BITS Pilani
Hyderabad Campus

Database Systems (CSF212)

Dr.R.Gururaj
CS&IS Dept.

Acknowledgements

The content of the slides (both Text and Figures) are taken from the following source:

http://www.db-book.com/

Some changes are made as per the need.

Chapter 16: Hashing Schemes

Content

1. Introduction to Hashing
2. Internal hashing
3. Collision
4. External hashing
5. Static hashing
6. Dynamic hashing (Extendible and Linear hashing)

Introduction to Hashing

Hashing technique is an alternative to indexing, for fast retrieval of data records based on search key.

The search field is called as *hash field* of the file.

In most cases the hash field is also a key field of the file, in which case it is called as *hash key*.

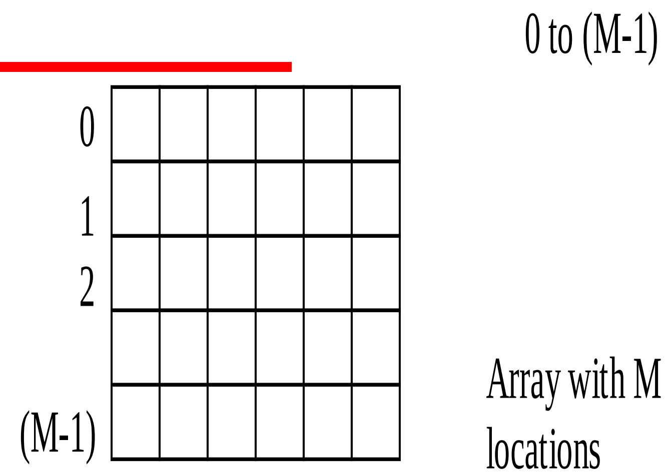
The basic idea of hashing is that a hash function *h*, when supplied a hash field value *K* of a record produces the address *B* of the disk block that contains the record with specified key value.

$$h : K \rightarrow \mathbf{B}$$

Hash function | Key value | Disk block address Or Location

For most records we require only one block access.

Internal Hashing



Used for internal files (on RAM).
Implemented as a hash table through use of an array of records.

The most common hash function used is $h(k)$ = K mod M
This gives the index of the location in the array.
For example- if M = 10 key value is 24
K mod M
$\Rightarrow$ 24 mod 10 = 4
Hence the record with key value 24 will be stored in 5th location of the array. If two or more records are hashed to same location it is called as *collision*. Then we need to find some other location for the new record. This process is known as *collision resolution*.

Methods for collision resolution

Open addressing: When collision occurs try with alternate cells until an empty cell is formed.
Chaining: for this various overflow locations are kept by extending the array by number of overflow positions. A pointer field is added to each record location. Collision is resolved by allocating an unused overflow position.
Multiple hashing: We apply a second hash function if the first hashing results in a collision.

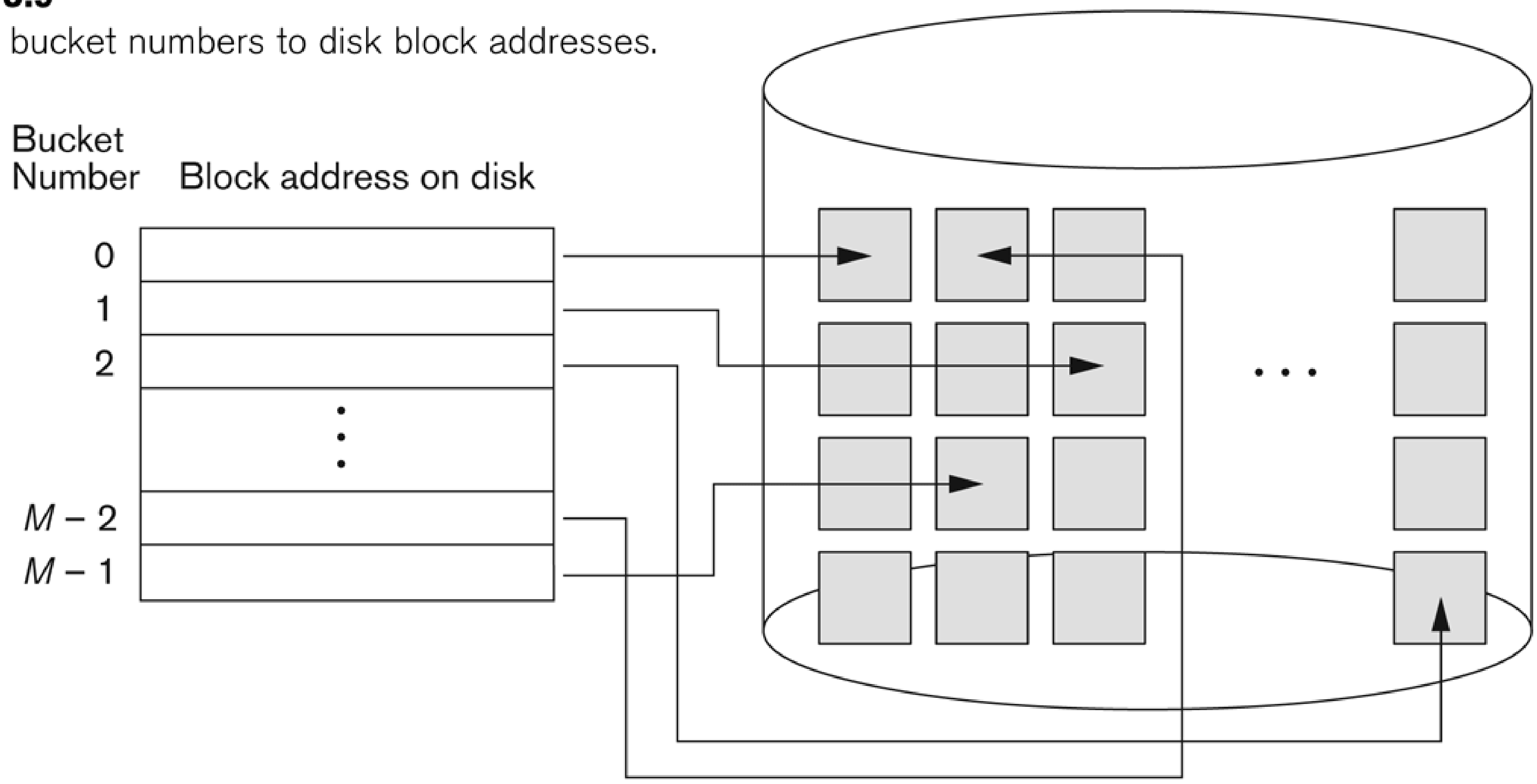
The goal of a good hashing function is to distribute the records uniformly over the address space so as to minimize collisions while not leaving many unused locations.

