Hash-Based Indexing

Motivation

- The primary goal is to locate the desired record in a single access of disk.
  - Sequential search: O(N)
  - B+ trees: $O(\log_k N)$
  - Hashing: O(1), actually a **one disk-access** method
- In hashing, the key of a record is transformed into an address and the record is stored at that address.
- Hash-based indexes are the best for **equality selections**. **Can not** support **range searches**.
- Static and dynamic hashing techniques exist.

Hash-based Index

- Data entries are kept in ***buckets*** (an abstract term)
- Each **bucket** is a collection of **one primary page** and **zero or more overflow pages**.
- Given a search key value, k, we can find the bucket where the **data entry k*** is stored as follows:
  - Use a *hash function*, denoted by *h*
  - The value of *h*(key) is the address for the desired bucket.
  - *h*(key) should distribute the search key values *uniformly* over the collection of buckets

Hash Functions

- It is always a bad idea to use field values themselves for hashing, as real data is never uniformly distributed.
  - Hash function is for transforming a non-uniform distribution to a (nearly) uniform one.
- **Key mod N:** N is the number of buckets, better if it is prime.
- **Folding:** e.g. 123|456|789: add them and take mod.
- **Truncation:** e.g. 123456789 map to a table of 1000 addresses by picking 3 digits of the key.
- **Squaring:** Square the key and then truncate
- **Radix conversion:** e.g. 1 2 3 4 treat it to be base 11, truncate if necessary.
- Hash functions **never preserve order**! → Hence, hash based indexes don't help for range search.

Static Hashing

- In a file organized using static hashing, the data entries k* are stored in the buckets in the **primary area** and, possibly, in the **overflow area**.
- **Primary Area**: # primary pages fixed, **allocated sequentially**, **never de-allocated**; (say **M** buckets).
  - A hash function works on the *search key* field, and maps keys to values from **0** to **M-1**
  - A simple hash function: **h**(*key*) = f(*key*) mod **M**
- **Overflow area**: disjoint from the primary area. It keeps overflow pages which hold data entries whose key maps to a full bucket.
  - **Chaining:** The address of an overflow page is added to the **overflow chain** of the full bucket

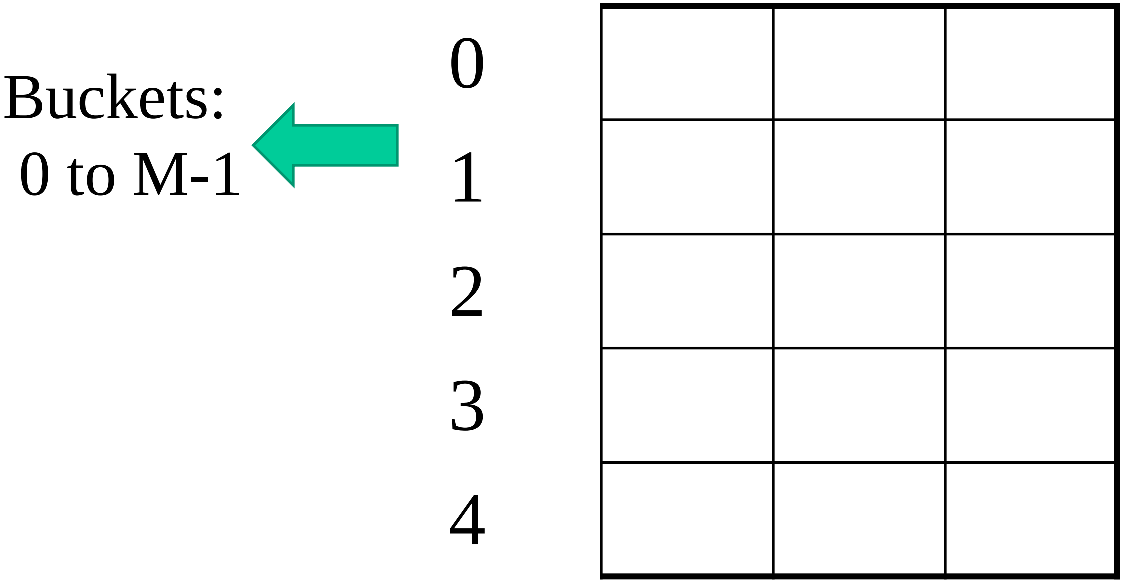
Static Hashing

- **Bucket factor** (Bkfr) is the number of data entries (recall: Alternative 1 or 2 applies) that can be held at a bucket.
- ***Collision*** does not cause a problem as long as there is still room in the mapped bucket. ***Overflow*** occurs during insertion when a data entry is hashed to the bucket that is already full.

Example

- Assume **f(key) = key**. Let **M = 5**. So, **h(key) = key mod 5**
- Bucket factor (Bkfr) = 3 data entries

Insert: 12*, 35*, 44*, 60*, 6*, 46*,57*,33*,62* ,17*

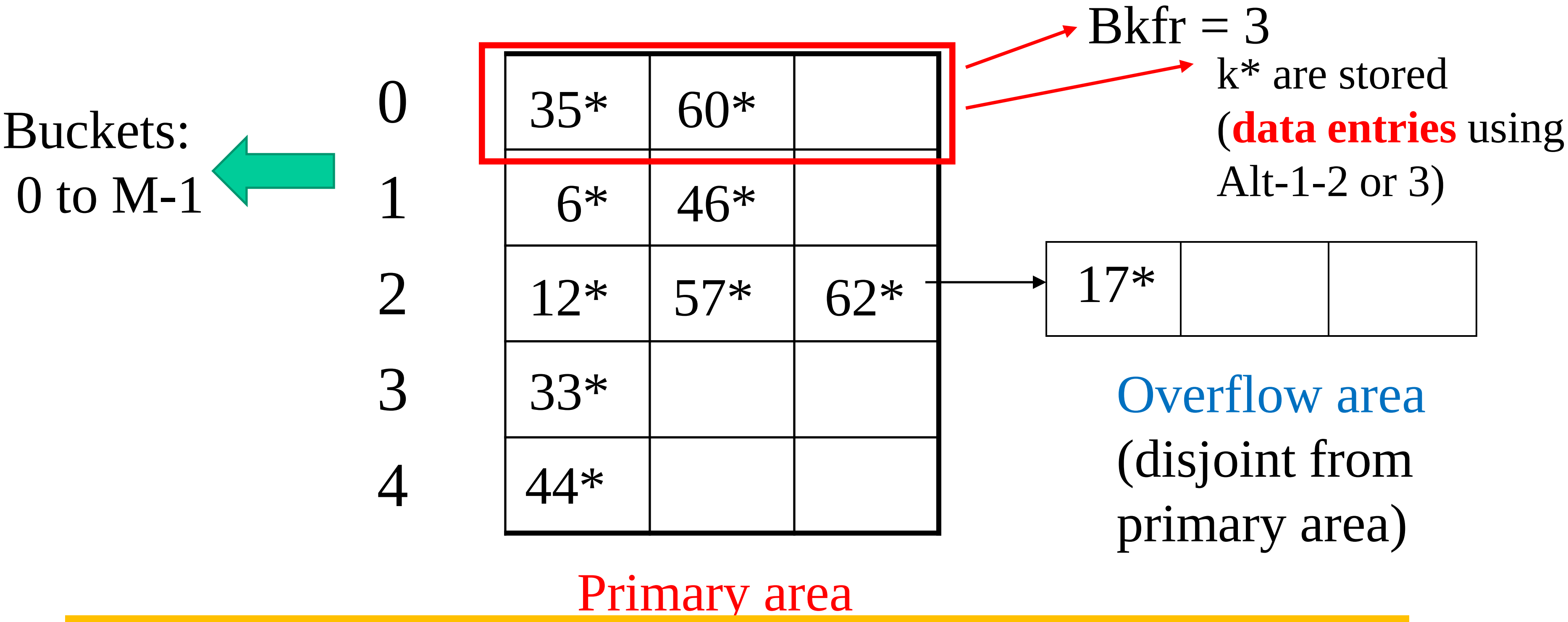

Buckets:
0 to M-1

Primary area
(primary bucket pages)

Example

- Assume **f(key) = key**. Let **M = 5**. So, **h(key) = key mod 5**
- Bucket factor (Bkfr) = 3 data entries

Insert: 12*, 35*, 44*, 60*, 6*, 46*,57*,33*,62* ,17*



Bkfr = 3

k* are stored (**data entries** using Alt-1-2 or 3)

Buckets:
0 to M-1

0	35*	60*	
1	6*	46*	
2	12*	57*	62*
3	33*		
4	44*		

17*		

Overflow area (disjoint from primary area)

Primary area

If no overflow, fetching a record costs only one disk access!

Load Factor (Packing density)

- To limit the amount of overflow we allocate more space to the primary area than we need (i.e. the primary area will be, say, 70% full)

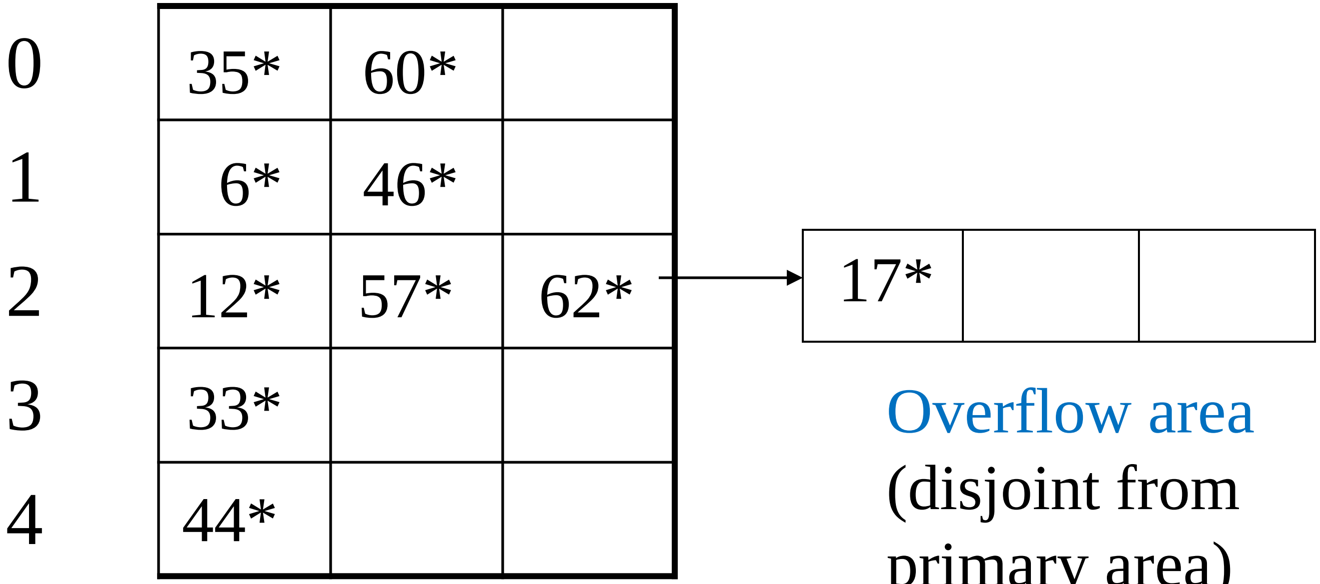
in Primary Area + # in Overflow Area

- Load Factor = $\dfrac{\text{\# of data entries in the file}}{\text{\# of spaces in primary area}}$

$$\Rightarrow Lf = \frac{n}{M * Bkfr}$$

Example: What is Load Factor?

- $$LF = \frac{n}{M * Bkfr} = \frac{\mathbf{9}+\mathbf{1} = 10 \text{ data entries}}{\mathbf{5 * 3 \text{ spaces}}} = 2/3$$
-



0	35*	60*				
1	6*	46*				
2	12*	57*	62*	17*		
3	33*					
4	44*					

Primary area
(primary bucket pages)

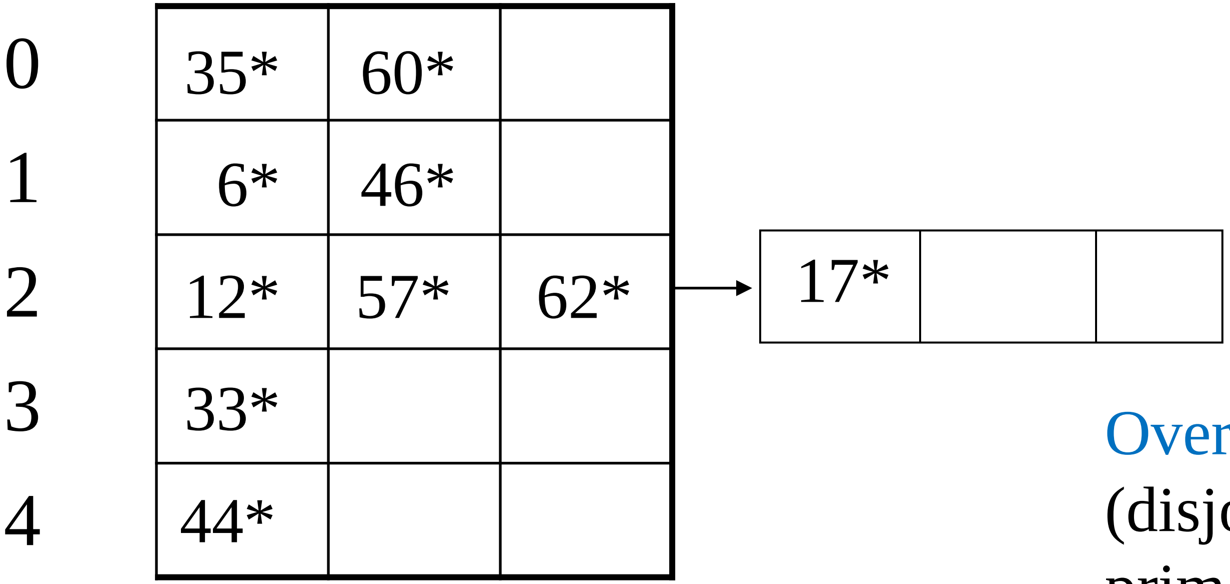
Overflow area
(disjoint from
primary area)

Effects of Lf and Bkfr

- Performance can be enhanced by the choice of Bkfr and load factor.
- In general, a smaller load factor means
  - less overflow and a faster fetch time;
  - but more wasted space.
- A larger Bkfr means
  - less overflow in general,
  - but slower fetch.

Insertion

- Insertion: New data entries are inserted at the end of the chain.
- Insert: 22*, 37*, 52*, 97*



0	35*	60*	
1	6*	46*	
2	12*	57*	62*
3	33*		
4	44*		

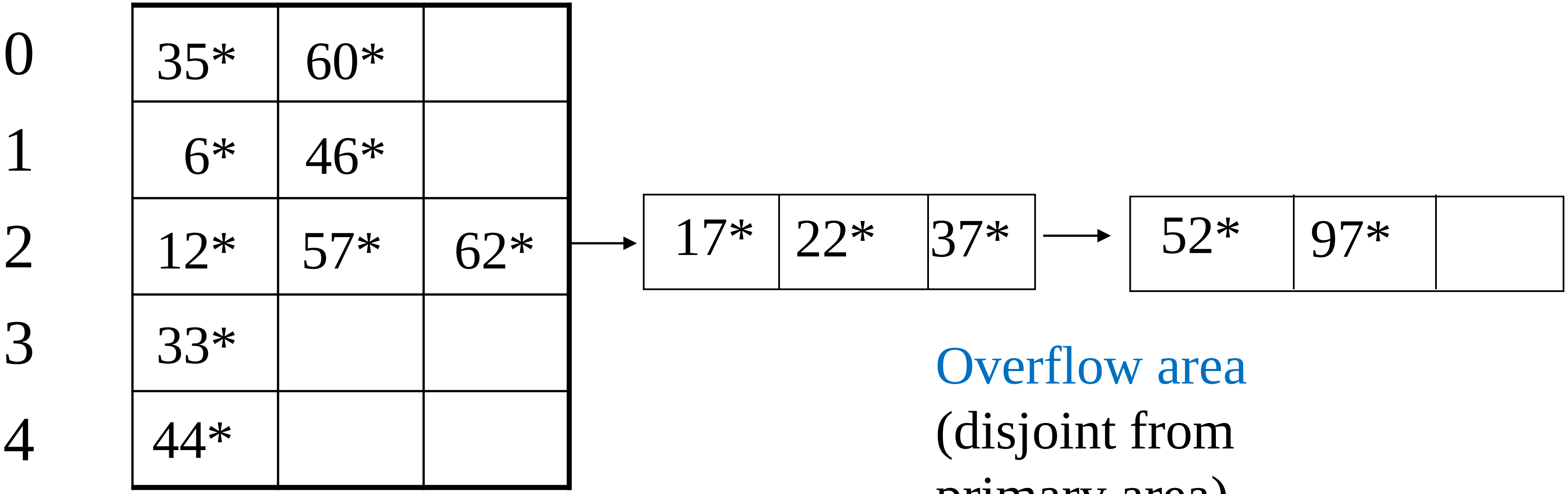
17*		

Primary area
(primary bucket pages)

Overflow area
(disjoint from primary area)

Insertion

- **Insertion:** New data entries are inserted at the end of the chain.
- Insert: 22*, 37*, 52*, 97*



0	35*	60*	
1	6*	46*	
2	12*	57*	62*
3	33*		
4	44*		

2 → 17* 22* 37* → 52* 97*

Overflow area
(disjoint from primary area)

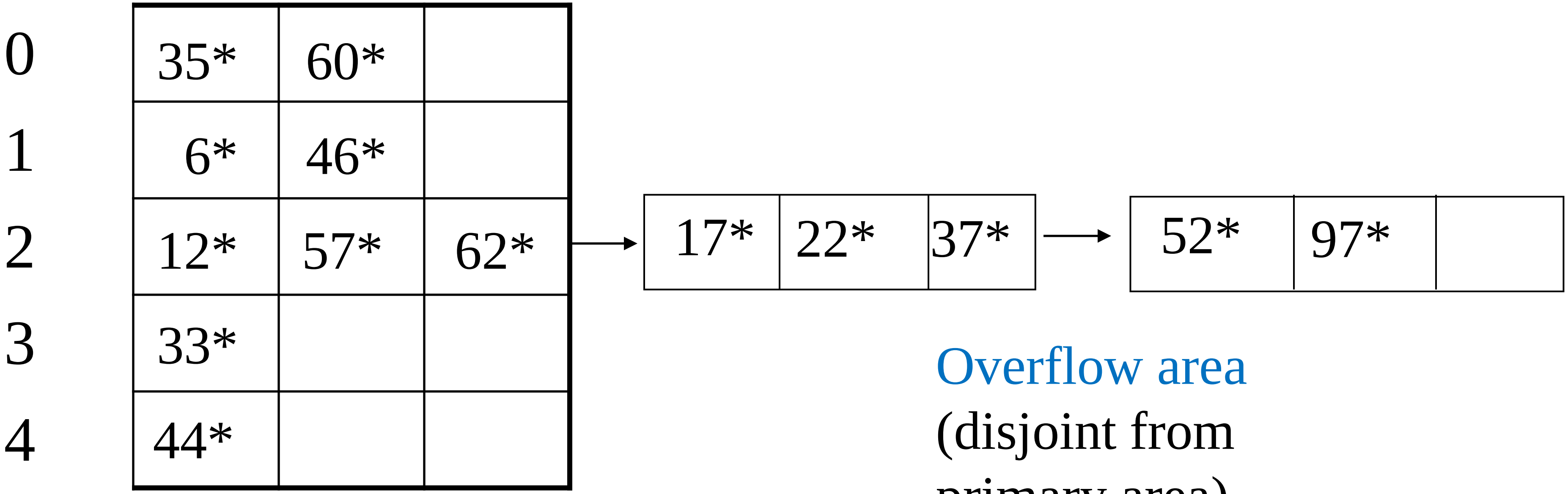
Primary area
(primary bucket pages)

Deletion

- Deletion: Two ways are possible:
  1. Mark the data entry to be deleted
  2. Consolidate sparse buckets when deleting data entries.
  - In the 2nd approach:
    - When a data entry is deleted, fill its place with the last data entry in the chain of the current bucket.
    - Deallocate the last bucket when it becomes empty.

