-saurus	Trias-sic	land	herb.	30	5	Bronto-saurus	Juras-sic	lake	herb.	70	25
3	→	11	Brachio-saurus	Juras-sic	lake	herb.	80	50	Compso-gnathus	Juras-sic	land	carn.	2	0
	•	10	Tyrann-osaurus	Cret-aceous	land	carn.	50	8						
4	•													

1. Einfügen des gewünschten Datensatzes

(Elasmosaurus, Cretaceous, sea, carn., 40, 5)

in die Dino-Datenbank:

v(Elsamosaurus) = 12

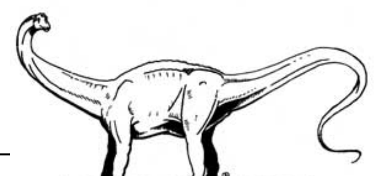


2. Modifikation des inkorrekten Datensatzes (Brontosaurus, Jurassic, lake, herb., 70, 25) in (Apatosaurus, Jurassic, lake, herb., 70, 25):

Dazu wird der ursprüngliche Datensatz **zwischen-gespeichert** und **gelöscht** und anschließend der neue Datensatz **eingefügt**.

0
1
2
3
4
bucket directory

•	11	Diplod-ocus	Juras-sic	lake	herb.	90	15	Allo-saurus	Juras-sic	land	carn.	35	5
•	11	Ptero-dactyl	Cret-aceous	air	carn.	1	0	Stego-saurus	Juras-sic	land	herb.	20	2
•	11	Tricer-atops	Cret-aceous	land	herb.	25	10	Apato-saurus	Juras-sic	lake	herb.	70	25
•	10	Plateo-saurus	Trias-sic	land	herb.	30	5						
•	10	Elasmo-saurus	Cret-aceous	sea	carn.	40	5						
•	11	Brachio-saurus	Juras-sic	lake	herb.	80	50	Compso-gnathus	Juras-sic	land	carn.	2	0
•	10	Tyrann-osaurus	Cret-aceous	land	carn.	50	8						



Laufzeitanalyse für Hashing:

Jede der betrachteten Operationen Lookup, Modify, Insert und Delete benötigen einen Zugriff auf den Sekundärspeicher, falls die Hashtabelle nicht im Hauptspeicher gehalten wird, um den entsprechenden Block der Hashtabelle zu laden.

Bei der sequentiellen Suche nach dem Satz werden im Schnitt die Hälfte der Blöcke eines Buckets durchsucht.

Für jede Operation außer Lookup muss außerdem der modifizierte Block wieder zurückgeschrieben werden.

Wenn also n die maximale Anzahl der Blöcke in einem Bucket ist, dann benötigt man für

- Lookup: $1 + n/2$ Zugriffe und für
- alle anderen Operationen $1 + n/2 + 1$

Zugriffe auf den Sekundärspeicher.

Best Case:

- Lookup: $1 + 1 = 2$ Zugriffe
- sonst $1 + 1 + 1 = 3$ Zugriffe

Optimierung:

Hashing kann daher optimiert werden, indem man die Anzahl der Blöcke pro Bucket möglichst reduziert.

$$\rightarrow \; Anzahl\ der\ Buckets \cong \frac{Anzahl\ der\ Sätze}{Sätze\ in\ einem\ Block}$$

Wenn die Datei dann größer wird, wird es notwendig, die Datei zu reorganisieren, indem die Hashfunktion geändert und die Hashtabelle vergrößert wird. Dies kann wie folgt geschehen:

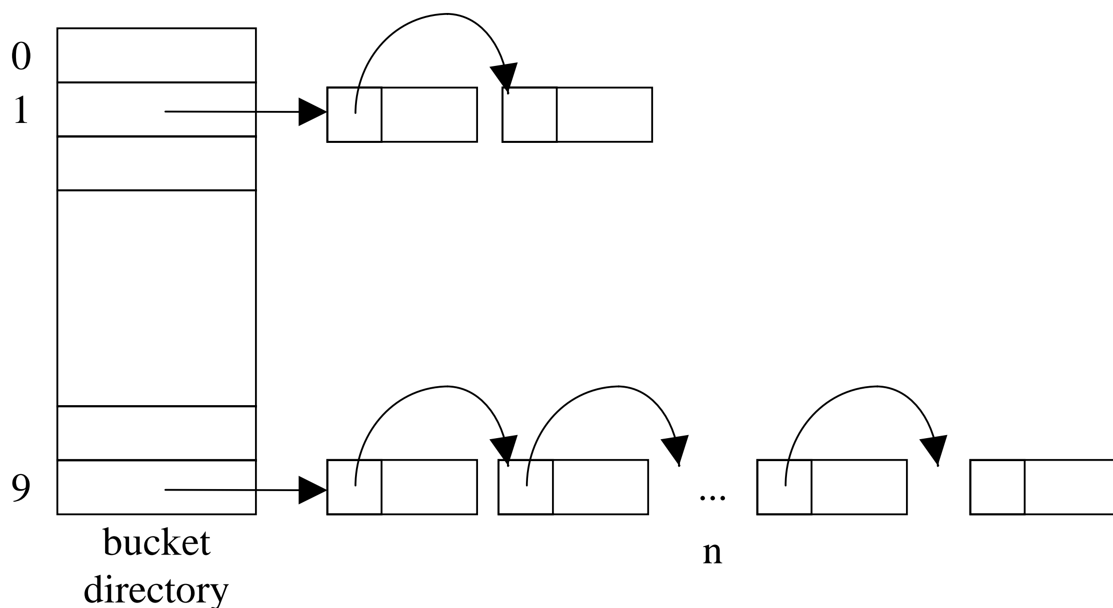
1. Die Hashfunktion berechnet aus dem Schlüssel v eine große Integerzahl (>> # Buckets) und dividiert diese modulo der Anzahl der Buckets.

2. Wenn die Datei reorganisiert wird, wird die Anzahl der Buckets um einen bestimmten Faktor c vergrößert (c=2). Wird die Anzahl der Buckets von n auf $2n$ vergrößert, werden alle Blöcke aus dem Bucket i ausschließlich auf die Buckets i und $i + n$ verteilt.

Entartung einer Hashtabelle:

Die Effizienz einer Hashtabelle beruht auf dem schnellen Finden des Buckets, das einen bestimmten Datensatz enthält.

Innerhalb dieses Buckets wird dann sequentiell nach dem Datensatz gesucht. Sind in einem Bucket zu viele Datensätze (bzw. Blöcke), weil zum Beispiel die Hashfunktion schlecht gewählt wurde, ist die Suche ineffizient:

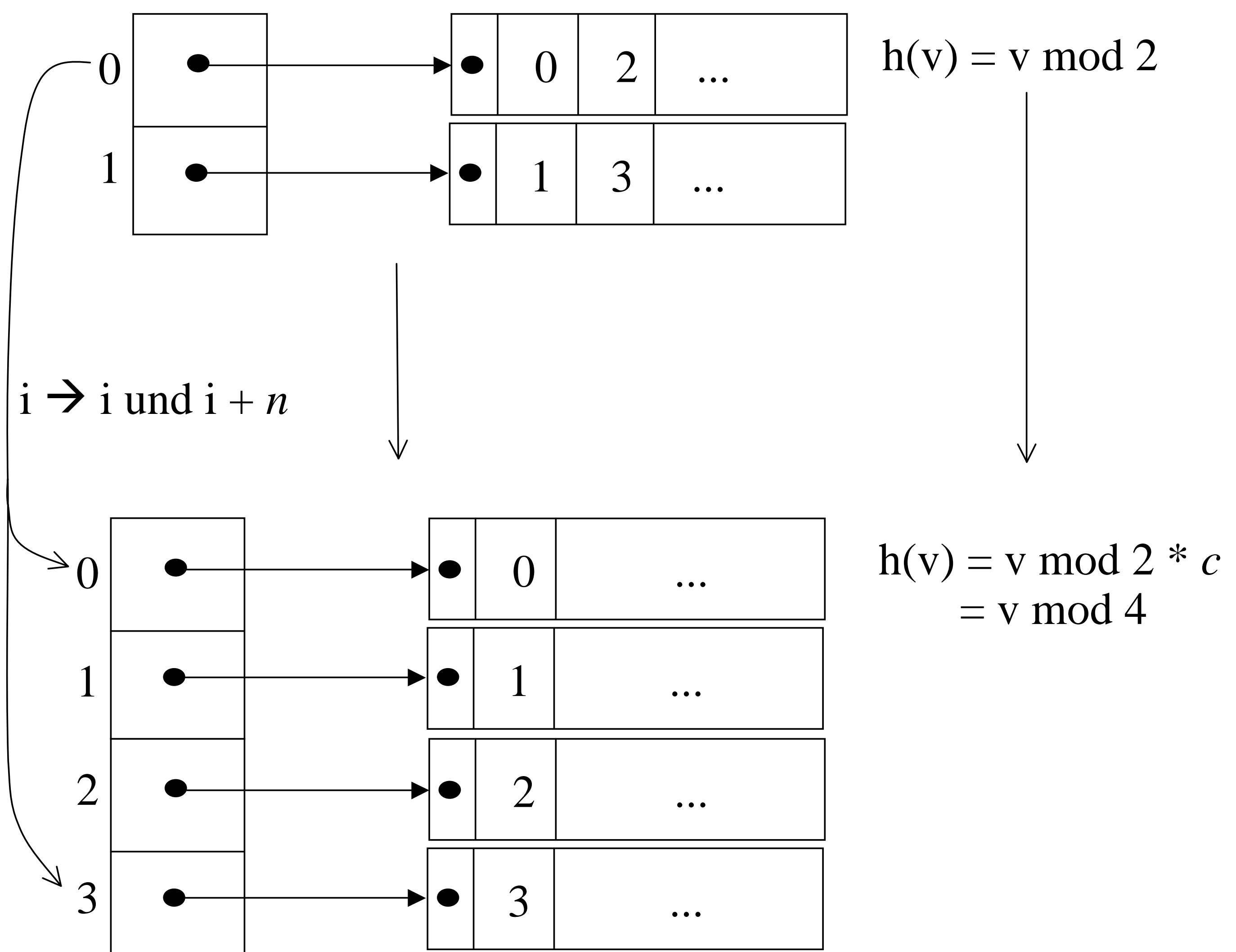


In diesem Fall ist eine Umstrukturierung (Reorganisation) der Hashtabelle erforderlich.

Beispiel zum Reorganisieren:

Ausgangssituation:

- Anzahl der Buckets: $n = 2$
- Faktor $c = 2$
- Daten: 0, 1, 2, 3, ...



Beispiele für häufige Anfragen in SQL (Bank):

BANK-KONTO

Kt.Nr	Name	Saldo	...
50	Otto Müller	6000	
100	Karsten Tolle	1000	
110	Otto Müller	80	
197	Peter Werner	5000	

Eine Hash-Struktur ist vorteilhaft für alle Anfragen gegen den **Schlüssel** einer Relation:

select * from BANK-KONTO where Kt.Nr = 100

... jedoch problematisch bei **Range-Queries** oder Anfragen nach **Nicht-Schlüssel-Attributen**:

select * from BANK-KONTO where Kt.Nr **>** 100

select * from BANK-KONTO where **Saldo** = 5000

Annahme: Es stehen 7 Buckets zur Verfügung:

h(50) = 50 mod 7 = 1

h(100) = 2

h(110) = 5

h(197) = 1