External Hashing

External hashing is used for Disk files.
And best suited for Database systems.
The target address space is divided into buckets, each of which can hold multiple records.
Sometime it could be a cluster of blocks.

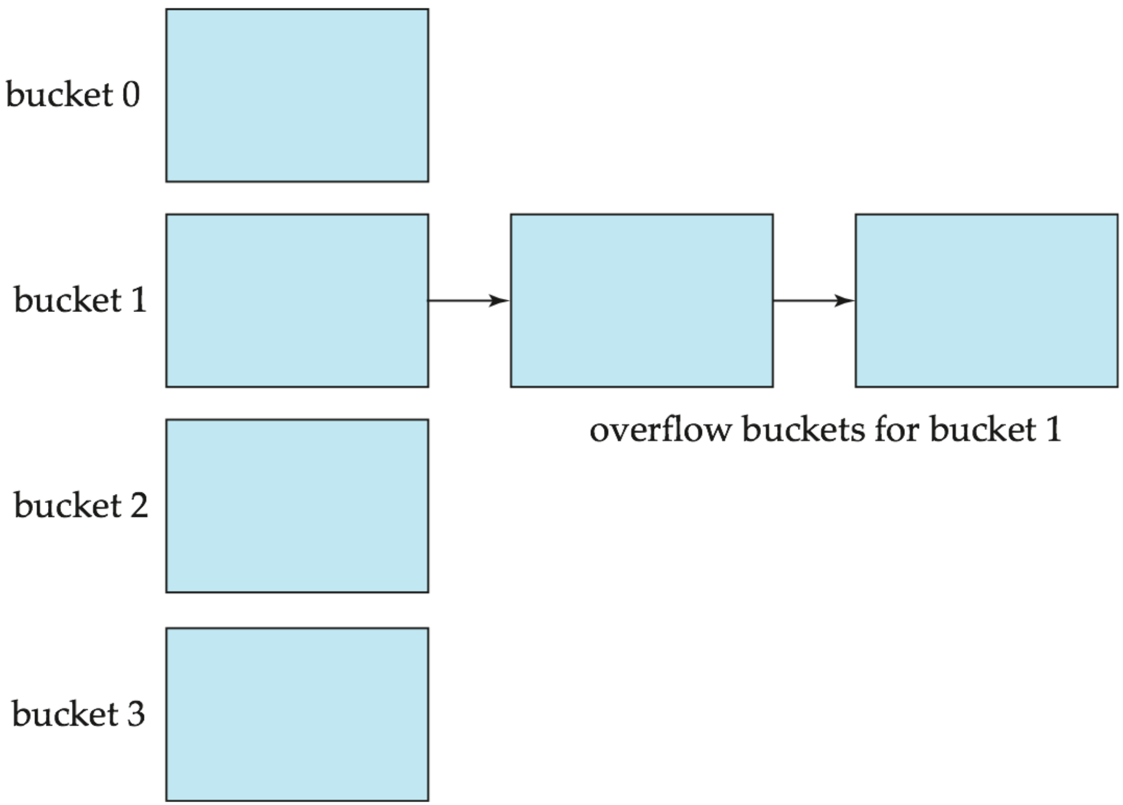
A hashing function maps a key to relative bucket number, rather than absolute block address.
A table maintained in the header will map the bucket number to physical block address.
Once the disk block is known, the actual search for the record within the block is carried out in main memory buffer.

Figure 13.9
Matching bucket numbers to disk block addresses.



- **Overflow chaining** – the overflow buckets of a given bucket are chained together in a linked list. Best suited for Databases systems.