Deletion

- When a data entry is deleted, fill its place with the last data entry in the chain of the current bucket.
- Deallocate the last bucket when it becomes empty.
- Delete: 57*, then 22*



0	35*	60*	
1	6*	46*	
2	12*	57*	62*
3	33*		
4	44*		

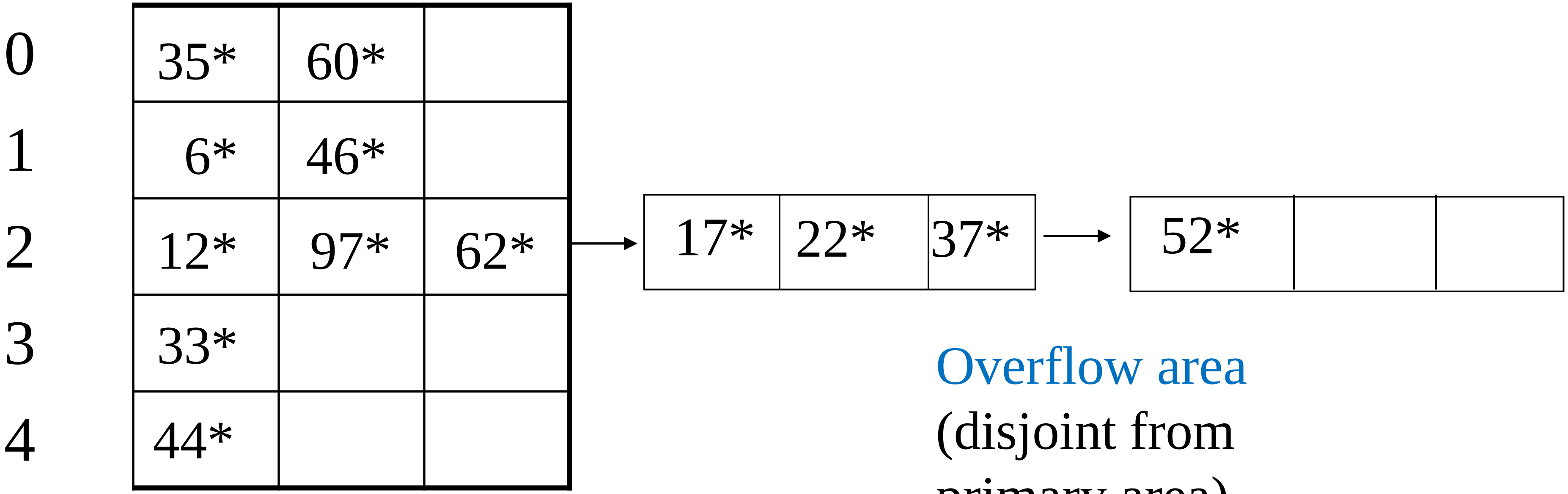
17* 22* 37* → 52* 97*

Primary area
(primary bucket pages)

Overflow area
(disjoint from
primary area)

Deletion

- This is the hash table right after 57* is deleted
- Then 22*



0	35*	60*	
1	6*	46*	
2	12*	97*	62*
3	33*		
4	44*		

17*	22*	37*

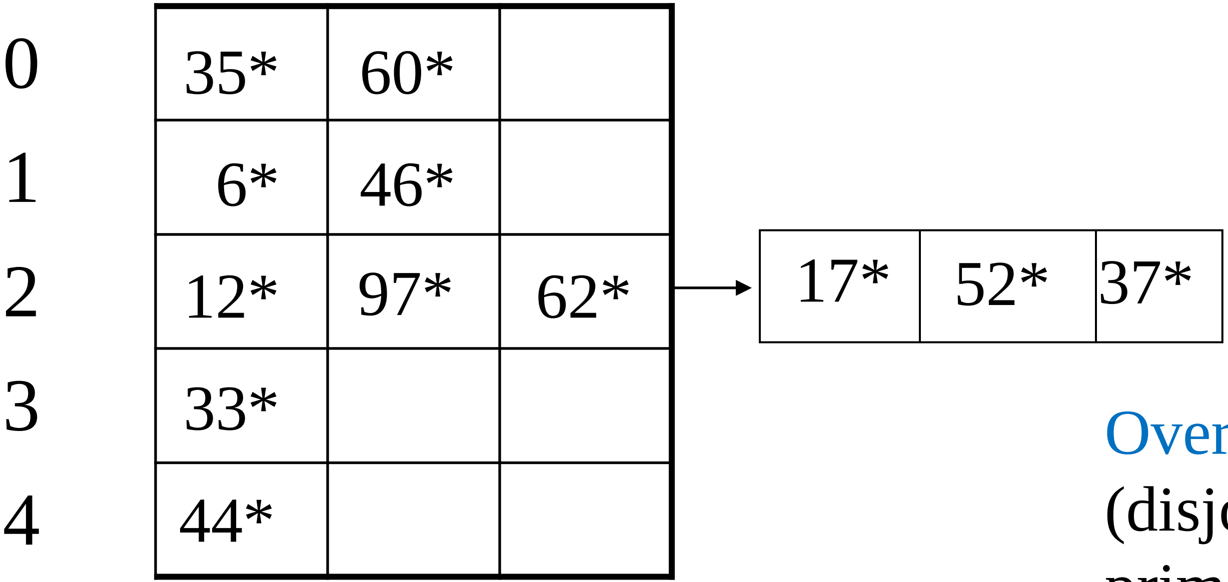
52*		

Overflow area
(disjoint from
primary area)

Primary area
(primary bucket pages)

Deletion

- Right after deleting 57*, and 22*



0	35*	60*	
1	6*	46*	
2	12*	97*	62*
3	33*		
4	44*		

17*	52*	37*

Primary area
(primary bucket pages)

Overflow area
(disjoint from
primary area)

Problem of Static Hashing

- The main problem with **static** hashing: **the number of buckets is fixed**:
  - **Long overflow chains** can develop and degrade performance. → would require **re-organization** at some point
  - On the other hand, if a file shrinks greatly, a lot of bucket space will be wasted.

- There are some other hashing techniques that allow **dynamically growing and shrinking** hash index. These include:
  - extendible hashing
  - linear hashing

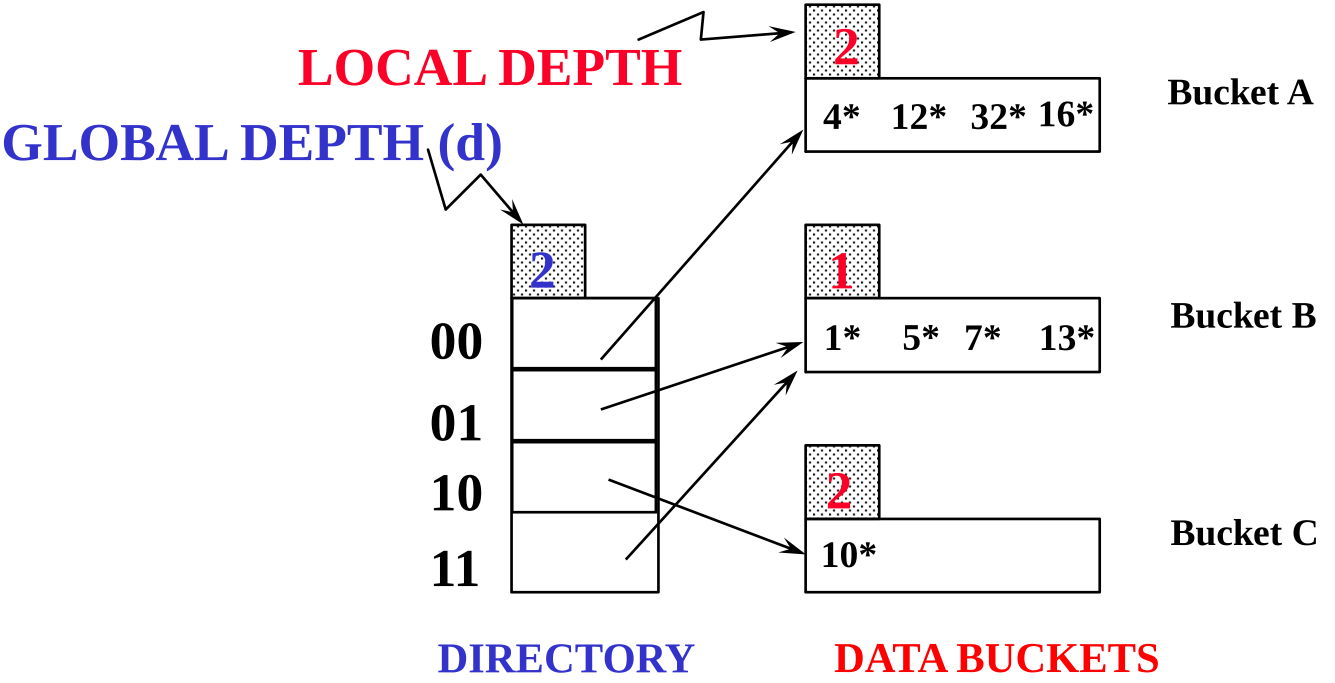
Extendible Hashing

Extendible Hashing

- **Basic Idea:**
  - No overflow buckets
  - Instead add a level of indirection
- Use **directory of pointers** to buckets
- Double # of buckets by doubling the directory
  - Directory much smaller than file, so doubling it is much cheaper.
- Split only the bucket that just overflowed!
  - Adjust the hash function

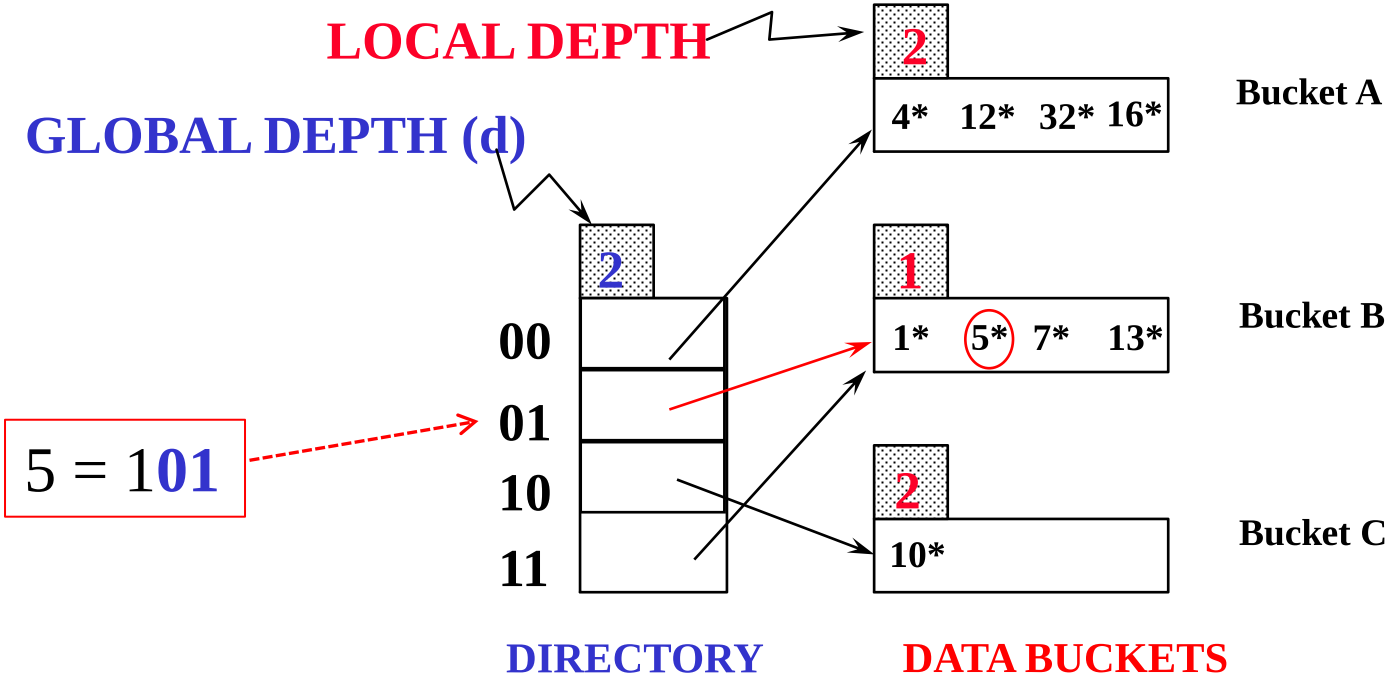
General Structure

- Directory is an array of size 4 with each element being a pointer to a data bucket
  - 4 entries in the directory so 2 bits needed.
- Bucket for data entry *k* with key k* is in the element with index = \`*global depth*' ***least significant*** bits of h(*k*);



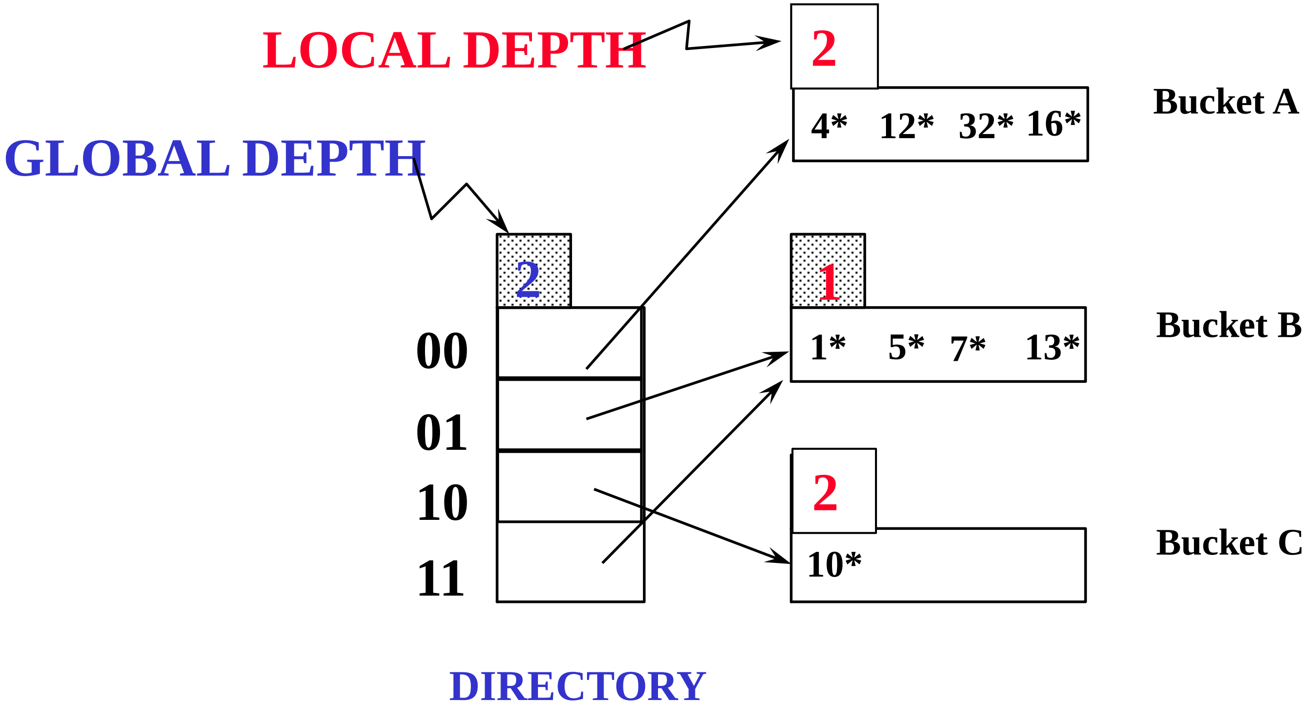
Search

- To search for a data entry, apply a hash function h to the key and take **the last d bits** of its binary representation to reach the directory entry, which points the required bucket
- Example: search for 5*



Insert

Let’s work on a few sample cases.



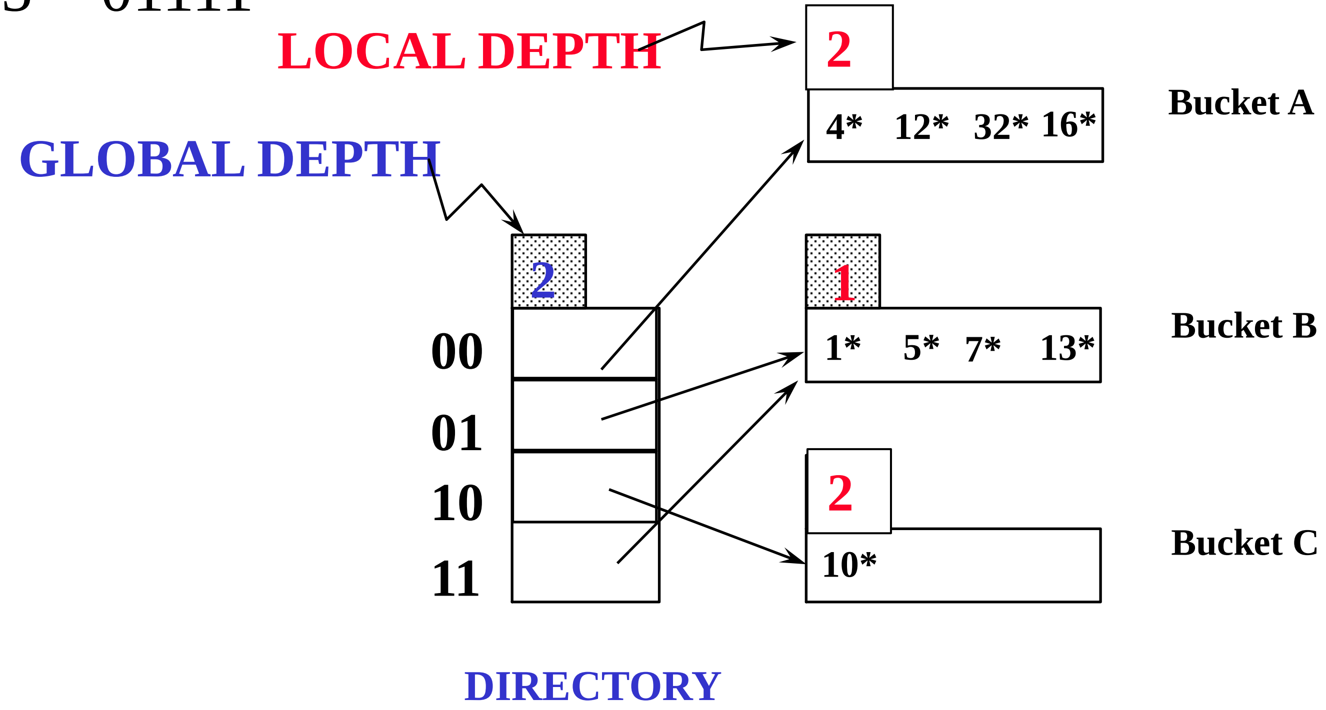
Handling Insertions

- Find bucket where data entry belongs.
- If there's room, put it there.
- Else, if bucket is full, *split* it:
  - increment local depth of original page
  - allocate new page with new local depth
  - re-distribute data entries from original page
  - add entry for the new page to the directory
  - double the directory *if necessary*

Example: Insert 21*,19*, 15*

Assume h(key) = key (in binary)

- 21 = 10101
- 19 = 10011
- 15 = 01111



Example

21 = 10101
- 19 = 10011
- 15 = 01111

if bucket is full, *split* it:

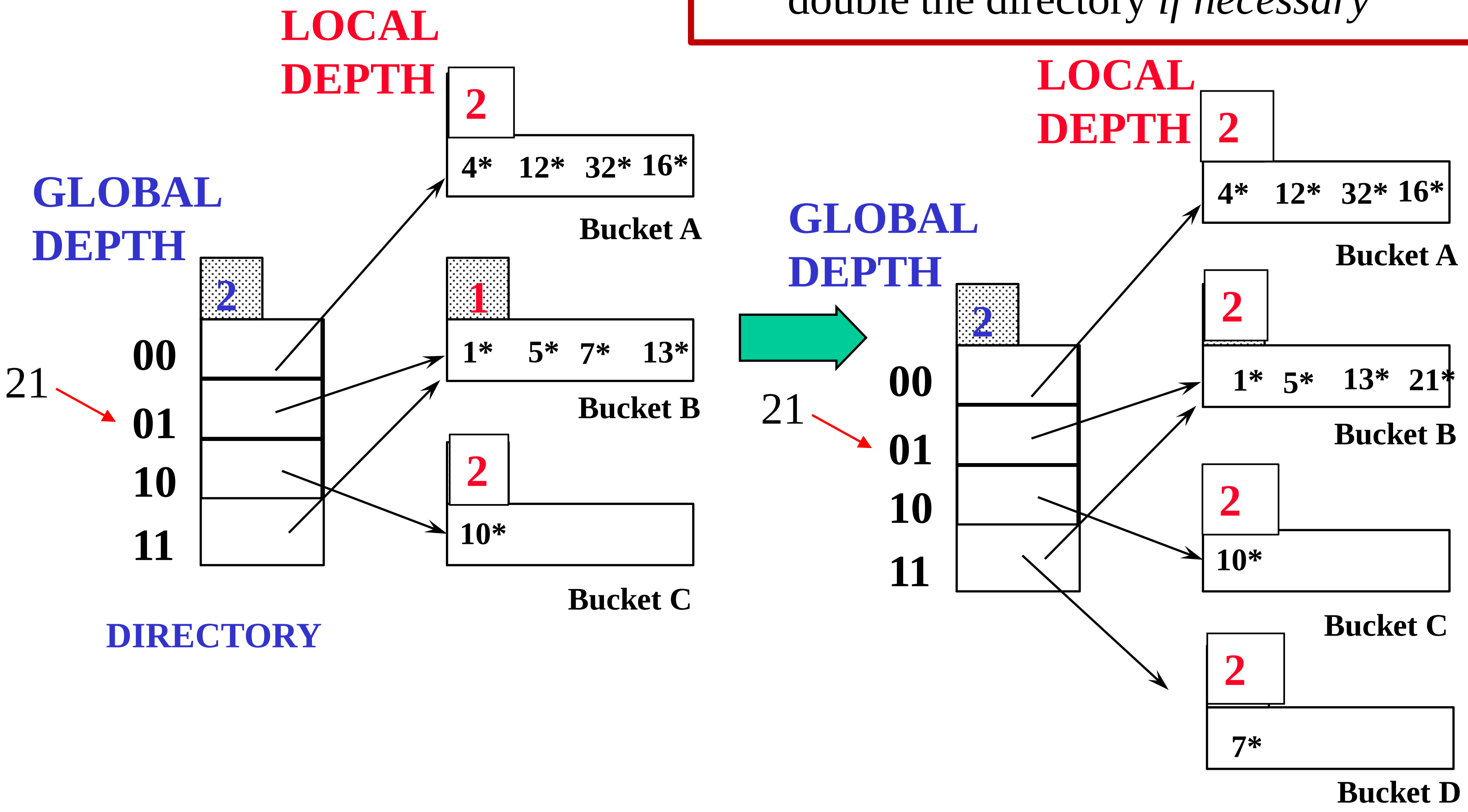
increment local depth

allocate new page with new local depth

re-distribute data entries of original page.

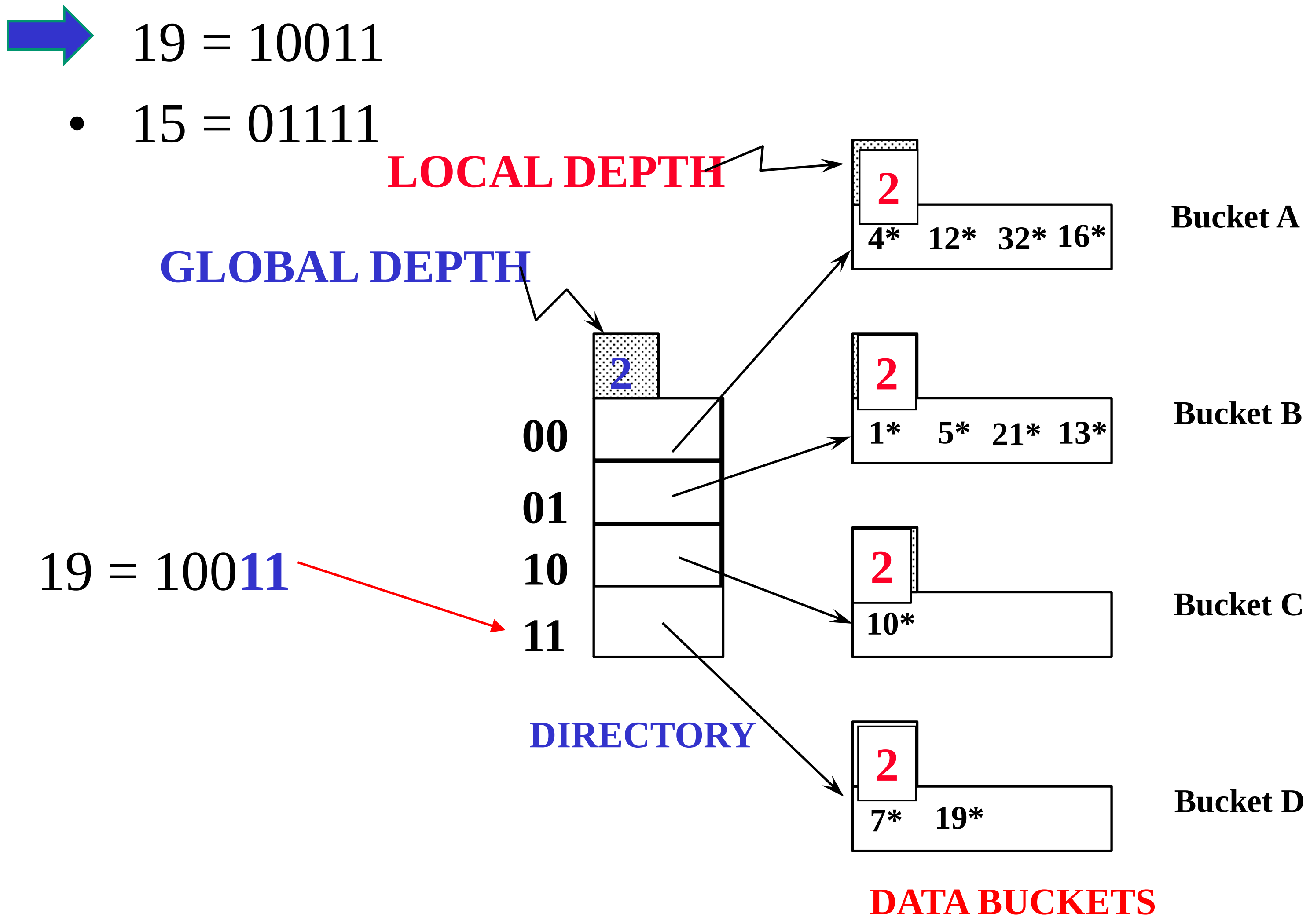
add entry for the new page to the directory

double the directory *if necessary*



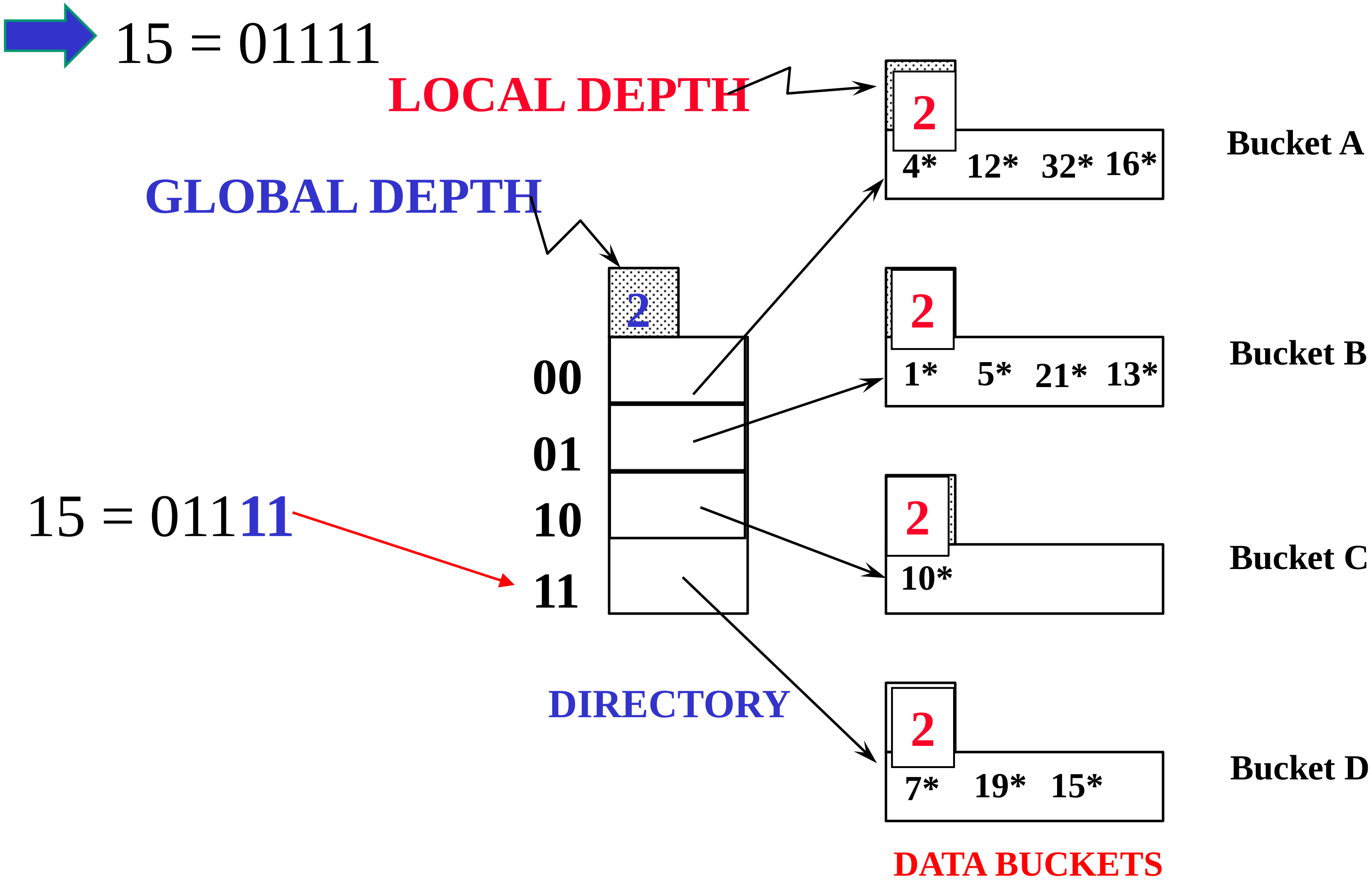
Example: Insert 21*,19*, 15*

- 21 = 10101
- 19 = 10011
- 15 = 01111



Example: Insert 21*,19*, 15*

- 21 = 10101
- 19 = 10011
- 15 = 01111



Example:
Insert now 20*

- insert 20*

if bucket is full, *split* it:

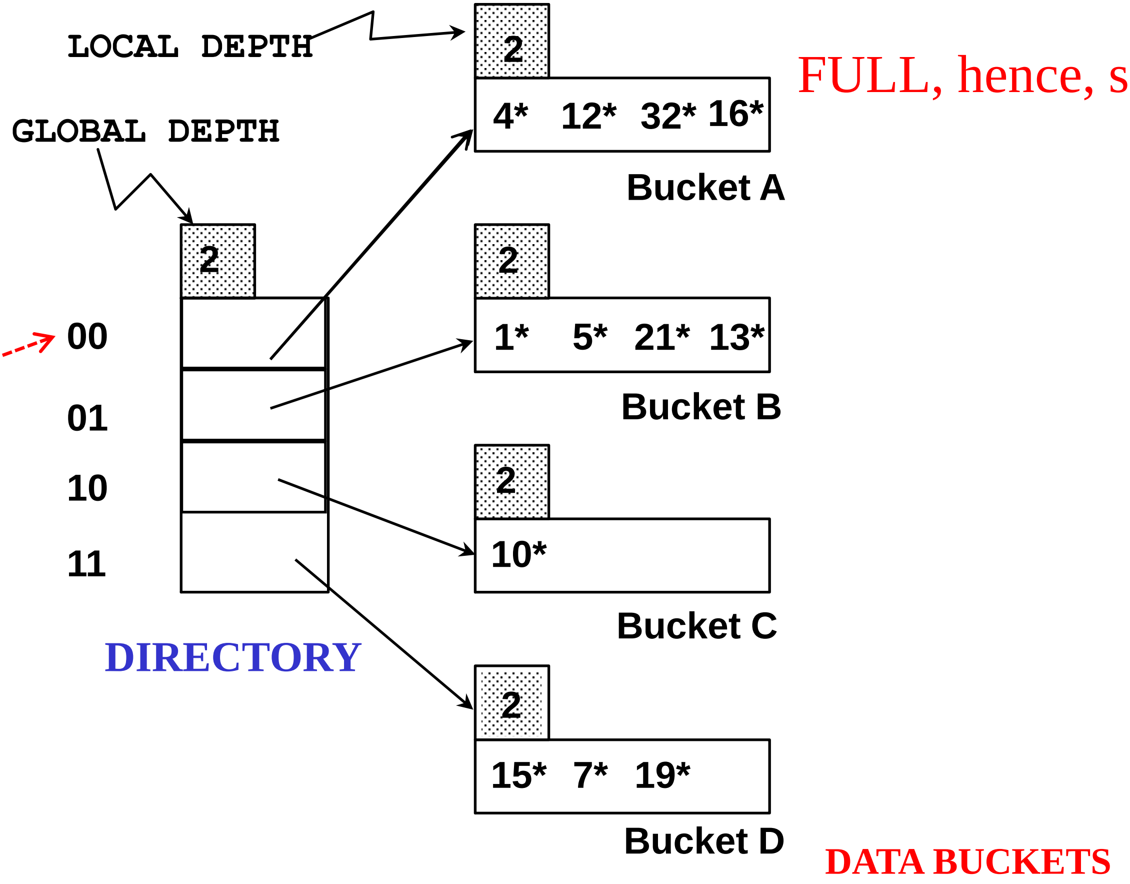
increment local depth

allocate new page with new local depth

re-distribute data entries of original page.

add entry for the new page to the directory

double the directory *if necessary*



FULL, hence, split!

20 = 10100

Because the local depth and the global depth are both 2, we *should* double the directory!

Example:
Insert now 20*

- insert 20*

if bucket is full, *split* it:

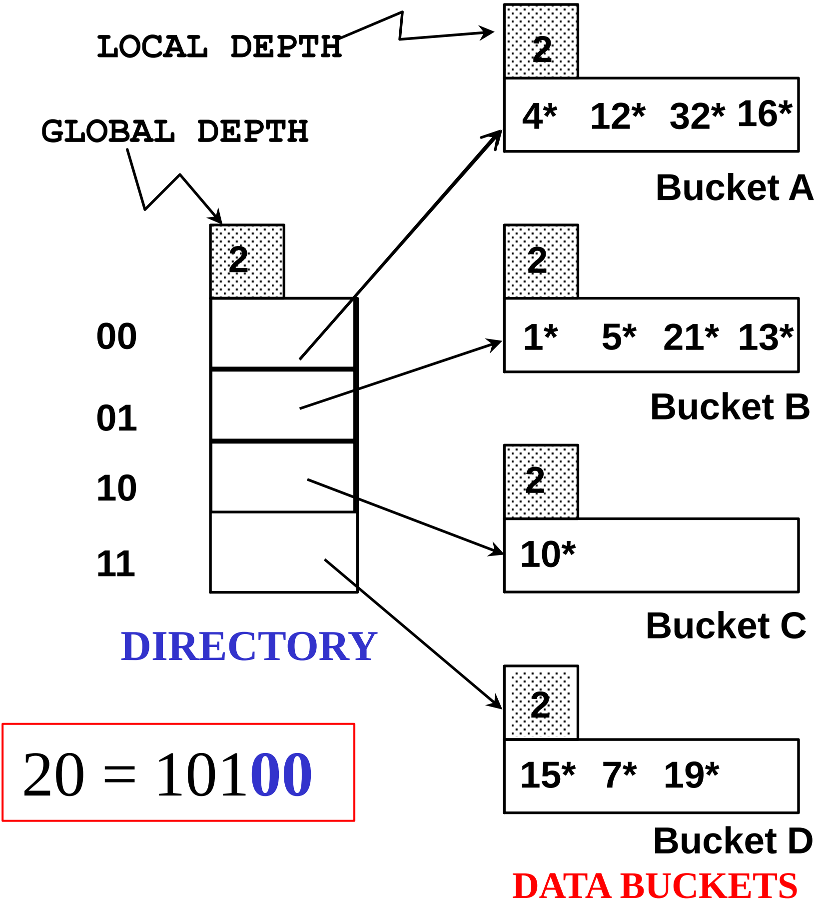
increment local depth

allocate new page with new local depth

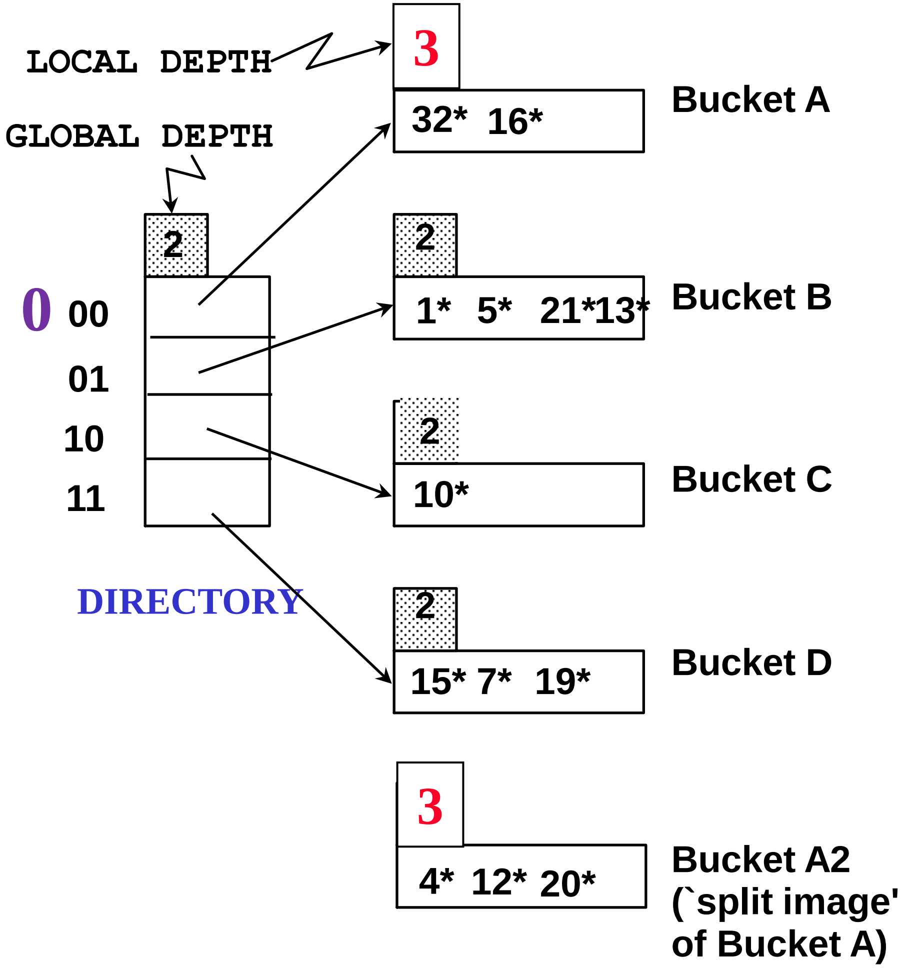
re-distribute data entries of original page.

add entry for the new page to the directory

double the directory *if necessary*

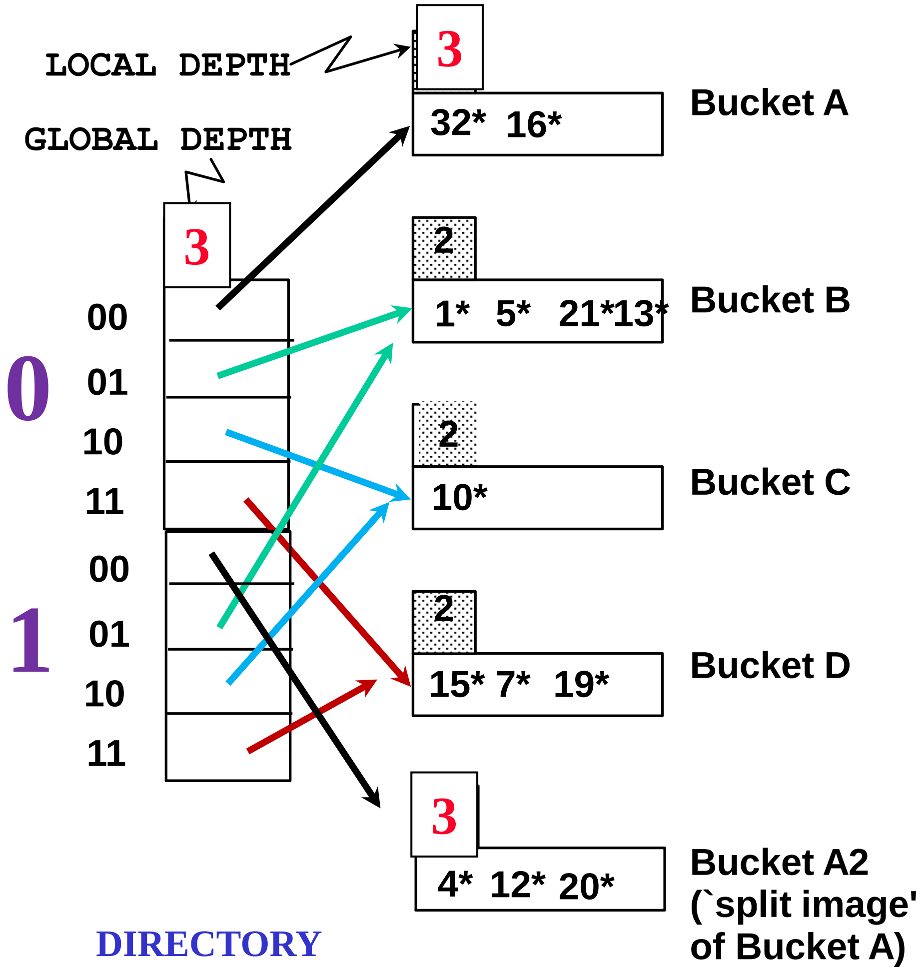


20 = 10100



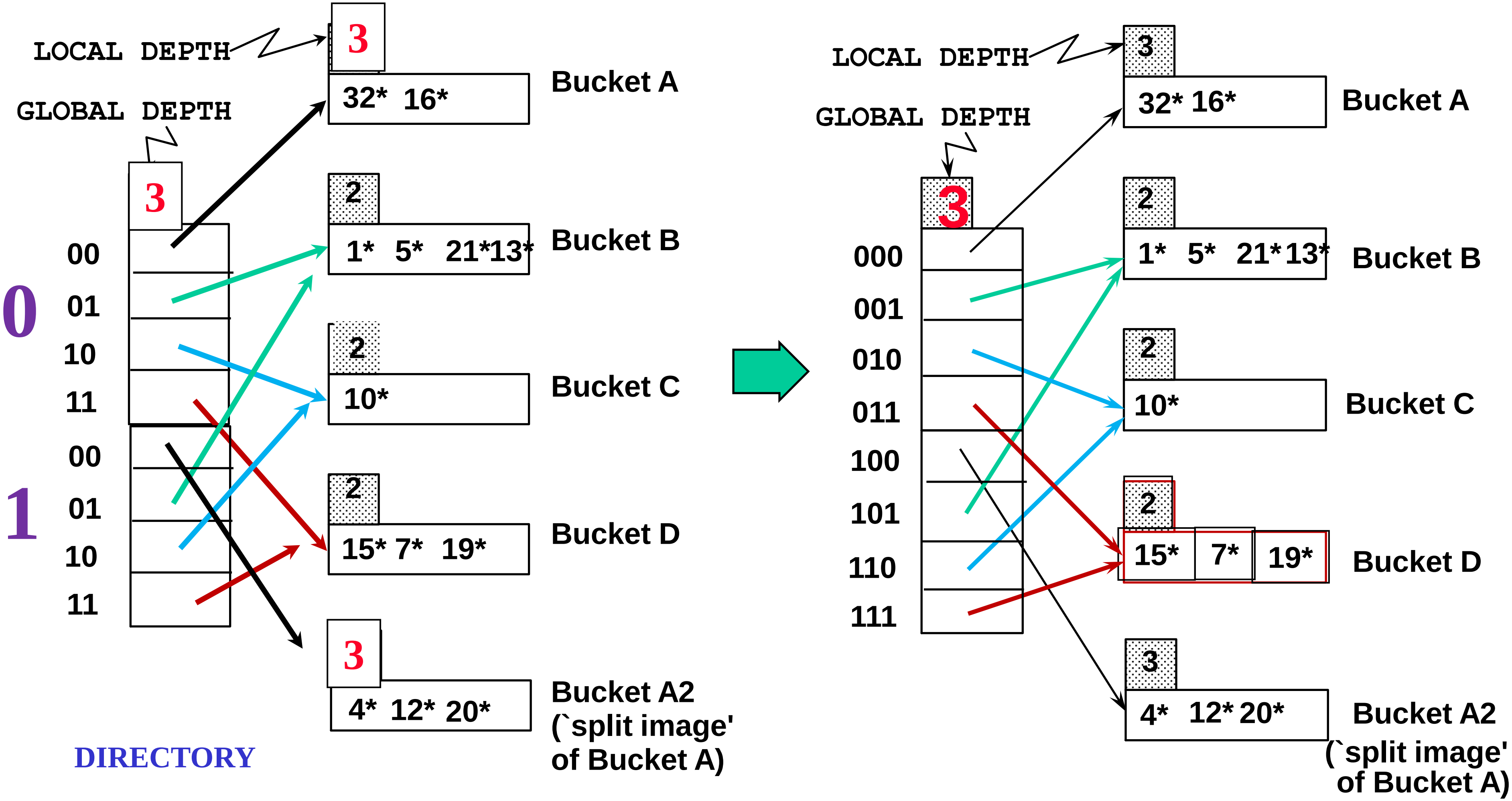
Example: Insert now 20*

Doubling the directory (also increment GlobalDepth)



Example: Insert now 20*

Doubling the directory (also increment GlobalDepth)