Chaining


bucket 0
bucket 1
bucket 2
bucket 3
overflow buckets for bucket 1

Handling overflows in Static External Hashing

Keys: 340, 321, 460, 399, 22, 72, 761, 91, 522, 89, 182, 981, 652

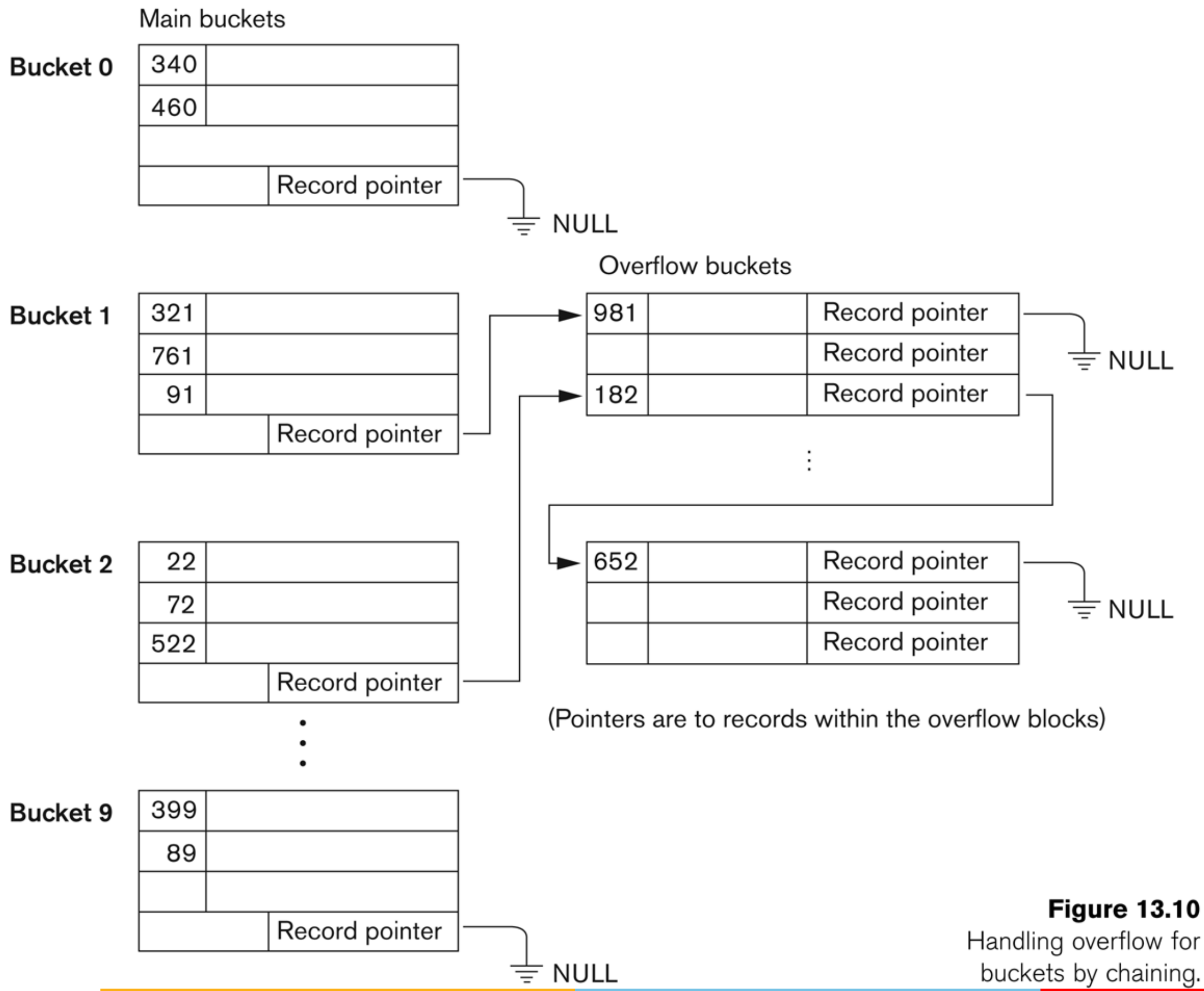


Figure 13.10
Handling overflow for buckets by chaining.

The above scheme is called as *static hashing* because- the number of buckets allocated is fixed. This is a big constraint for files that are dynamic.

When a bucket is filled to capacity and if the new record is hashed on to the same bucket, then chaining is adopted, where a pointer is maintained in each bucket to a linked list of overflow records for the bucket.

Here the pointers are record pointers which include both block address and a relative record position with in that block.

Deficiencies of Static Hashing

- In static hashing, function *h* maps search-key values to a fixed set of *B* of bucket addresses. Databases grow or shrink with time. Hash function is fixed. Original number of buckets is fixed.
  - If initial number of buckets is too small, and file grows, performance will degrade due to too much overflows/chaining.
  - If space is allocated for anticipated growth, a significant amount of space will be wasted initially (and buckets will be underutilized).
  - If database shrinks, again space will be wasted.