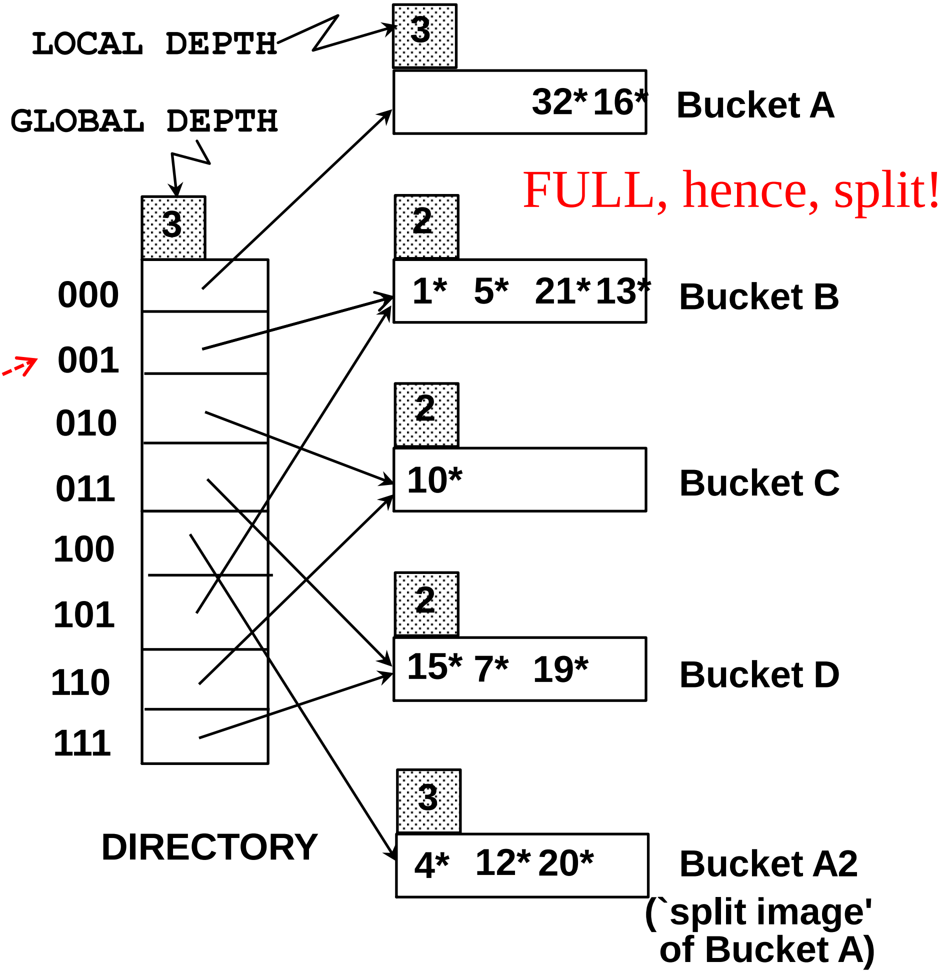


Example:
Insert now 9*

if bucket is full, *split* it:
    increment local depth
    allocate new page with new local depth
    re-distribute data entries of original page.
    add entry for the new page to the directory
    double the directory *if necessary* → *if LD = GD*

- insert 9*

9 = 1001

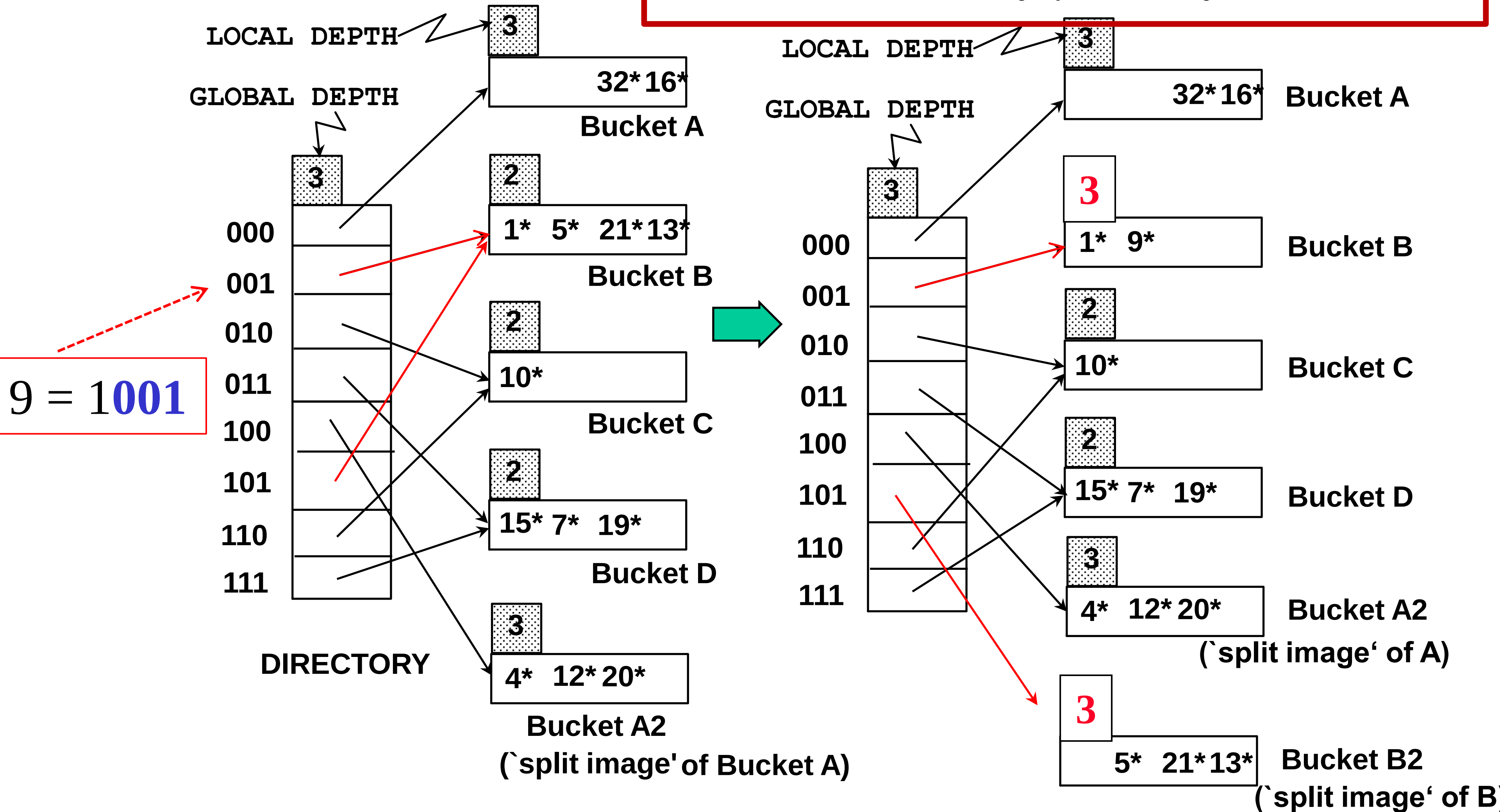


Because the local depth (i.e., 2) is *less than* the global depth (i.e., 3), NO need to double the directory

Example:
Insert now 9*

if bucket is full, *split* it:

- increment local depth
- allocate new page with new local depth
- re-distribute data entries of original page.
- add entry for the new page to the directory
- double the directory *if necessary*



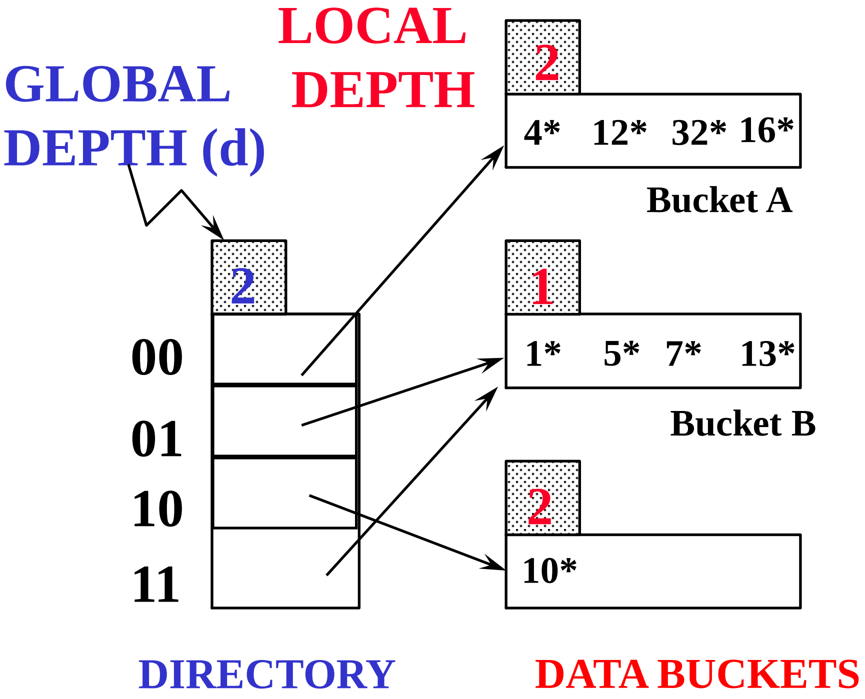
Points to Note

  - *Global depth of directory*: Max # of bits needed to tell which bucket an entry belongs to.
  - *Local depth of a bucket*: # of bits used to determine if an entry belongs to this bucket (esp. during a split)
- When does split cause directory doubling?
  - Before insert, *local depth* of bucket = *global depth*. Insert causes *local depth* to become > *global depth*; directory is doubled by *copying it over* and `fixing' pointer to split image page.

RECAP

Extendible Hashing

- **Basic Idea:**
  - No overflow buckets
  - A level of indirection: a **directory of pointers** to buckets
- Double the directory periodically
  - Directory much smaller than file, so doubling is cheaper.
- Split only the bucket that just overflowed!
  - Adjust the hash function



Comments on Extendible Hashing

- If directory fits in memory, **equality search** answered with **one disk access**.
  - A typical example: a100MB file with 100 bytes/entry and a page size of 4K contains 1,000,000 records (as data entries) but only about 25,000 directory elements

    $\Rightarrow$ chances are high that directory will fit in memory.

- If the distribution *of hash values* is skewed (e.g., a large number of search key values all are hashed to the same bucket), directory can grow large.
  - But this kind of skew can be avoided with a well-tuned hashing function

Skewed Insertions

Add the following entries in sequence in an intially empty extendible hash file (**Bucketing factor is 2**)

16 = 10000

32= 100000

4 = 100

(assume h(k) = k in binary)

Skewed Insertions

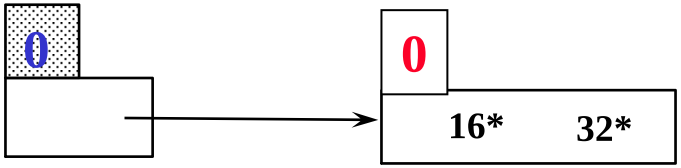
if bucket is full, *split* it:

- increment local depth
- allocate new page with new local depth
- re-distribute data entries of original page.
- add entry for the new page to the directory
- double the directory *if necessary*

Insert 16* and 32*

16 = 10000 and

32= 100000



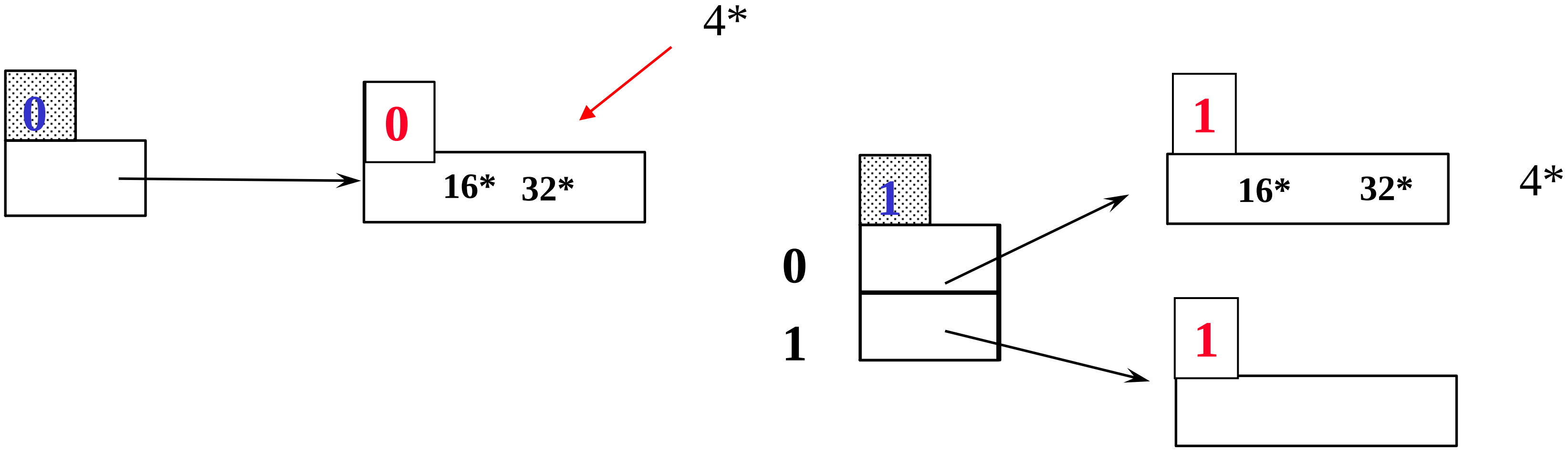
Skewed Insertions

if bucket is full, *split* it:

- increment local depth
- allocate new page with new local depth
- re-distribute data entries of original page.
- add entry for the new page to the directory
- double the directory *if necessary*

Insert 4*

4= 100



We need to double the directory

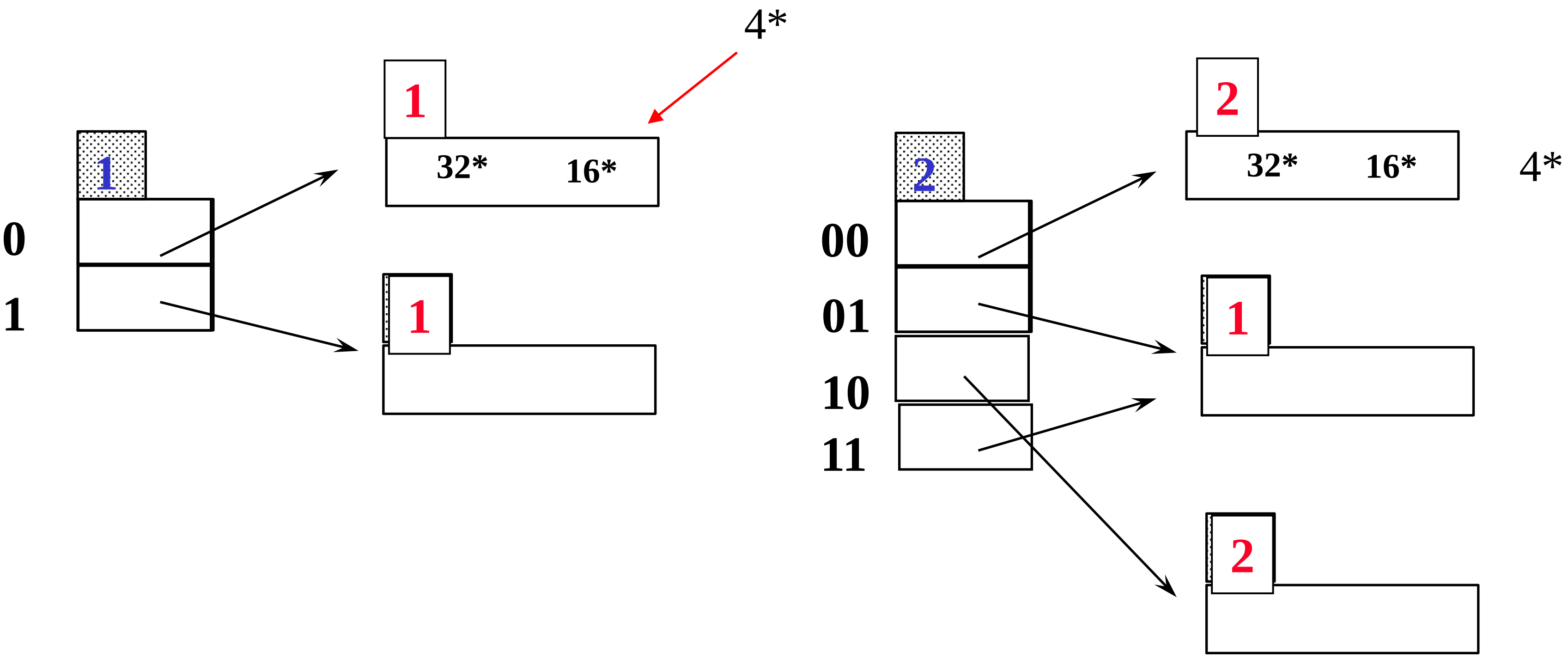
Skewed Insertions

if bucket is full, *split* it:

- increment local depth
- allocate new page with new local depth
- re-distribute data entries of original page.
- add entry for the new page to the directory
- double the directory *if necessary*

Insert 4*

4 = 100



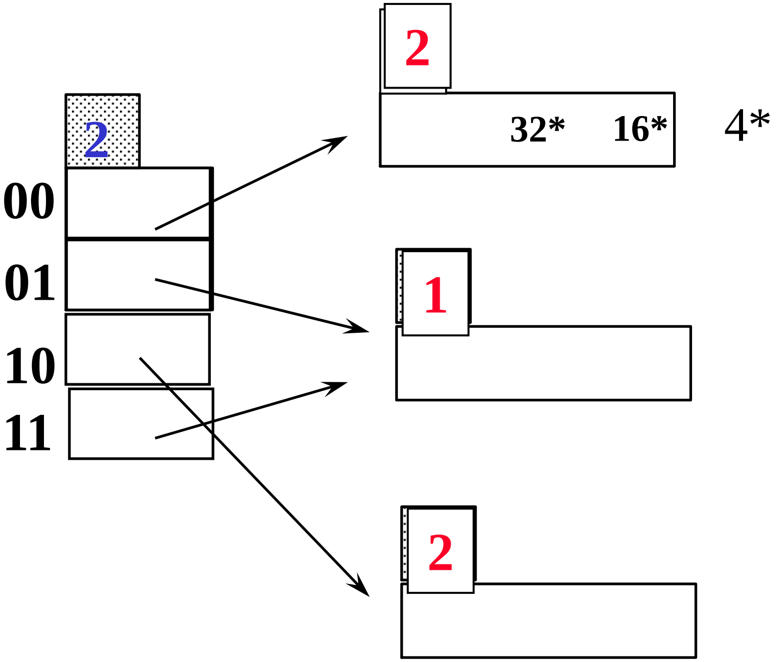
We need to double the directory again!

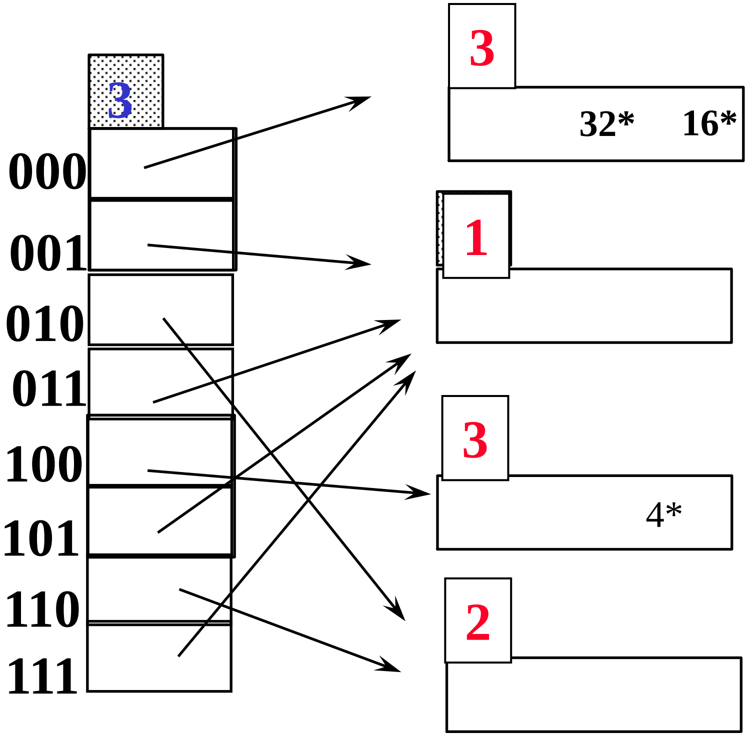
Skewed Insertions

if bucket is full, *split* it:

- increment local depth
- allocate new page with new local depth
- re-distribute data entries of original page.
- add entry for the new page to the directory
- double the directory *if necessary*

Still inserting 4*





We need to double the directory again !

How many direrctory elements pointing to a bucket?

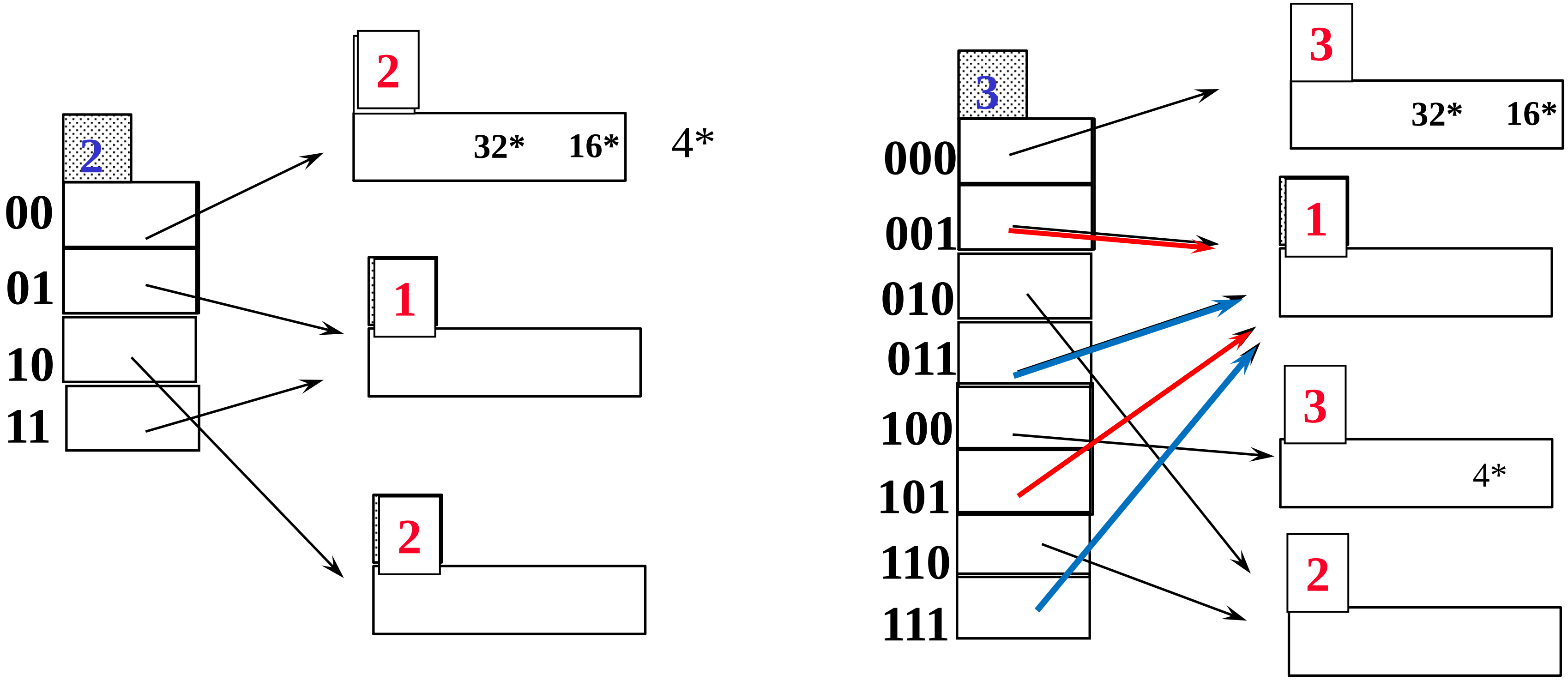
2^{GD-LD}

Skewed Insertions

if bucket is full, *split* it:

- increment local depth
- allocate new page with new local depth
- re-distribute data entries of original page.
- add entry for the new page to the directory
- double the directory *if necessary*

Still inserting 4*



We need to double the directory again !

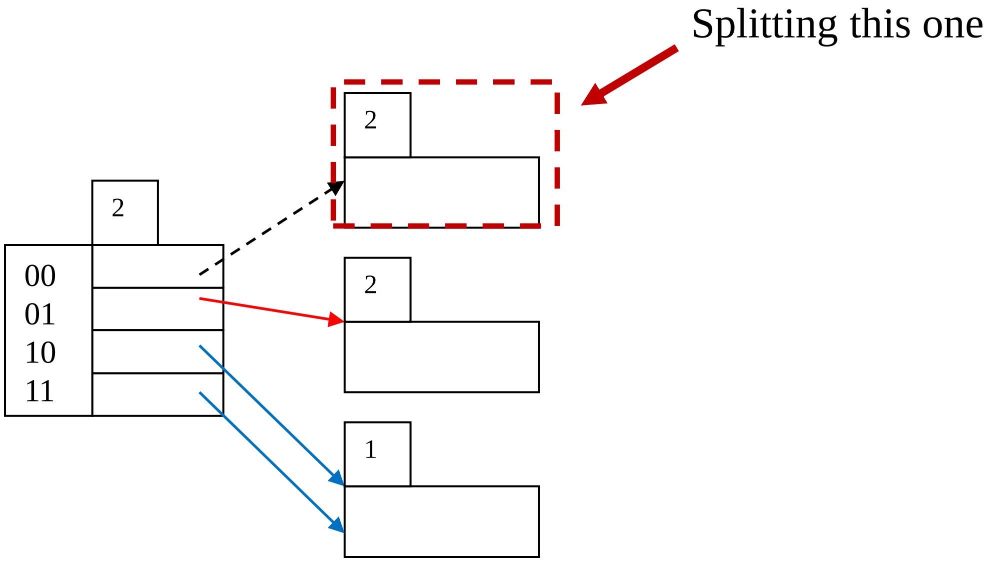
What if the bucket with LD = 1 is full?

Deletion

- Locate data entry in its bucket and remove it.
- If removal of data entry makes bucket empty, can be merged with `split image’
- If each directory element points to same bucket as its split image, can halve directory.
  - Note: decreasing directory size is an expensive operation and should be done only if number of buckets becomes much smaller than the size of the directory

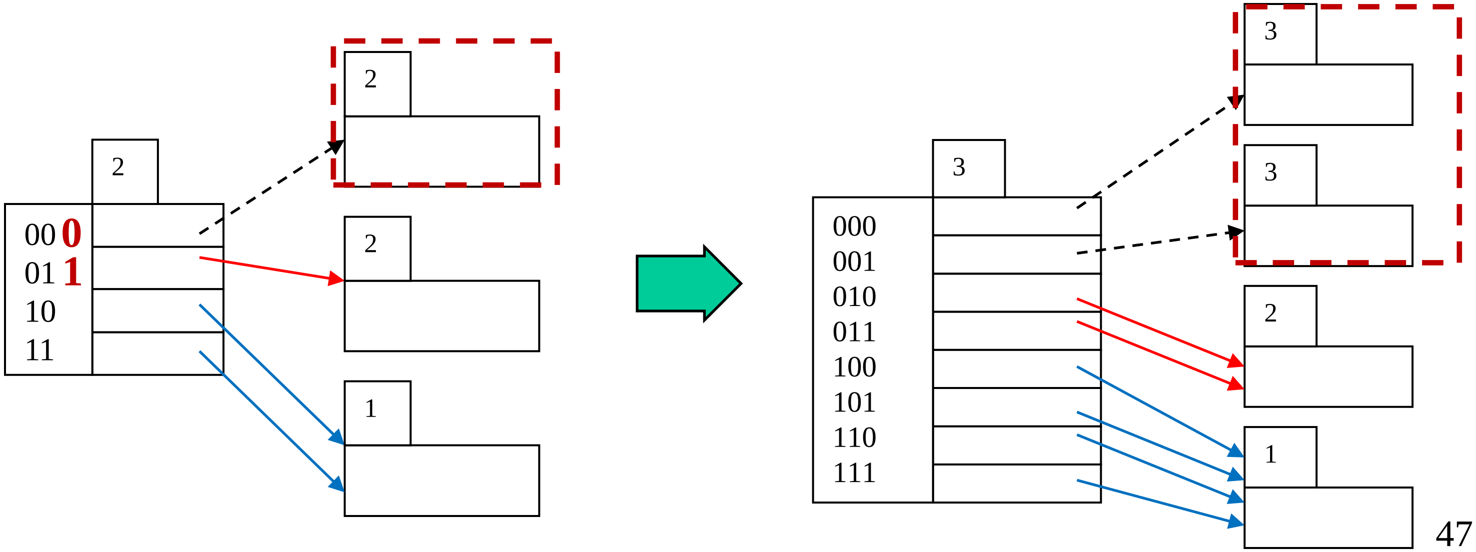
Can we use the most significant bits?

- Splitting (**Case 1**: i_j=i)
  - Only one element in directory (bucket address table) points to data bucket j
  - i++; split data bucket j to j, z; i_j=i_z=i; rehash all items previously in j;



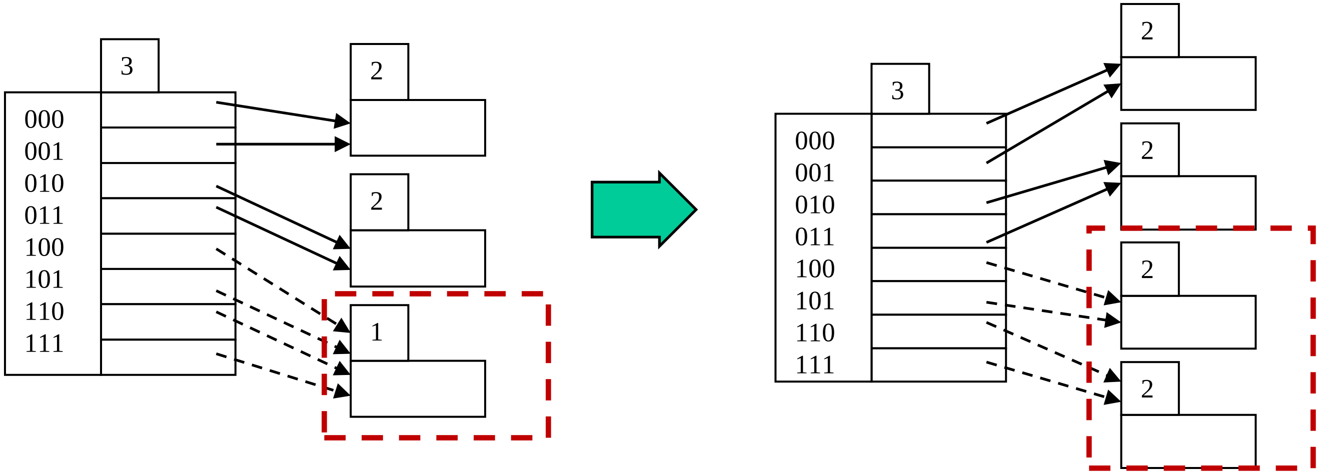
Can we use the most significant bits?

- Splitting (**Case 1**: i_j=i)
  - Only one element in directory (bucket address table) points to data bucket j
  - i++; split data bucket j to j, z; i_j=i_z=i; rehash all items previously in j;



Can we use the most significant bits ?

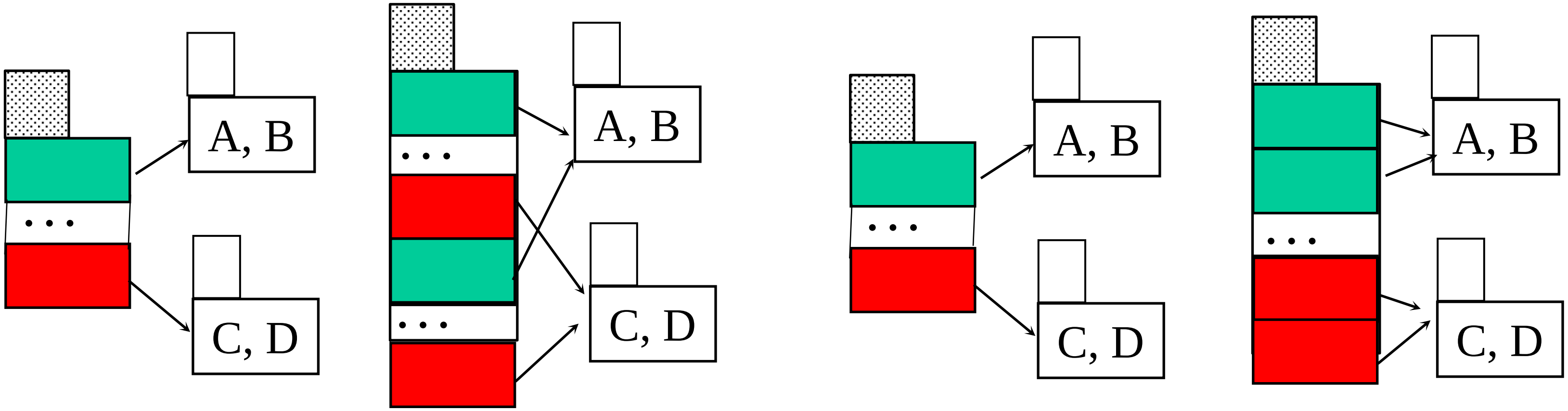
- Splitting (**Case 2:** $i_j < i$)
  - More than one element in directory (bucket address table) point to data bucket j
  - split data bucket j to j, z; $i_j = i_z = i_j + 1$; Adjust the pointers previously point to j to j and z; rehash all items previously in j;



Directory Doubling

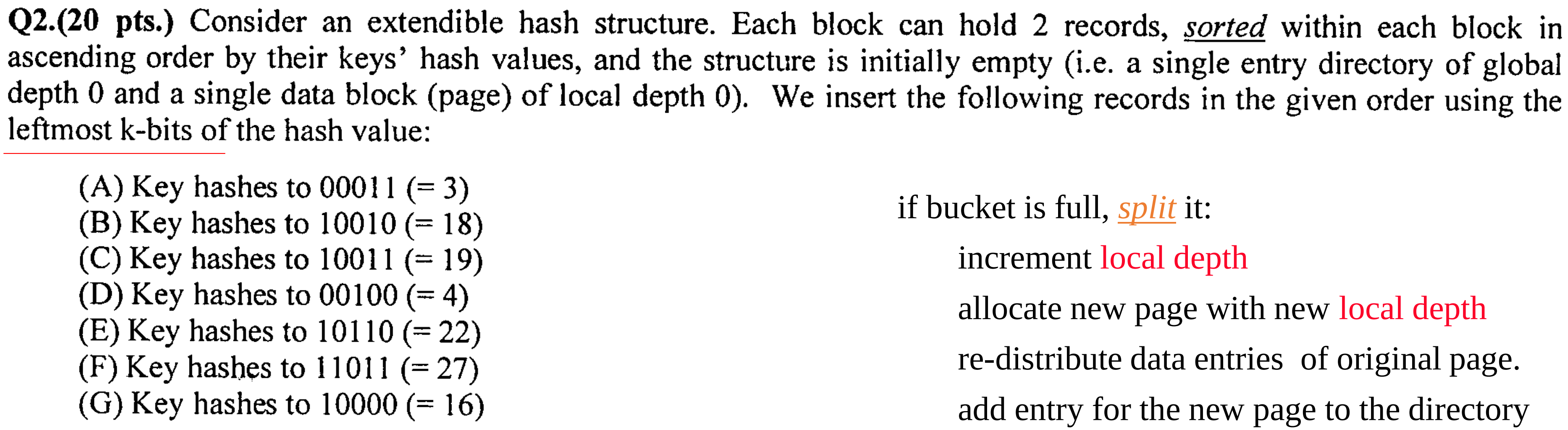
Why prefer least significant bits in directory (instead of the *most* significant ones)?

Allows for doubling by copying the directory and appending the new copy to the original.



Least Significant **vs.** **Most Significant**

Skewed Insertions using most significant bits



Q2.(20 pts.) Consider an extendible hash structure. Each block can hold 2 records, *sorted* within each block in ascending order by their keys' hash values, and the structure is initially empty (i.e. a single entry directory of global depth 0 and a single data block (page) of local depth 0). We insert the following records in the given order using the leftmost k-bits of the hash value:

(A) Key hashes to 00011 (= 3)
(B) Key hashes to 10010 (= 18)
(C) Key hashes to 10011 (= 19)
(D) Key hashes to 00100 (= 4)
(E) Key hashes to 10110 (= 22)
(F) Key hashes to 11011 (= 27)
(G) Key hashes to 10000 (= 16)

if bucket is full, *split* it:
- increment local depth
- allocate new page with new local depth
- re-distribute data entries of original page.
- add entry for the new page to the directory
- double the directory *if necessary*

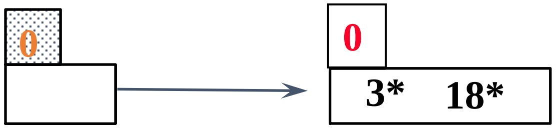
Skewed Insertions using most significant bits

Q2.(20 pts.) Consider an extendible hash structure. Each block can hold 2 records, *sorted* within each block in ascending order by their keys' hash values, and the structure is initially empty (i.e. a single entry directory of global depth 0 and a single data block (page) of local depth 0). We insert the following records in the given order using the leftmost k-bits of the hash value:

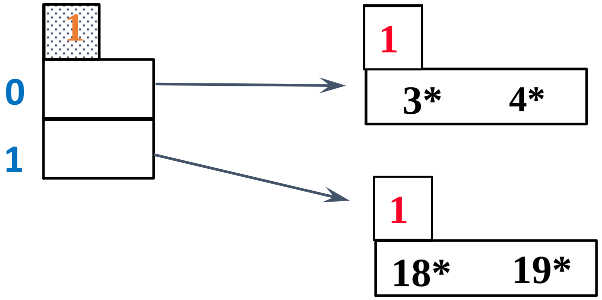
(A) Key hashes to 00011 (= 3)
(B) Key hashes to 10010 (= 18)
(C) Key hashes to 10011 (= 19)
(D) Key hashes to 00100 (= 4)
(E) Key hashes to 10110 (= 22)
(F) Key hashes to 11011 (= 27)
(G) Key hashes to 10000 (= 16)

if bucket is full, *split* it:

- increment local depth
- allocate new page with new local depth
- re-distribute data entries of original page.
- add entry for the new page to the directory
- double the directory *if necessary*



After 3 and 18 are inserted

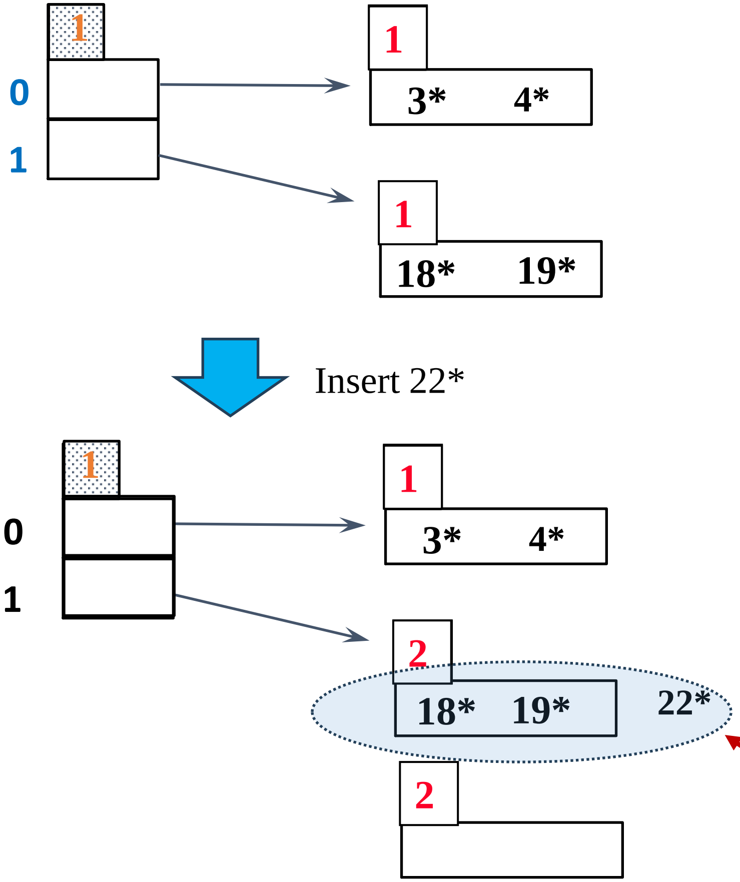


After 19 and 4 are also inserted

(A) Key hashes to 00011 (= 3)
(B) Key hashes to 10010 (= 18)
(C) Key hashes to 10011 (= 19)
(D) Key hashes to 00100 (= 4)
(E) Key hashes to 10110 (= 22)
(F) Key hashes to 11011 (= 27)
(G) Key hashes to 10000 (= 16)

if bucket is full, *split* it:

- increment local depth
- allocate new page with new local depth
- re-distribute data entries of original page.
- add entry for the new page to the directory
- double the directory *if necessary*



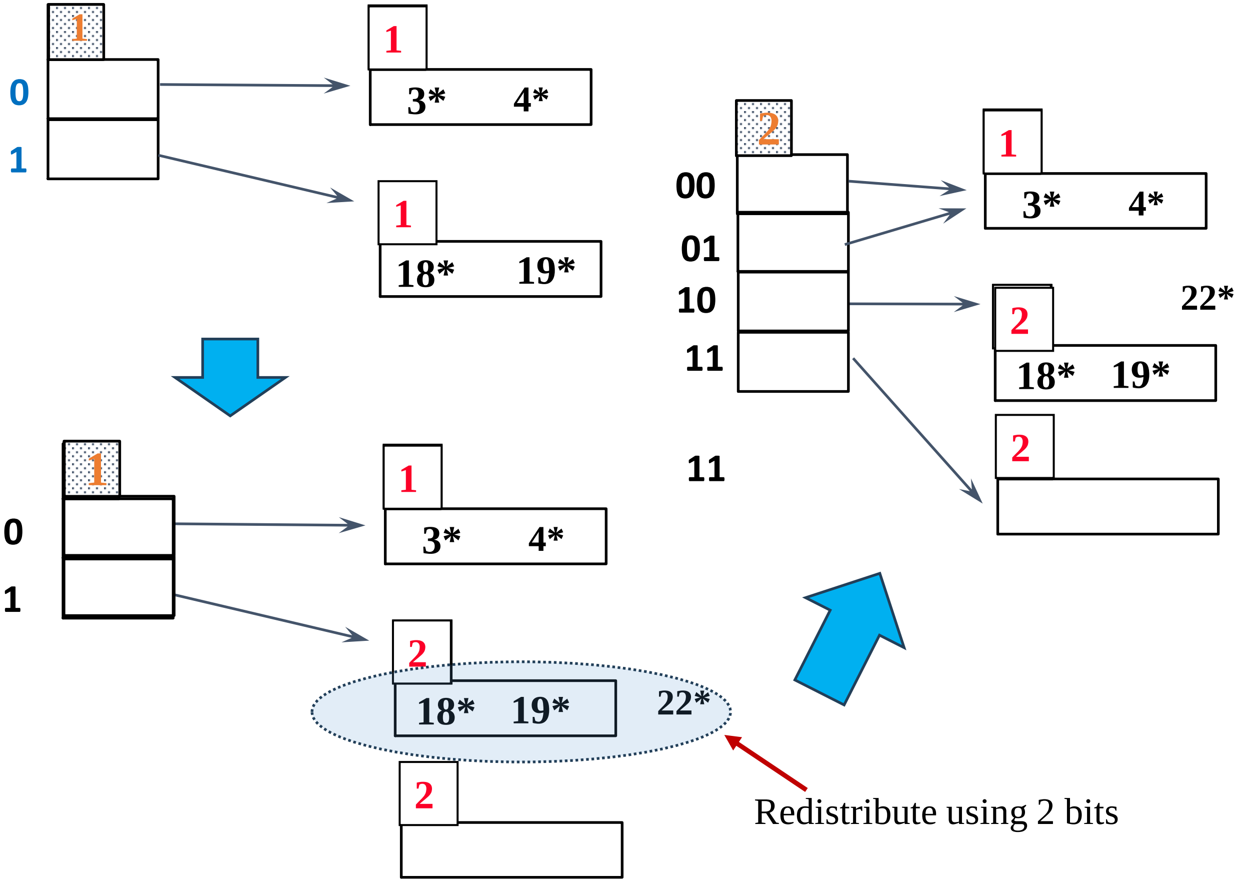
Insert 22*

Redistribute using 2 bits

if bucket is full, *split* it:

- increment local depth
- allocate new page with new local depth
- re-distribute data entries of original page.
- add entry for the new page to the directory
- double the directory *if necessary*