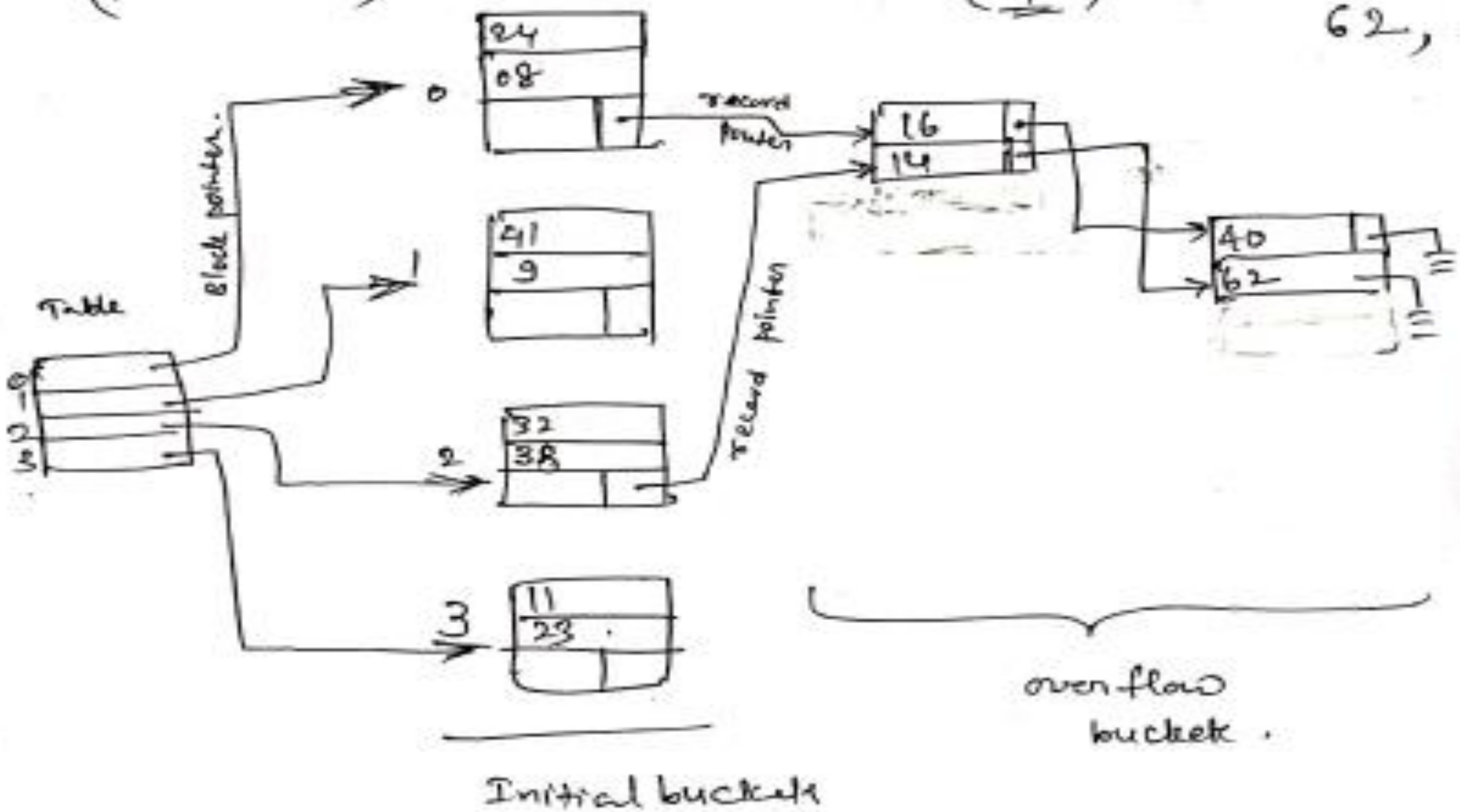
example

Let us look at the following.

Keys: 24, 32, 41, 38, 08, 9, 16, 11, 14, 40, 62, 23

h: (K MOD 4).

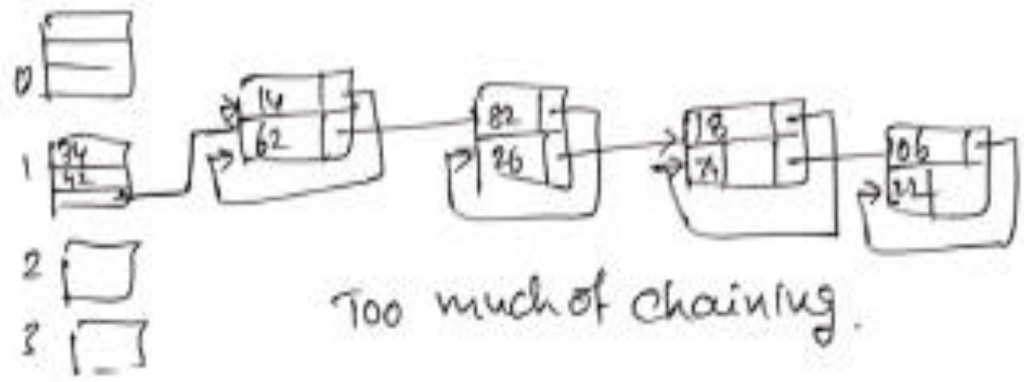
Bucket can take 2 records. (Bfr.)



Static Hashing Scheme

Now

Imagine the situation: 34, 42, 14, 62, 82, 26, 18, 74, 06, 22



Too much of chaining.

Dynamic Hashing

This scheme allows us to expand or shrink the hash address space dynamically.
We study the following schemes.

1. Extendible Hashing

2. Linear Hashing

Extendible Hashing

The first technique is called as *extendible hashing*. This scheme stores a directory structure in addition to the file. This access structure is based on the result of the hash function to the search field.

Each result of applying the hash function is a nonnegative integer and hence can be represented with a binary pattern.

This we call it as *hash value* of the record.

Records are distributed among the buckets based on the values of the leading bits in their hash value.

The major advantage of extendible hashing is that performance does not degrade because of chaining, as the file grows as we have seen in static hashing.

In extendible hashing no additional space is wasted towards the allocations for future growth, but additional buckets can be allocated dynamically as needed.

The only overhead in this scheme is that a directory structure needs to be searched before the buckets are accessed.