(A) Key hashes to 00011 (= 3)
(B) Key hashes to 10010 (= 18)
(C) Key hashes to 10011 (= 19)
(D) Key hashes to 00100 (= 4)
(E) Key hashes to 10110 (= 22)
(F) Key hashes to 11011 (= 27)
(G) Key hashes to 10000 (= 16)

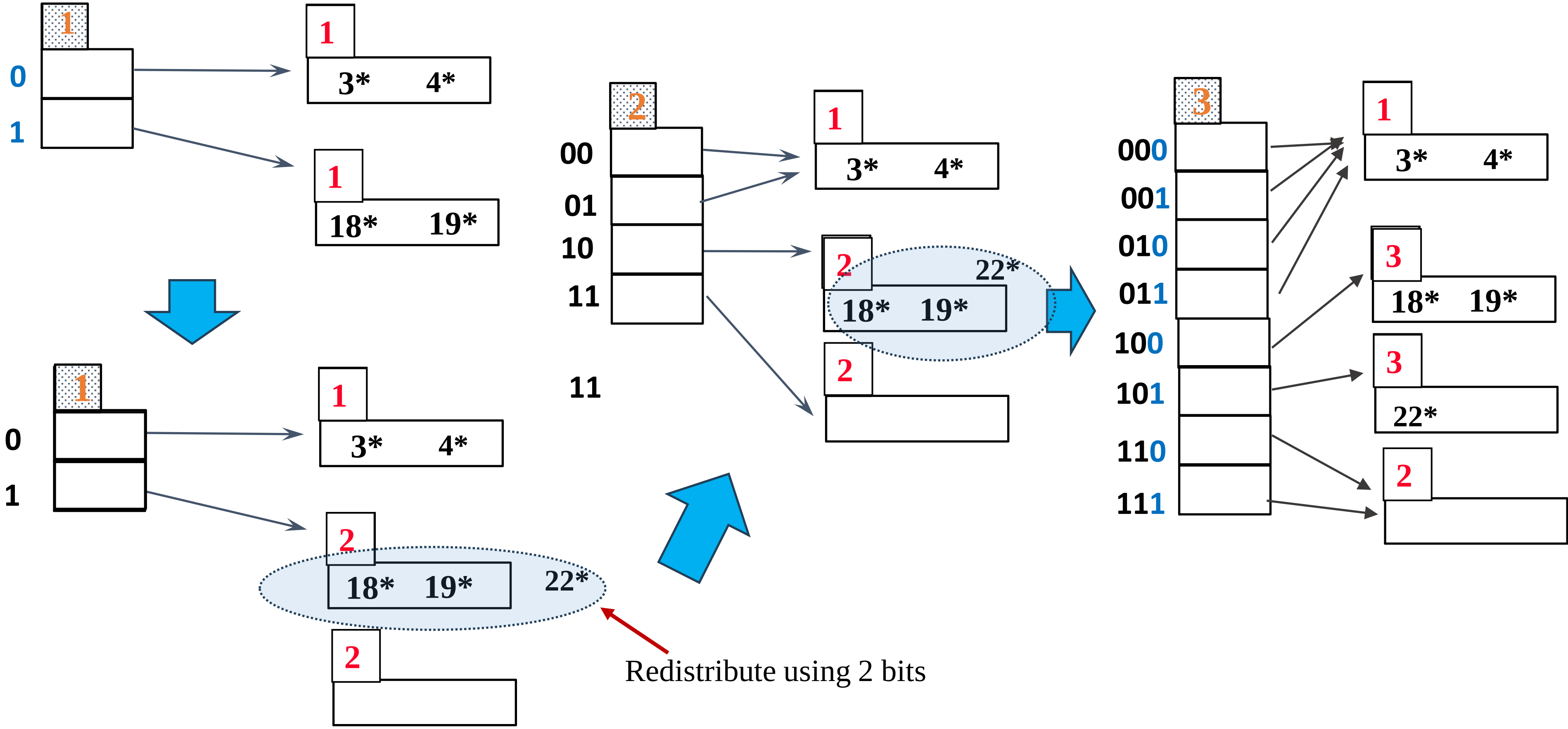


Redistribute using 2 bits

(A) Key hashes to 00011 (= 3)
(B) Key hashes to 10010 (= 18)
(C) Key hashes to 10011 (= 19)
(D) Key hashes to 00100 (= 4)
(E) Key hashes to 10110 (= 22)
(F) Key hashes to 11011 (= 27)
(G) Key hashes to 10000 (= 16)

if bucket is full, *split* it:

- increment local depth
- allocate new page with new local depth
- re-distribute data entries of original page.
- add entry for the new page to the directory
- double the directory *if necessary*

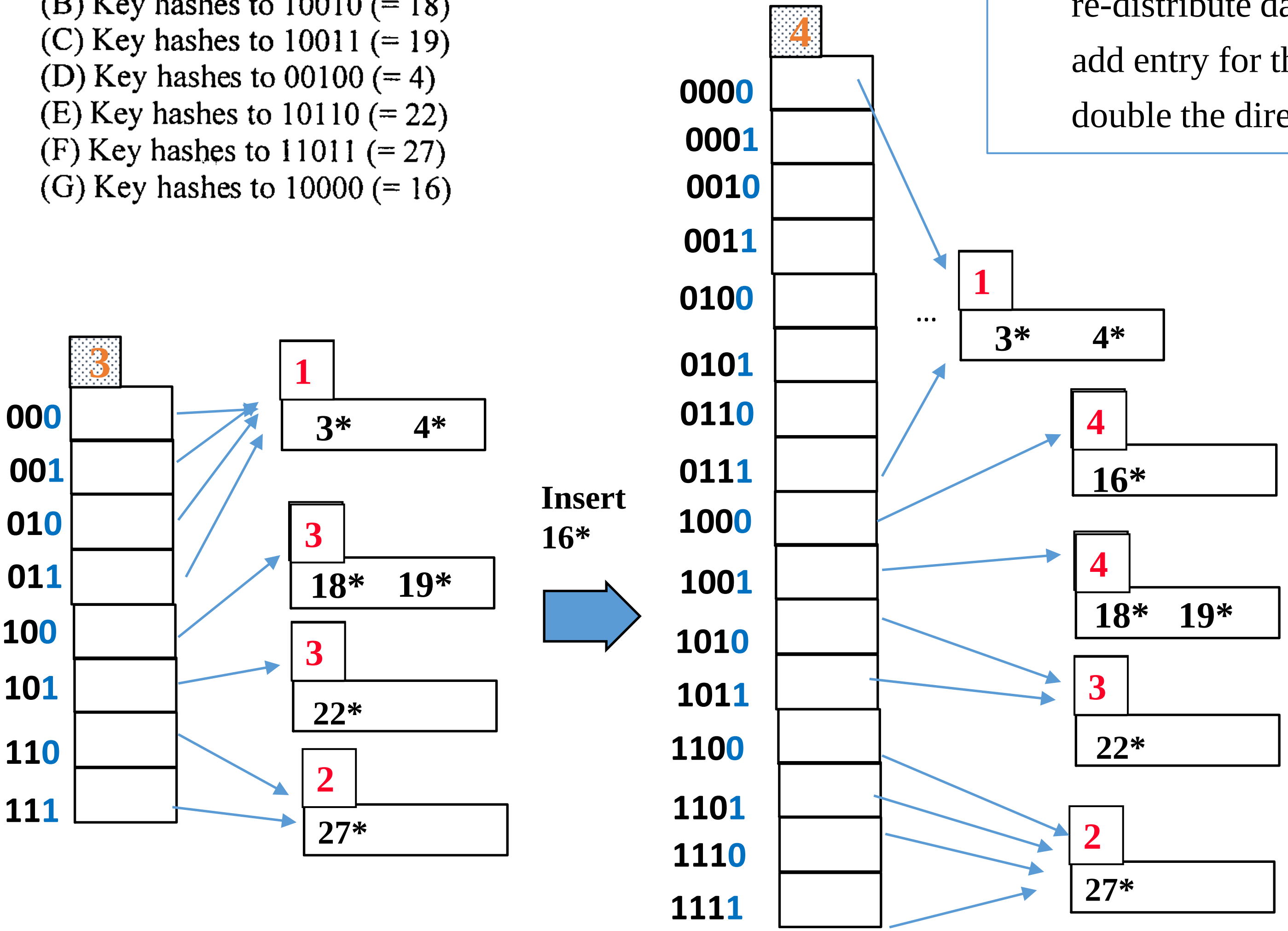


Redistribute using 2 bits

(A) Key hashes to 00011 (= 3)
(B) Key hashes to 10010 (= 18)
(C) Key hashes to 10011 (= 19)
(D) Key hashes to 00100 (= 4)
(E) Key hashes to 10110 (= 22)
(F) Key hashes to 11011 (= 27)
(G) Key hashes to 10000 (= 16)

if bucket is full, *split* it:

- increment local depth
- allocate new page with new local depth
- re-distribute data entries of original page.
- add entry for the new page to the directory
- double the directory *if necessary*



How many direrctory elements pointing to a bucket?

2^{GD-LD}

Summary

- Extendible Hashing avoids overflow pages by splitting a full bucket when a new data entry is to be added to it.
  - Directory to keep track of buckets, doubles periodically.
  - Can get large with skewed data; additional I/O if this does not fit in main memory.

Linear Hashing

Linear Hashing

- It maintains a **constant load factor**.
  - Thus, avoids reorganization.
- It does so, by **incrementally adding** new buckets to the primary area.
- In linear hashing, the last bits in the hash number are used for placing the data entries.

Example

e.g.
34: 100010
28: 011100
08: 001000

Desired Lf = 67%= 2/3

Lf = 14/24

= 58%

Last 3 bits ↓

000	8*	16*	32*
001	17*	25*	
010	34*	50*	
011	11*	27*	
100	28*		
101	5*		
110	14*		
111	55*	15*	

Primary area

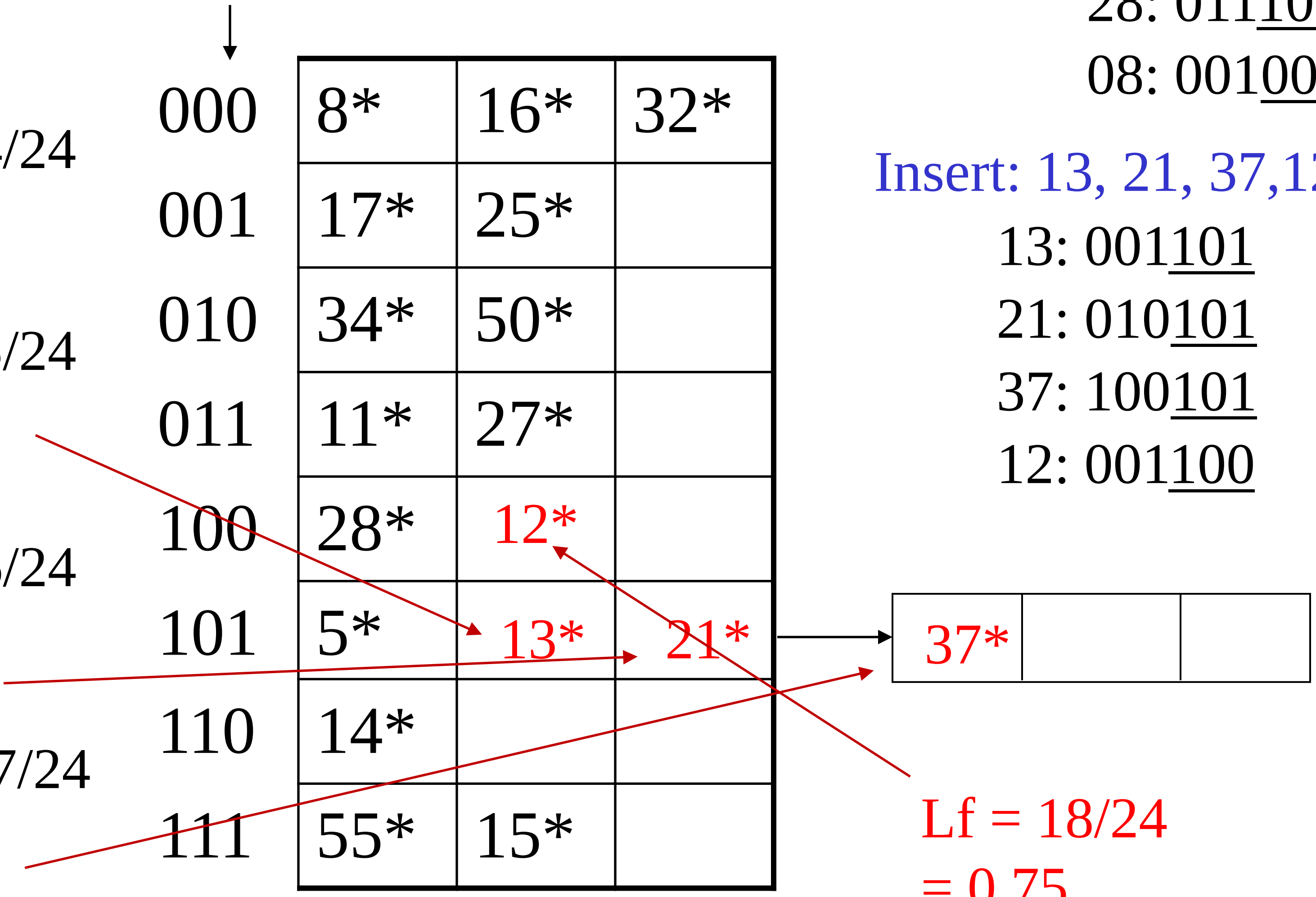
Insert: 13, 21, 37,12
13: 001101
21: 010101
37: 100101
12: 001100

Example

e.g.
34: 100010
28: 011100
08: 001000

Last 3 bits

Desired Lf = 67%= 2/3



000	8*	16*	32*
001	17*	25*	
010	34*	50*	
011	11*	27*	
100	28*	12*	
101	5*	13*	21*
110	14*		
111	55*	15*	

37*		

Primary area

Lf = 14/24
= 58%

Lf = 15/24
= 63%

Lf = 16/24
= 67%

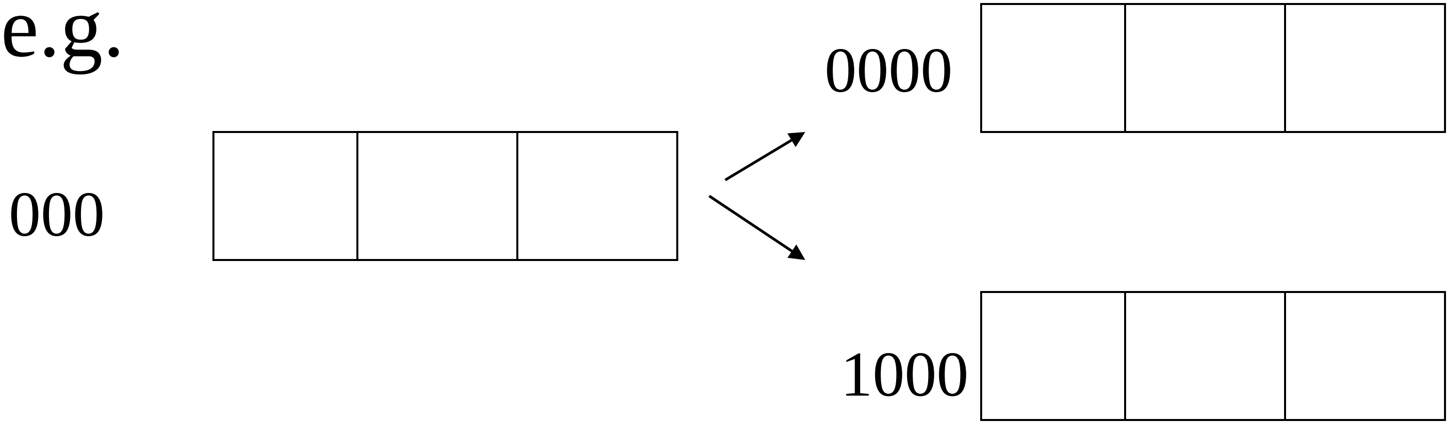
Lf = 17/24
= 70%

Lf = 18/24
= 0.75

Insert: 13, 21, 37,12
13: 001101
21: 010101
37: 100101
12: 001100

Insertion of data entries

- To **expand** the file: **split** an existing bucket denoted by **k digits** into two buckets using the last **k+1 digits**.
- e.g.



When to Split?

- When there are *Lf*Bkfr* data entries **more than** needed for **the given Lf.**

Split when there are *Lf*Bkfr* data entries more than needed for the desired Lf

Last 3 bits

Desired Lf = 67%=2/3

Lf = 14/(3*8)

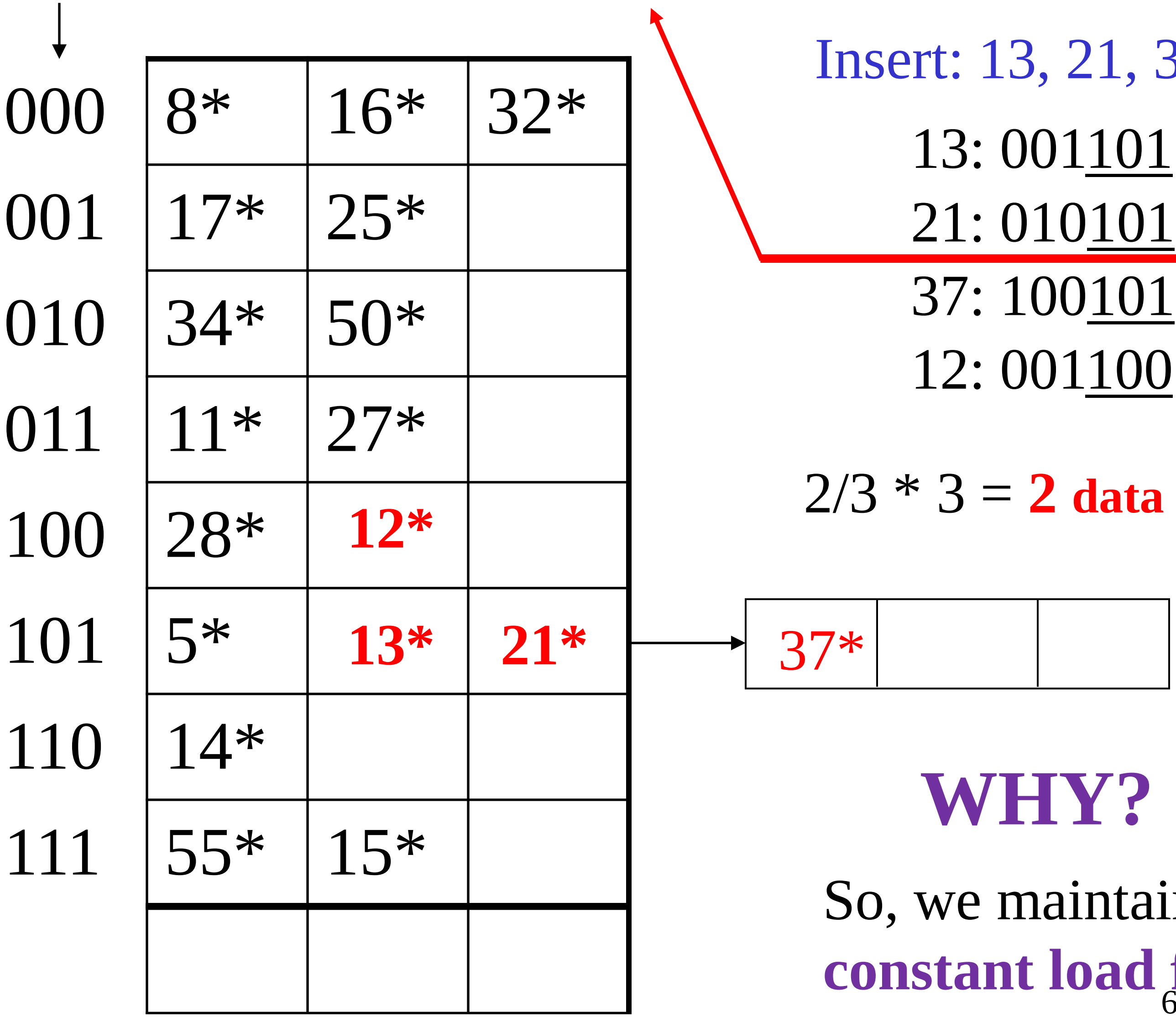
= 58%

Lf = 15/24

= 63%

Lf = 16/24

= 67%



000	8*	16*	32*
001	17*	25*	
010	34*	50*	
011	11*	27*	
100	28*	**12***	
101	5*	**13***	**21***
110	14*		
111	55*	15*	

37*		

Insert: 13, 21, 37,12

13: 001101

21: 010101

37: 100101

12: 001100

2/3 * 3 = **2 data entries**

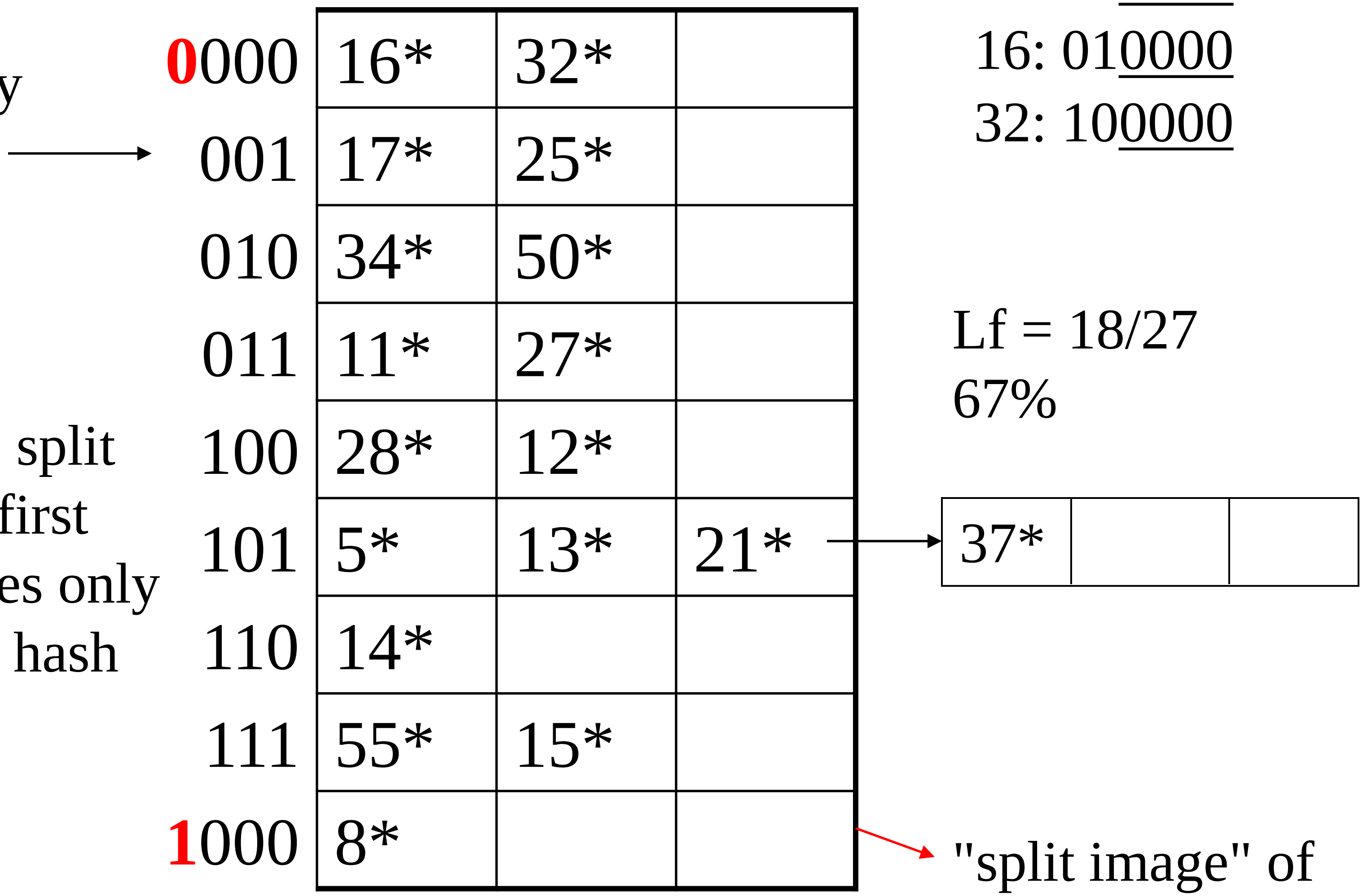
WHY?

So, we maintain a **constant load factor.**

Expanding the file

Boundary value

- Points to the next bucket to split
- Poins to the first bucket that uses only k-digits of the hash value

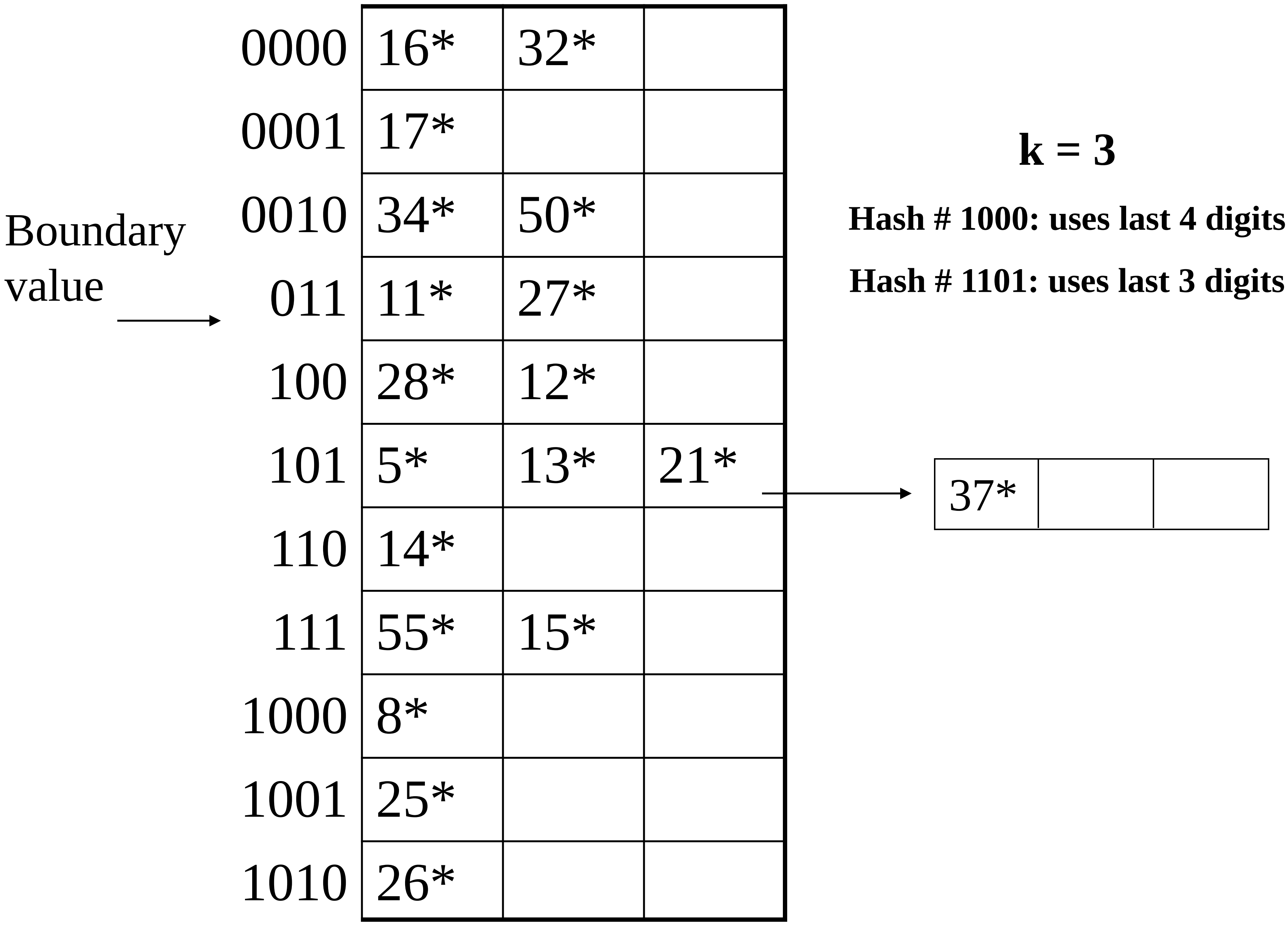


0000	16*	32*	
001	17*	25*	
010	34*	50*	
011	11*	27*	
100	28*	12*	
101	5*	13*	21*
110	14*		
111	55*	15*	
1000	8*		

8: 001000
16: 010000
32: 100000

Lf = 18/27
67%

"split image" of the bucket 000



Boundary value

0000	16*	32*	
0001	17*		
0010	34*	50*	
011	11*	27*	
100	28*	12*	
101	5*	13*	21*
110	14*		
111	55*	15*	
1000	8*		
1001	25*		
1010	26*		

37*		

k = 3

Hash # 1000: uses last 4 digits

Hash # 1101: uses last 3 digits

Fetching a data entry

- Calculate the hash function.
- Look at the last k digits.
  - If it’s less than the boundary value, the location is in the bucket labeled with the last k+1 digits.
  - Otherwise it is in the bucket labeled with the last k digits.
- Follow overflow chains as with static hashing.

$$\text{key} < \text{bv} \rightarrow \text{k+1 digits}$$

$$\underbrace{\text{key}}_{\text{last k digits}} \geq \text{bv} \rightarrow \text{k digits}$$

Insertion

- Search for the correct bucket into which to place the new record.
- If the bucket is full, allocate a new overflow bucket.
- If there are now *Lf*Bkfr* data entries more than needed for the given Lf,
  - Add one more bucket to the primary area.
  - Distribute the data entries from the **bucket chain** at the boundary value between the original area and the new primary area buckets
  - Add 1 to the boundary value.

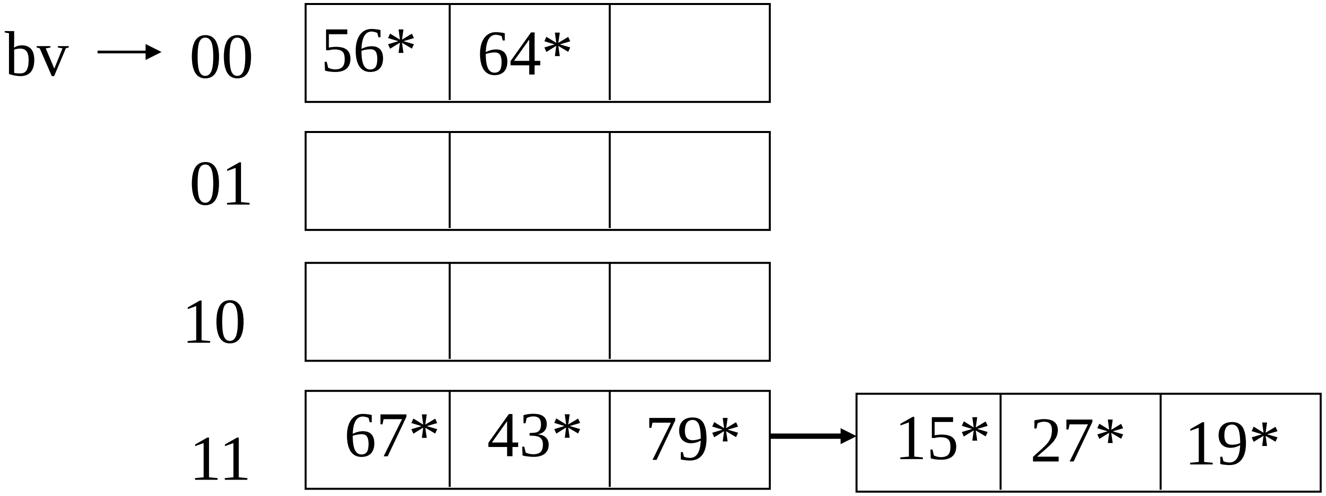
Bkfr = 3, desired LF= 2/3,
M=4 primary area buckets initially
What is k?

What is the initial structure?
When to reach the desired load value when the given elements are inserted?
When to split the first node?

56 = 011 1000
67 = 100 0011
43 = 010 1011
79 = 100 1111
15 = 000 1111
27 = 001 1011
19 = 001 0011
64 = 100 0000

12 = 000 1100
33 = 010 0001
57 = 011 1001
65 = 100 0001
29 = 001 1101

Bkfr = 3, desired LF= 2/3,
M=4 primary area buckets initially
What is k?



56 = 011 1000
67 = 100 0011
43 = 010 1011
79 = 100 1111
15 = 000 1111
27 = 001 1011
19 = 001 0011
64 = 100 0000

12 = 000 1100
33 = 010 0001
57 = 011 1001
65 = 100 0001
29 = 001 1101

LF = 2 / 3 = n / (3*4)

Desired LF is reached at n = 8

key < bv → k+1 digits
key ≥ bv → k digits
last k digits

After Lf * Bkfr = 2/3 * 3 = 2
more data entries, **split!**

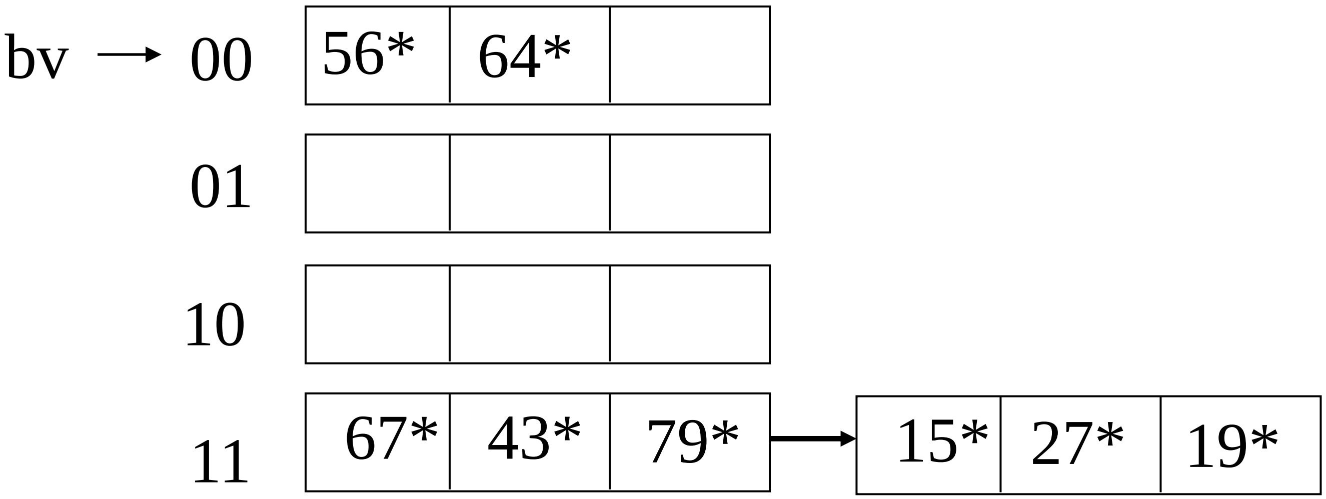
Bkfr = 3, desired LF= 2/3,
4 primary area buckets initially
k=2

56 = 011 1000
64 = 100 0000

12 = 000 1100
33 = 010 0001

57 = 011 1001
65 = 100 0001
29 = 001 1101

insert



key < bv → k+1 digits
key ≥ bv → k digits

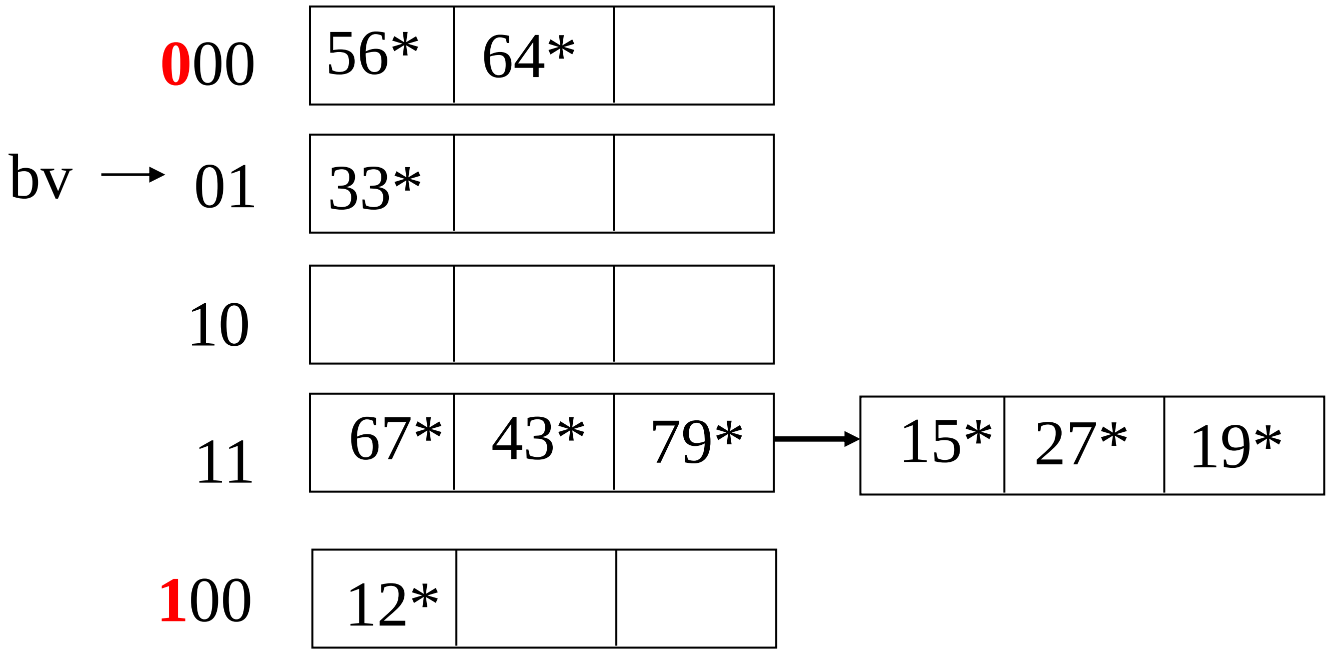
last k digits

Bkfr = 3, desired LF= 2/3,
4 primary area buckets initially

33 = 010 0001

57 = 011 1001
65 = 100 0001
29 = 001 1101

insert



	000	56*	64*				
bv →	01	33*					
	10						
	11	67*	43*	79*	→ 15*	27*	19*
	100	12*					

After Lf * Bkfr = 2/3 * 3 = 2 more data entries, split!

- Add one more bucket to the primary area.
- Distribute the data entries from the bucket chain at the boundary value
- Add 1 to the boundary value.

key < bv → k+1 digits
key ≥ bv → k digits

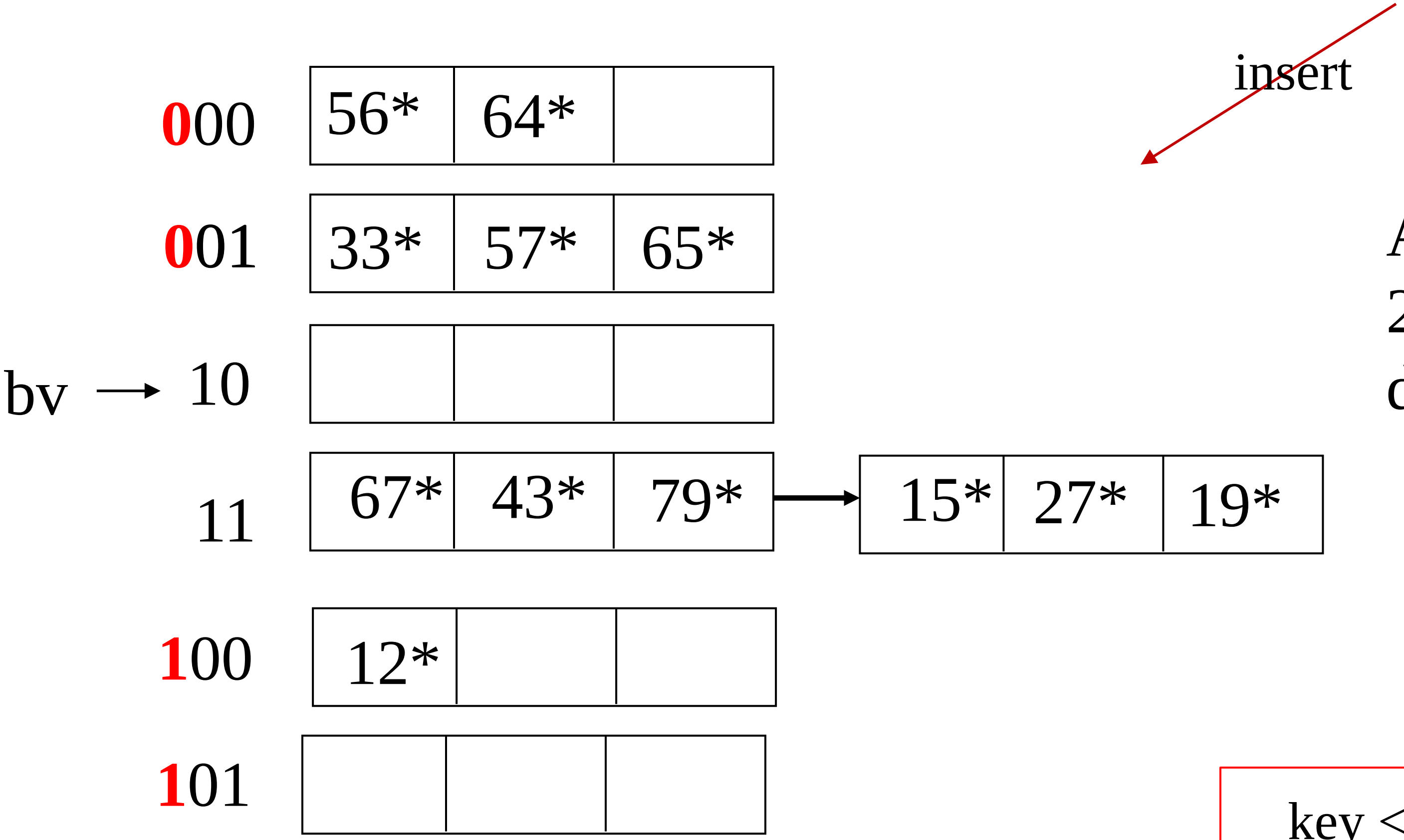
$\underbrace{\hspace{3em}}$
last k digits

Bkfr = 3, desired LF= 2/3,
4 primary area buckets initially

33 = 010 0001

57 = 011 1001
65 = 100 0001

29 = 001 1101
2 = 000 0010



After Lf * Bkfr = 2/3 * 3 = 2 more data entries, split!