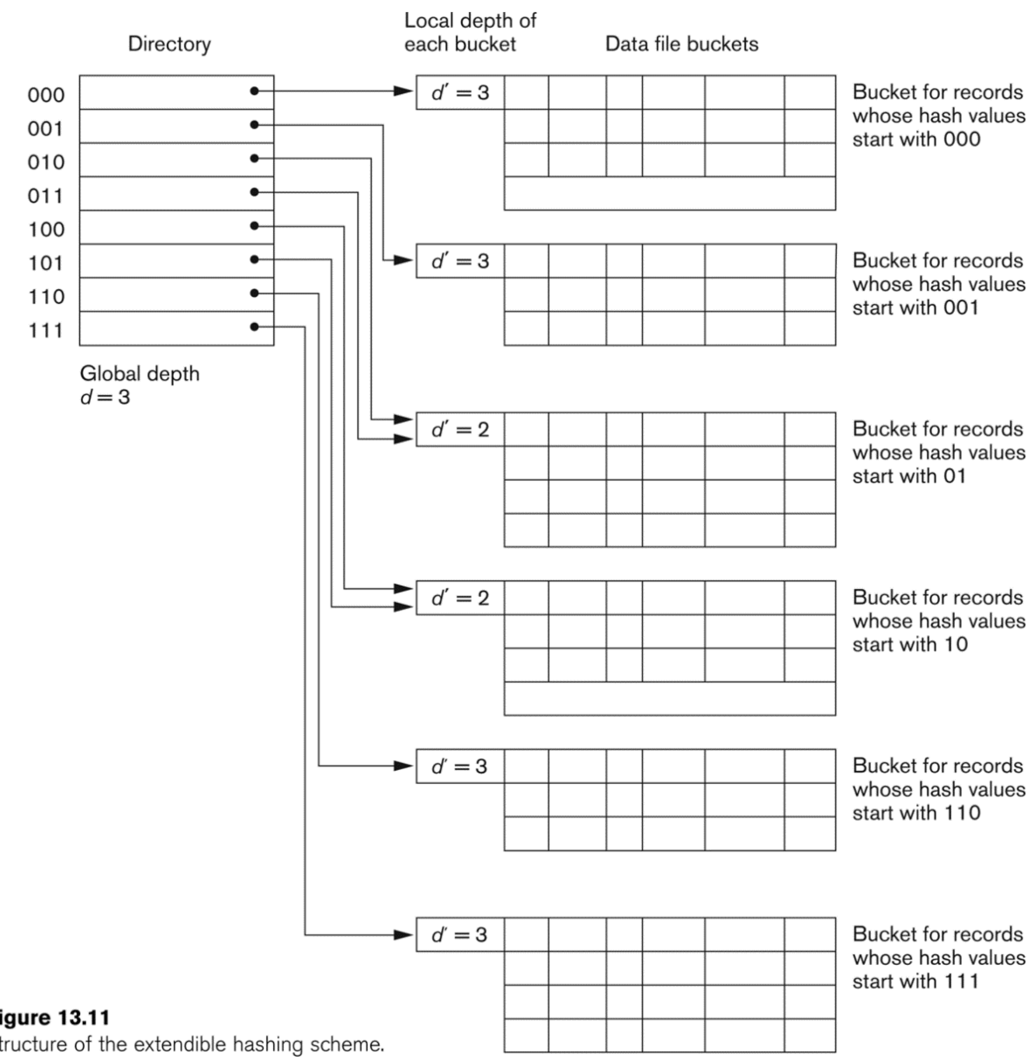


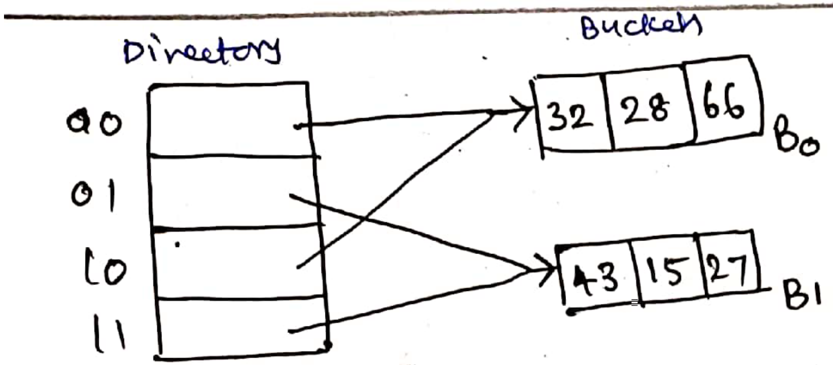
Figure 13.11
Structure of the extendible hashing scheme.

Example- 1
(Extendible hashing)

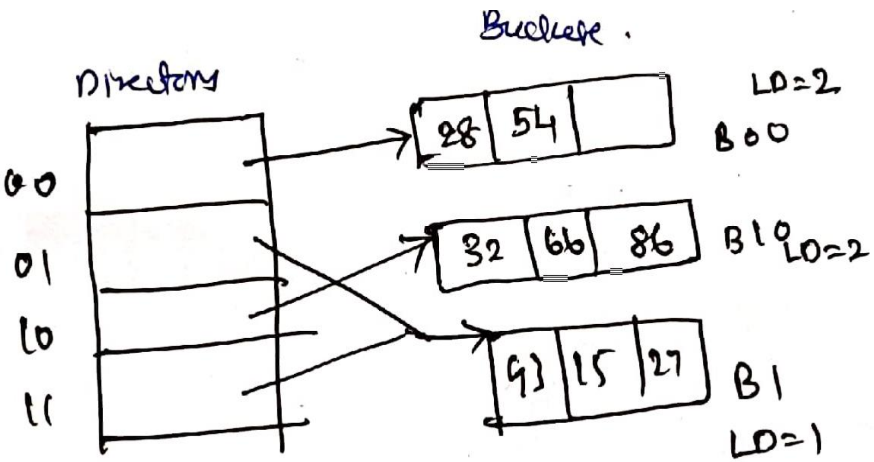
Assume that we need to load some records of the relation *EMP* into *expandable hashfiles* based on *extendible hashing technique*. Records are inserted into the file with following key field values: 32, 28, 43, 15, 66, 27, 86, 54, 35, 98, 72. The order of insertion should be same as the above. Each bucket is one disk block and each block can store maximum of three records. Show the structure of the directory at each step, and the local and global depths. Use the hash function *h(K)=K mod 10.* Start with Global depth-2, Local depth-1. Consider LSB of the hash value for directory entries.

Keys	32	28	43	15	66	27	86	54	35	98	72
KMod10	2	8	3	5	6	7	6	4	5	8	2
Hash Value	0010	1000	0011	0101	0110	0111	0110	0100	0101	1000	0010

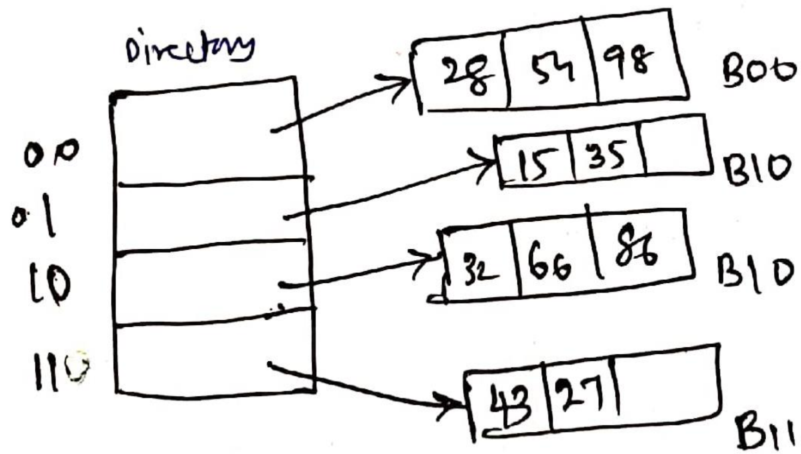
We start with GD = 2 LD=1 and Consider LSB. Bfr = 3.



When we try to insert 86 it is mapped to B0 But it is full Hence split B0, ie increase the LD of B0 to 2



But when we insert 35 it is going into = B1 it is full hence split B1 i.e increase the LD of B1 = 2

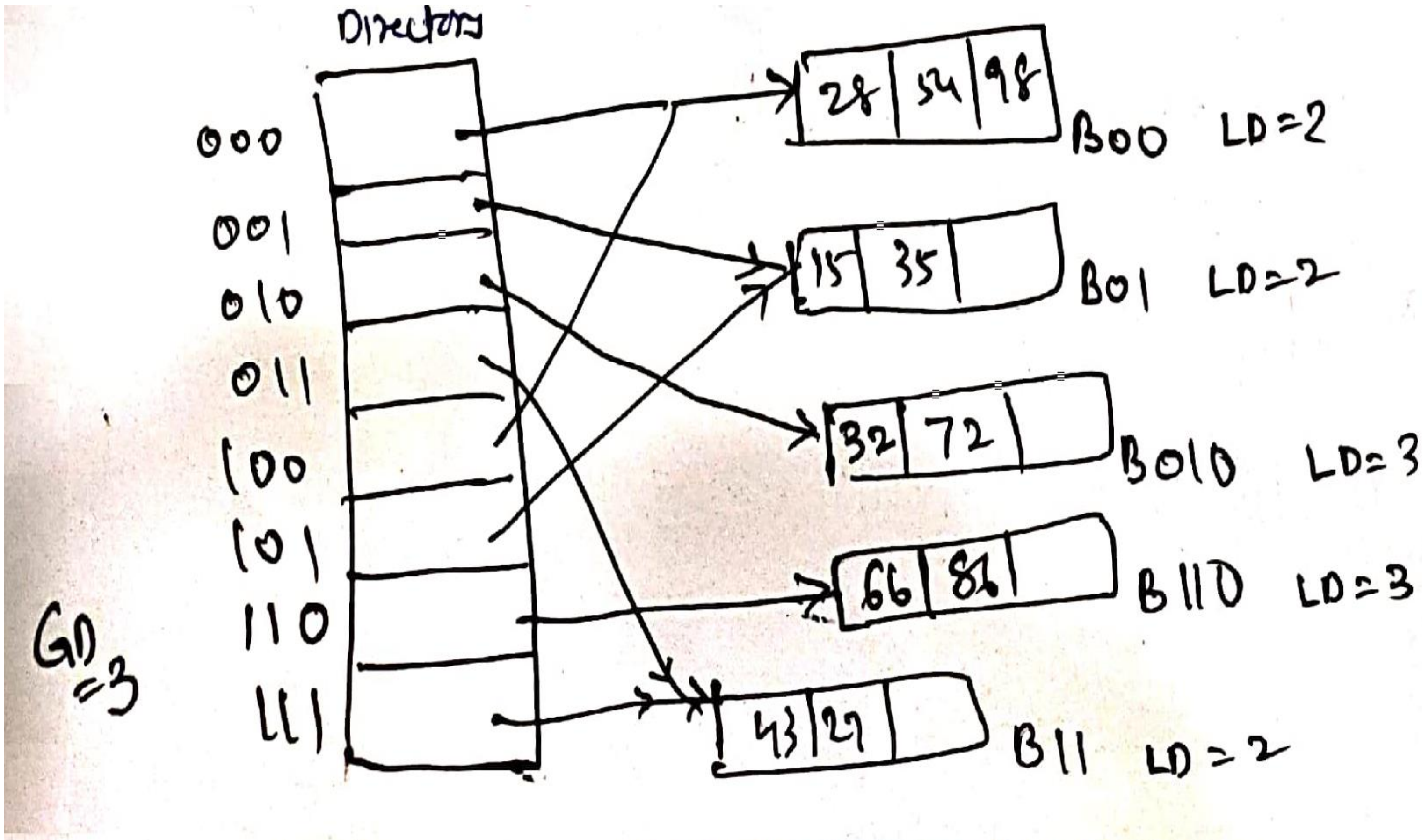


Now,

when we insert 72 → B10 it is full hence increase the LD = 2

But LD can't be greater than GD

Hence first increase GD=3.



Directory
000
001
010
011
100
101
110
111
GD=3
28
54
98
B00 LD=2
15
35
B01 LD=2
32
72
B010 LD=3
66
86
B110 LD=3
43
27
B11 LD=2

Linear Hashing

In the second scheme called *linear hashing*, no directory structure is used. Here instead of one hash function, multiple hash functions are used. When collision occurs with one hash function, the bucket that overflows is split in to two and the records in the original bucket are distributed among two buckets using the next hash function $h_{(i+1)}(k)$. Hence we have multiple hash functions.

Example-1 (Linear Hashing)

Assume that we use ***Linear hashing*** technique in some situation and we use the hash functions h0, h1, h2, ... as (K mod 2), (K mod 4), (K mod 8) and so on. Assume that a bucket(one block) can accommodate 2 records. Now insert the records with following keys in same order and show the dynamic structure of the hashing scheme after each insertion.

Keys to be inserted are: 14, 21, 7, 24, 6, 22, 5, 19. Note that split occurs when the *fileload factor* (*f*) exceeds 0.7.

Q3. Linear Hashing Scheme

$h_0 = K \bmod 2$ $\quad h_1 = K \bmod 4$ $\quad h_2 = K \bmod 8$

Keys :	14	21	7	24	6	22	5	19
K mod 2	0	1	1	0	0	0	1	1
K mod 4	2	1	3	0	2	2	1	3
K mod 8	6	5	7	0	6	6	5	3

f = file loadfactor $= 0.7 = \dfrac{\text{no. records}}{\text{no. blocks} \times Bfr}$ $\quad Bfr = 2$.