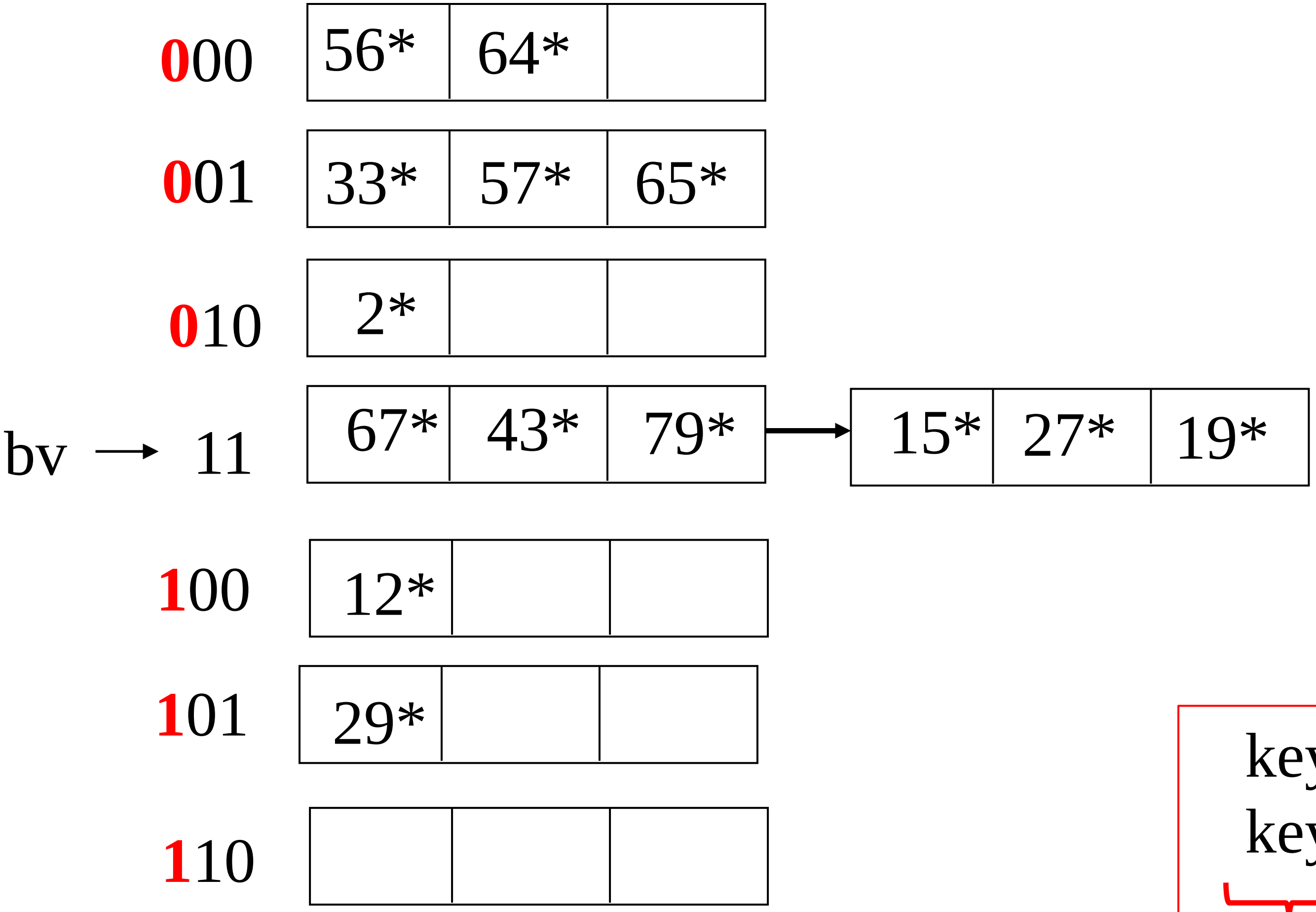
- Add one more bucket to the primary area.
- Distribute the data entries from the bucket chain at the boundary value
- Add 1 to the boundary value.

key < bv → k+1 digits
key ≥ bv → k digits

$\underbrace{\quad\quad\quad}_{\text{last k digits}}$

Bkfr = 3, desired LF= 2/3,
4 primary area buckets initially

29 = 001 1101
2 = 000 0010



After Lf * Bkfr = 2/3 * 3 = 2 more data entries, split!

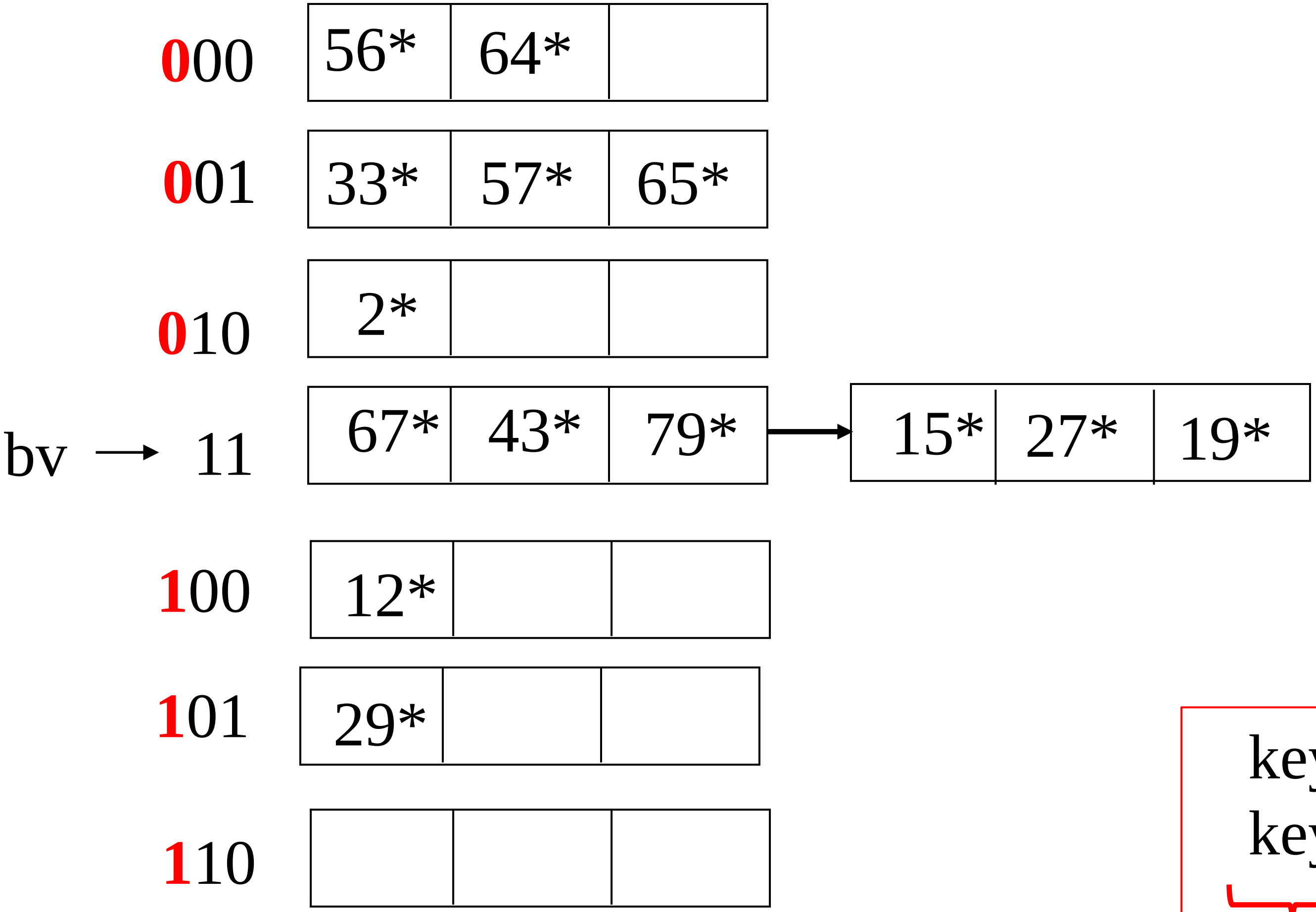
Distribute 2*: stays in 010!

key < bv → k+1 digits
key ≥ bv → k digits

last k digits

Bkfr = 3, desired LF= 2/3,
4 primary area buckets initially

Suppose two more data entries added



$67 = 100\ 0\underline{011}$

$43 = 010\ 1\underline{011}$

$79 = 100\ 1\underline{111}$

15 = 000 1111

27 = 001 1011

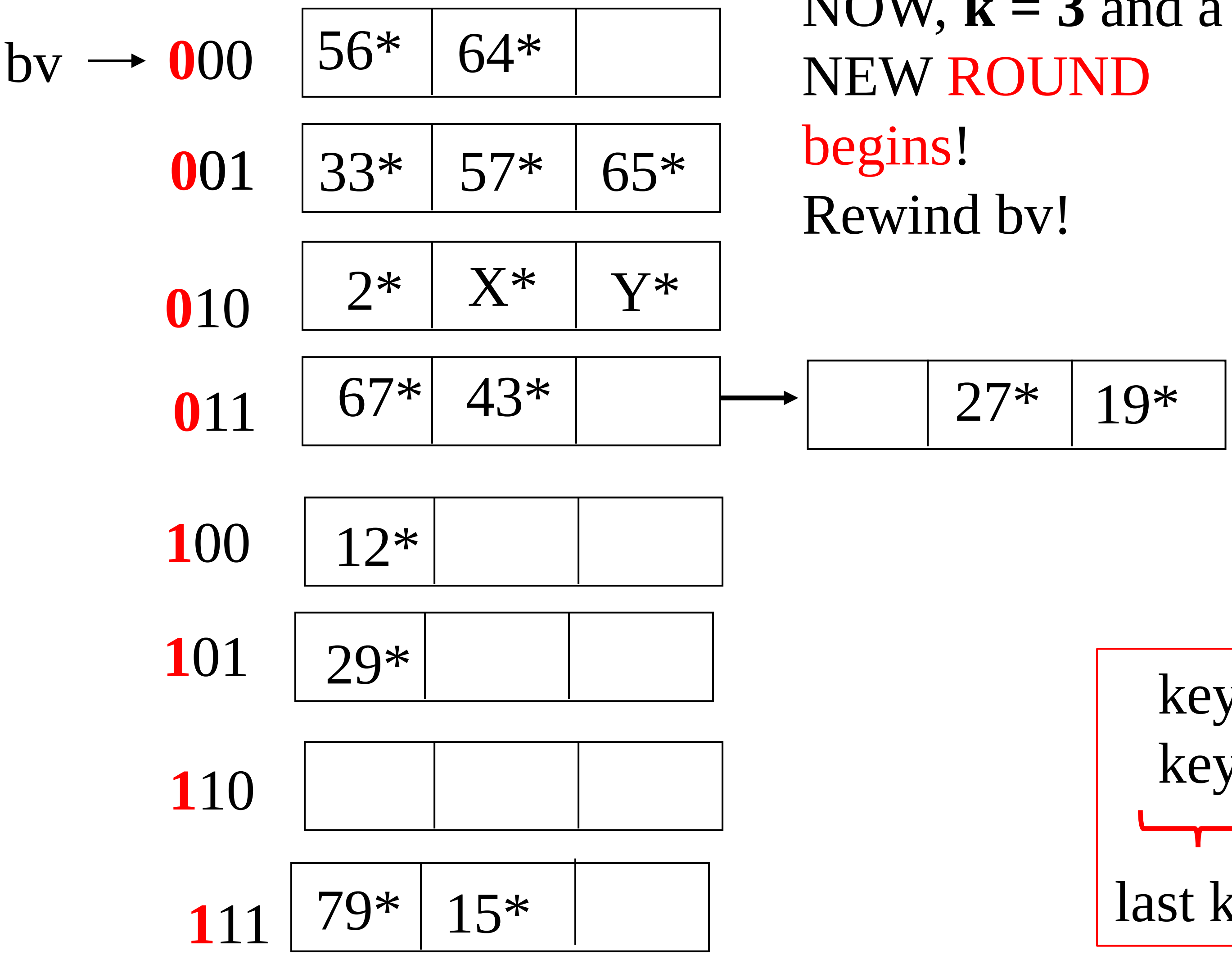
19 = 001 0011

key < bv → k+1 digits

$\underbrace{\text{key}}_{\text{last k digits}} \geq \text{bv} \rightarrow \text{k digits}$

Bkfr = 3, desired LF= 2/3,
4 primary area buckets initially

Suppose two more data entries added

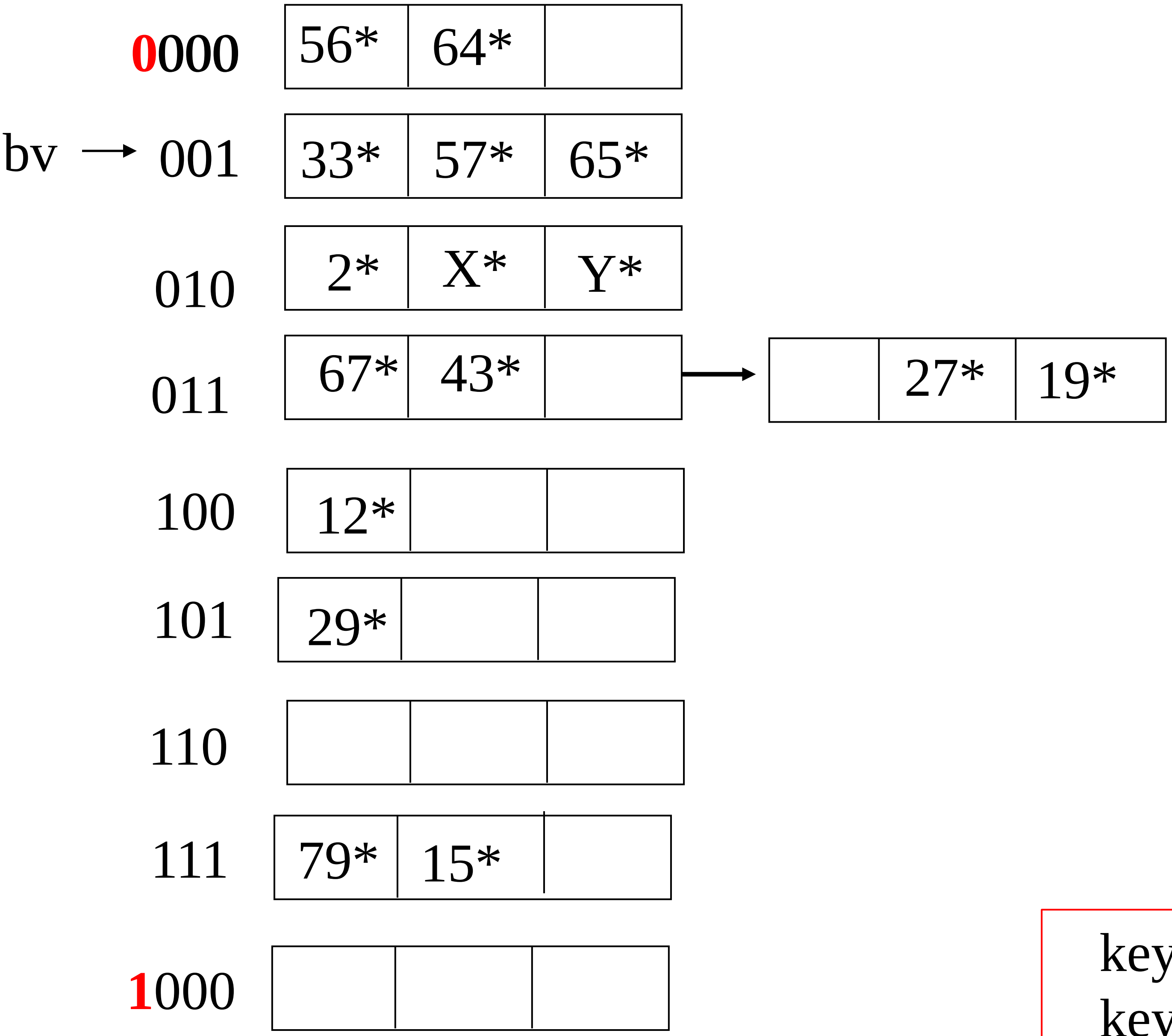


NOW, **k = 3** and a NEW ROUND begins!
Rewind bv!

Distribute:

$67 = 100\ 0\underline{011}$
$43 = 010\ 1\underline{011}$
$79 = 100\ 1\underline{111}$
$15 = 000\ 1111$
$27 = 001\ 1011$
$19 = 001\ 0011$

key < bv → k+1 digits
key ≥ bv → k digits
last k digits



Suppose two
more data
entries added

key < bv → k+1 digits

$\underbrace{\text{key}}_{\text{last k digits, k=3}}$ ≥ bv → k digits

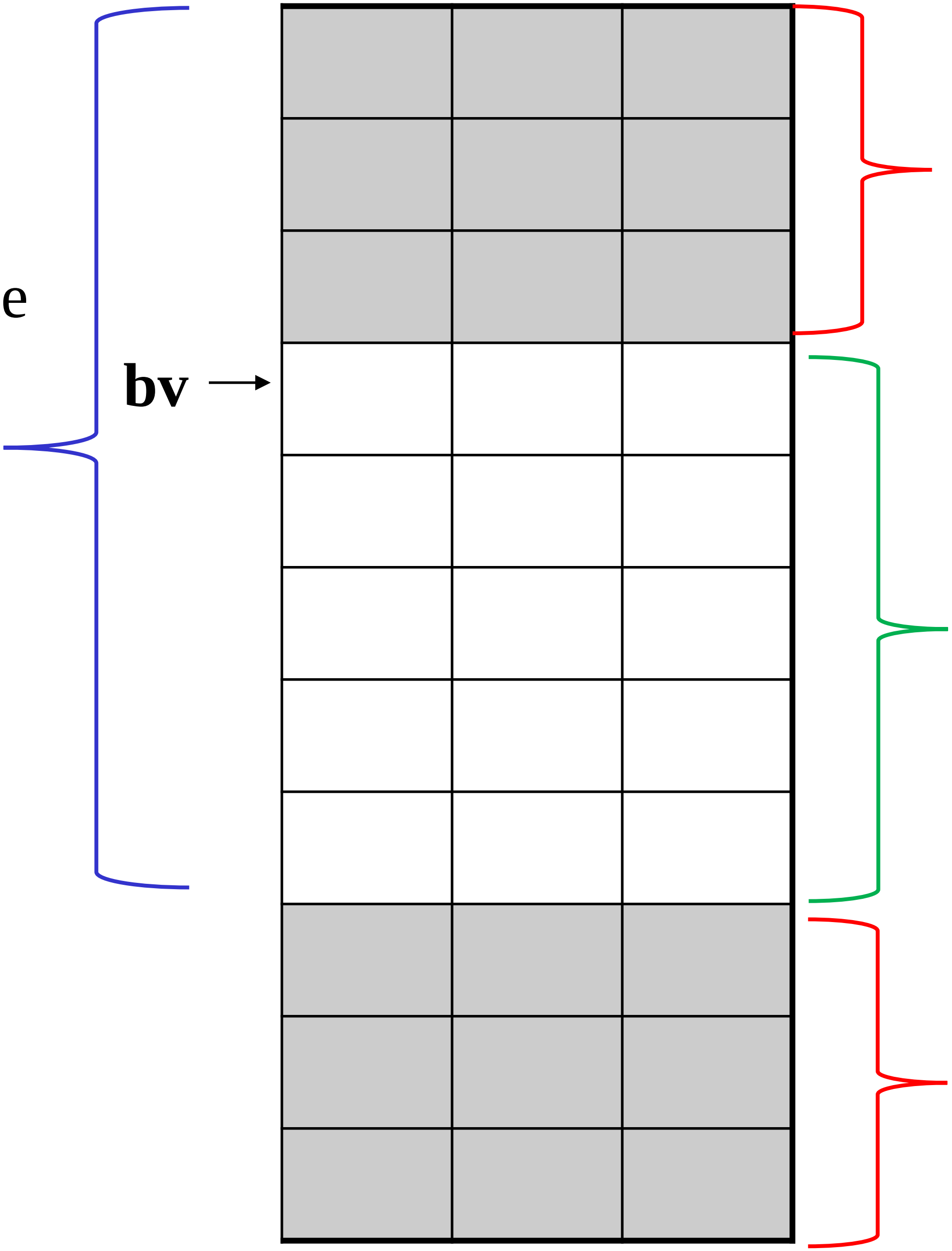
Linear Hashing

- Overflow and expansion are independent events
  - Overflow does not trigger bucket split
  - Bucket split does not necessarily remove an overflow chain (but eventually the overflowing bucket will also be split!)
- After splitting all buckets in a **round**, **rewind bv to 0**, **increase k by 1**
  - At this point, we **doubled** the **range** into which keys are hashed!

Linear Hashing

- No reorganization
- It maintains a **constant load factor.**
- There are still overfow chains (hopefully, won’t be too long, as the overflowing bucket will also be split eventually)
- Data entry fetch time → still close to **1 disk access**

Buckets during a Round in Linear Hashing



Buckets existing at the beginning of this round

bv →

Buckets split in this round

k+1 digits

Buckets not split yet in this round

k digits

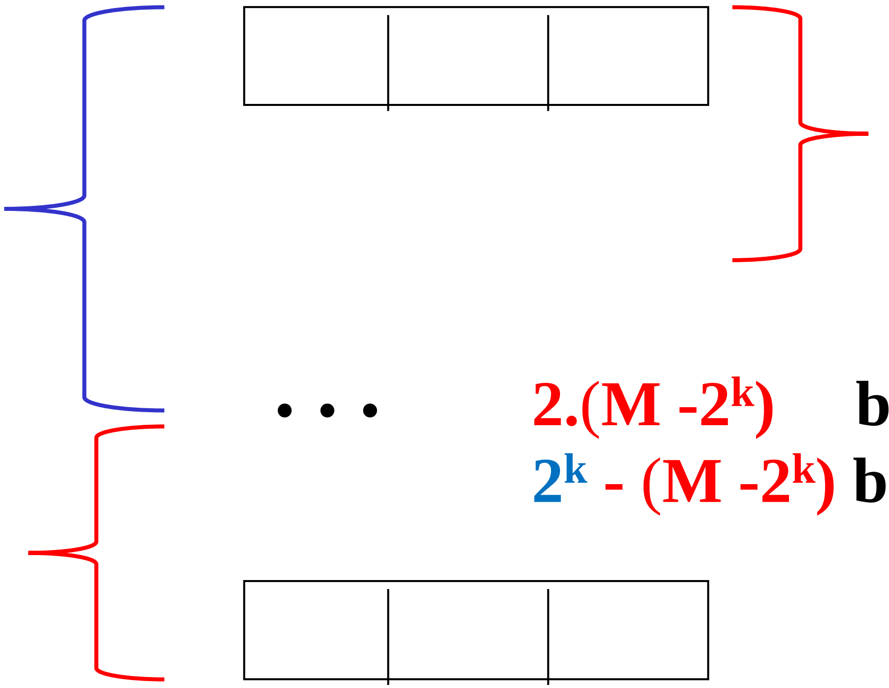
Split image buckets in this round

k+1 digits

Buckets during a Round in Linear Hashing

Buckets during a Round in Linear Hashing

- Suppose we have **M** primary area buckets during a round that has started with **k** digits
  - What can you say about the quantity of M?
  - $2^k \leq M < 2^{k+1} \Rightarrow k \leq \log M < k+1$



Existing at the beginning of this round

2^k

Split buckets

$M - 2^k$

. . .

$2.(M - 2^k)$ buckets use k+1 digits

$2^k - (M - 2^k)$ buckets use k digits

Split images

$M - 2^k$

M Buckets

Deletion

- Read in a chain of data entries.
- Replace the deleted data entry with the last data entry in the chain.
    - If the last overflow bucket becomes empty, deallocate it.
- When the number of data entries is Lf * Bkfr less than the number needed for Lf, contract the primary area by one bucket.

Compressing the file is exact opposite of expanding it:

- Keep the total # of data entries in the file and buckets in primary area.
- When we have Lf * Bkfr fewer data entries than needed, consolidate the last bucket with the bucket which shares the same last k digits.