$n = 0$

insert 14, 21, 7

0	14	
1	21	

When we insert 7 the $f = \dfrac{3}{2 \times 2} = 0.7$

Hence Split bucket 0

$n = 1$

0	24	
1	21	7
2	14	

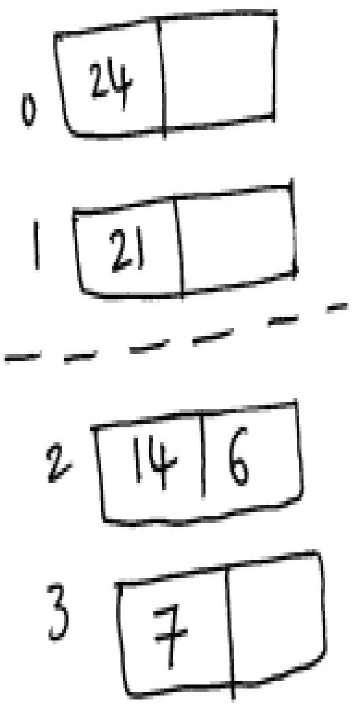
insert 24

$f = \dfrac{4}{6} = 0.66$ It is ok

Insert 6 $f = \frac{5}{6} > 0.7$

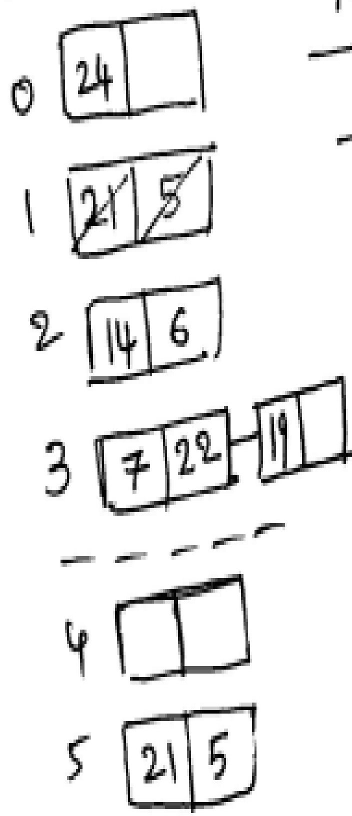
Hence Split Bucket 1

then $n = 0$



Insert 22 $f = \frac{6}{8} > 0.7$ Hence Split

$n = 1$



Insert 5

$f = \frac{7}{10} = 0.70$

no problem

Insert 19

$f = \frac{8}{10} > 0.7$

Split bucket 1

Shift 21, 5 to Bucket 5

$n = 2$

Example(Linear Hashing)

Keys:	8	7	13	5	22	11	42	21	16	31
h_0: K Mod 2	0	1	1	1	0	1	0	1	0	1
h_1: K Mod 4	0	3	1	1	2	3	2	1	0	3
h_2: K Mod 8	0	7	5	5	6	3	2	5	0	7

Bfr = 2,

Split Criterion:
when file load factor (f) > 0.75

NOTE: Do not consider overflow buckets while computing f.

First we go with h_1 : K Mod 2. and n = 0
Hence we need only 2 buckets 0, 1

Insert 8

00	8	
01		

$f = \frac{1}{4} = 0.25$

Insert 7

00	8	
01	7	

$f = \frac{2}{4} = 0.5$

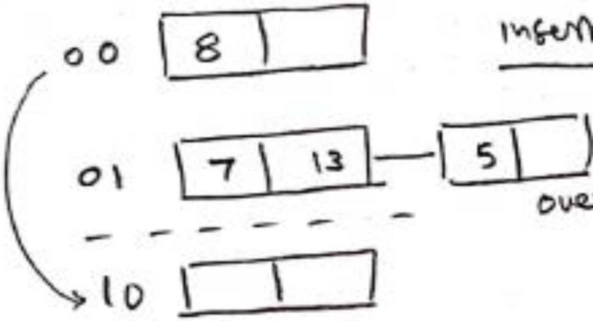
Insert 13:

00	8	
01	7	13

$f = \frac{3}{4} = 0.75$ ok ✓

If we try to insert 5
$f = \frac{4}{4} = 1 > 0.75$
Hence split the bucket-0
Since n = 0.
and make n = n+1
= 1

Redistribute keys in bucket 0 using h_1 : K mod 4



insert 5 using h_0 . K mod 2

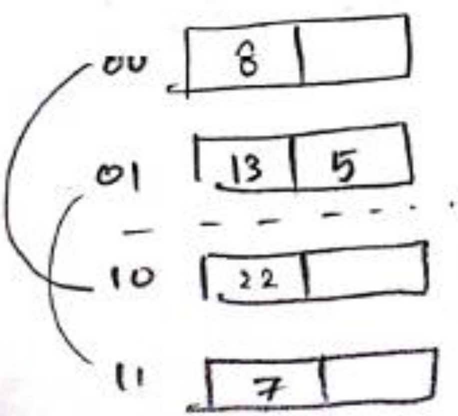
now $f = \frac{4}{6} = 0.66 < 0.75$

Now if we insert 22 f becomes $\frac{5}{6} > 0.75$ hence split

Now the bucket '01' will get split image. Then $n = n+1$
$= 0$

Redistribute keys in 01 among 01 and 11 using h_1 . K mod 4

insert 22
using K mod 4.



$f = \frac{5}{8} < 0.75$ ✓

By now both buckets 00, 01 in the frontward got their split images & now onwards we can apply 'K mod 4 directly

Now insert 11 → 11 Mod 4 = 3 = '11'

$f = \frac{6}{8} = 0.75$ OK.

00	8	
01	13	5
10	22	
11	7	11

Now n=0

Now if we try to insert 42

$f = \frac{7}{8} > 0.75$ Hence split

00 → it will get '100' as its image then redistribute keys in '00' between '00' and '100' - using k mod 8

$n = n+1 = 1$

$f = \frac{7}{10} = 0.70$ ✓

insert 42 42 mod 4 = 2 = '10'

000	8	
001	13	5
010	22	42
011	7	11
100		

Now if we try to insert 21

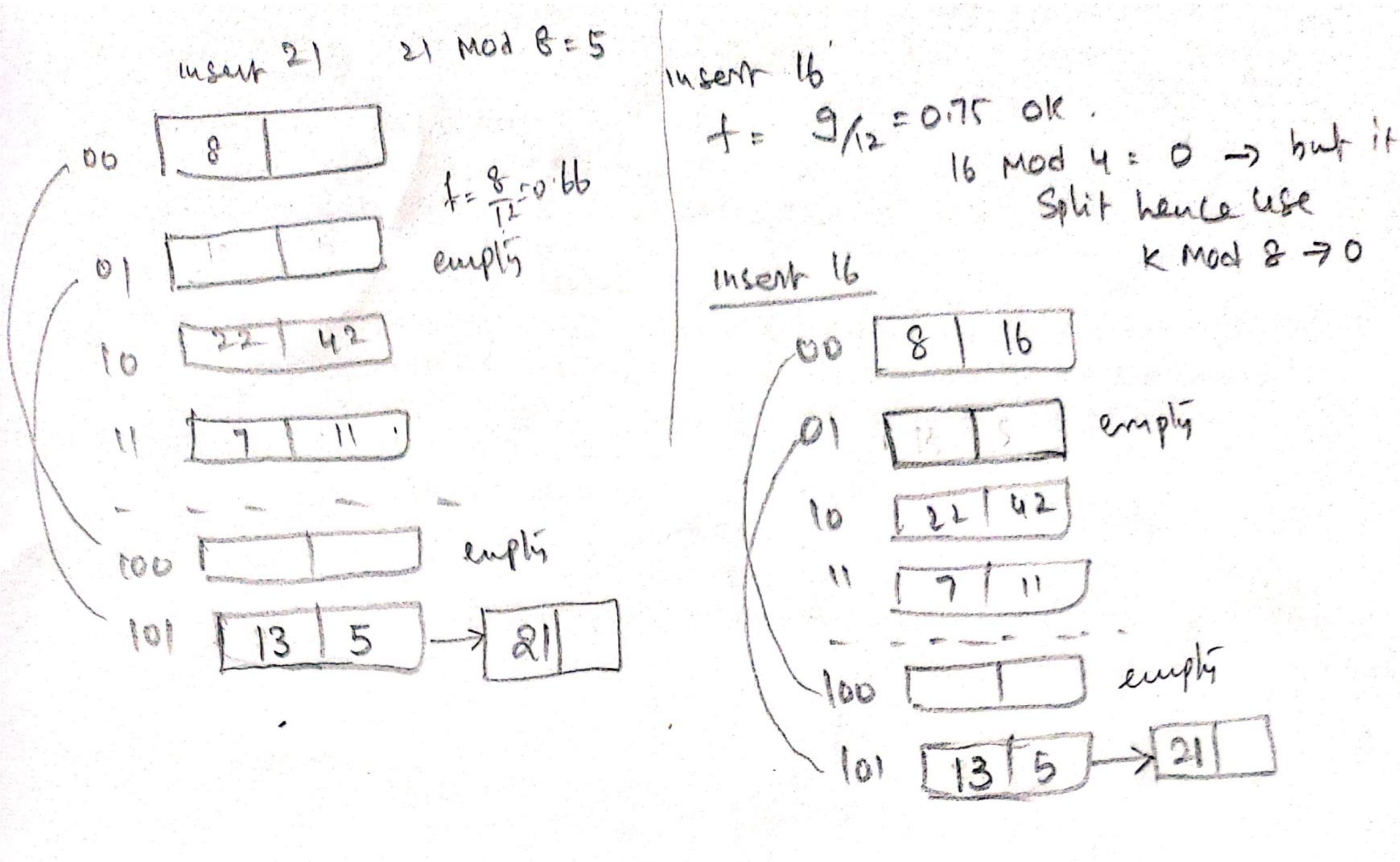
$f = \frac{8}{10} = 0.8 > 0.75$

Hence split 01 → 001, 101

$n = n+1 = 2$

distribute keys in 001 between 001 and 101

$f = \frac{8}{12} = 0.66$ ✓

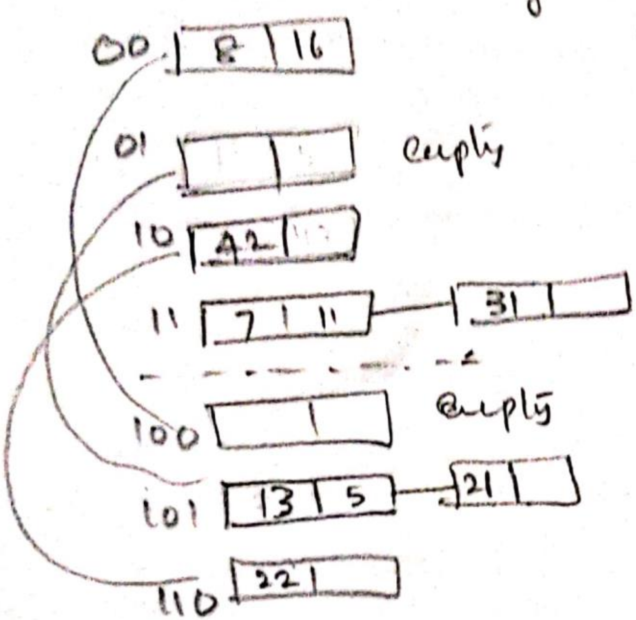

insert 21
21 Mod 8 = 5
00
8
f = 8/12 = 0.66
01
empty
10
22
42
11
7
11
100
empty
101
13
5
21
insert 16
f = 9/12 = 0.75 ok.
16 Mod 4 = 0 → but it
split hence use
k Mod 8 → 0
insert 16
00
8
16
01
empty
10
22
42
11
7
11
100
empty
101
13
5
21

Now insert 31 (tenth record)

$f = \frac{10}{12} > 0.75$

Hence split bucket '10' (2) since $n = 2$.
and make $n = 3$ and redistribute keys in '10' (22, 42) using K mod 8
between 010 and 110 $\quad f = \frac{10}{16} = 0.71$ ✓

insert 31 using K mod 4 = 3. since 3 ('11') is not split we must insert 31 into '11' only



This is the final status of the scheme.

Summary

- ✓ What is hashing
- ✓ Internal hashing
- ✓ External hashing
- ✓ What is static external hashing
- ✓ What is dynamic hashing
- ✓ How Extendible and Linear hashing techniques work