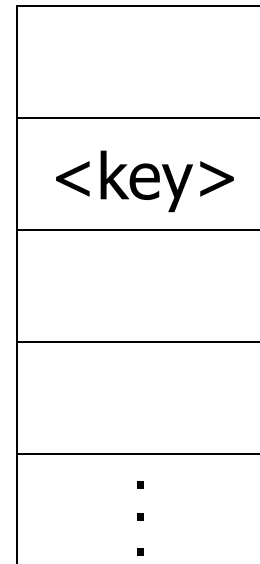
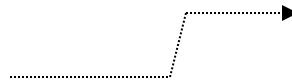


Ullman et al. :  
Database System Principles

**Notes 5: Hashing and More**

# Hashing

$\text{key} \rightarrow h(\text{key})$



← Buckets  
(typically 1  
disk block)

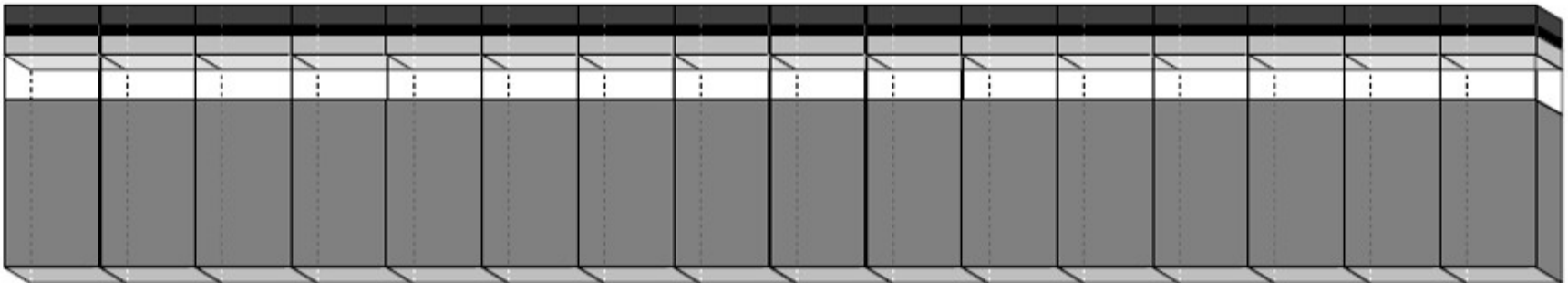
# Hashing

```
SELECT * FROM employees  
WHERE department_id = 20
```

↓  
Hash(20) → Hash Value 77

↓  
Block  
100

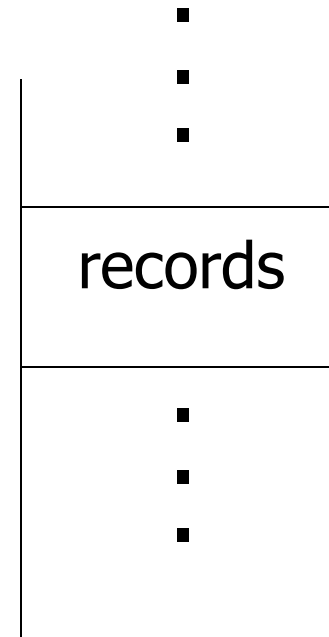
Data Blocks



# Two alternatives

(1)  $\text{key} \rightarrow \text{h}(\text{key})$

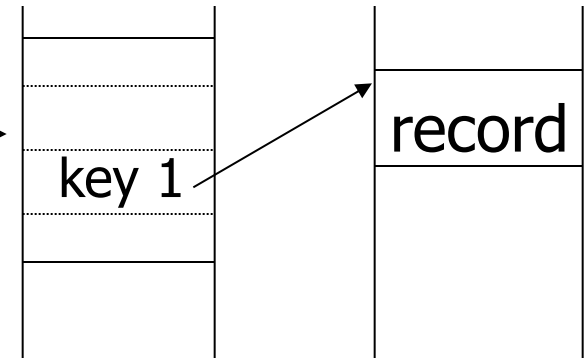
(direct reference, **not flexible**)



## Two alternatives

(2)  $\text{key} \rightarrow h(\text{key})$

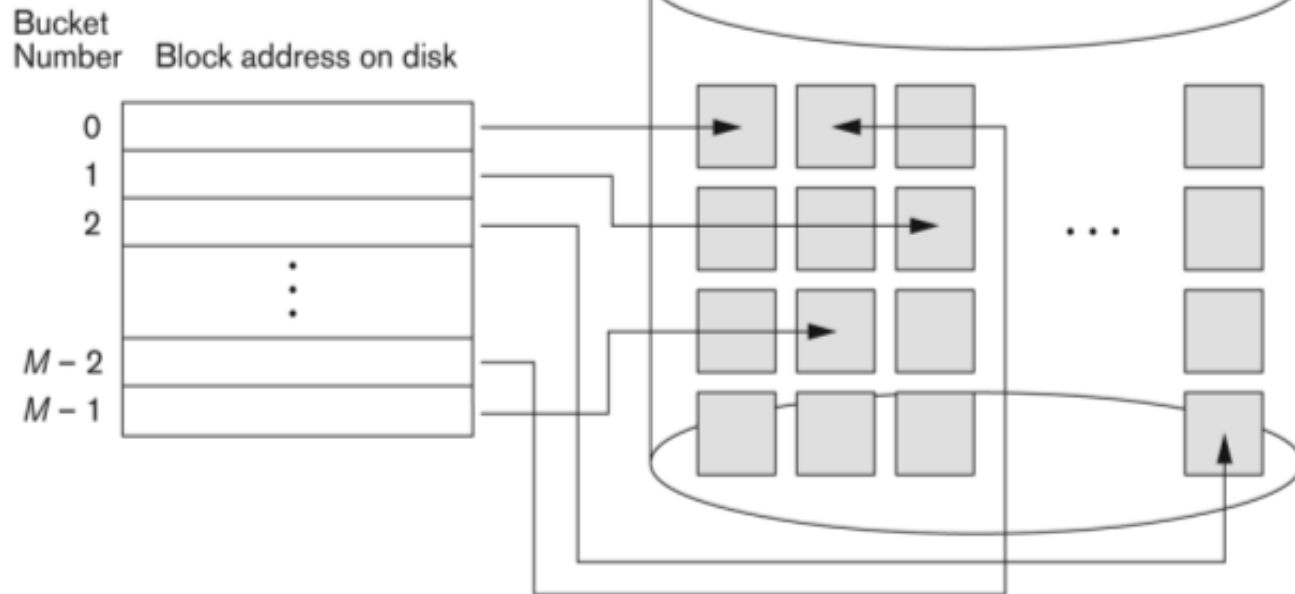
(indirect reference, **more flexible**)



- Alt (2) for “secondary” search key

# Typical implementation

Matching bucket numbers to disk block addresses.



## Example hash function

- Key = ' $x_1 x_2 \dots x_n$ '  $n$  byte character string
- Have  $b$  buckets
- $h$ : add  $x_1 + x_2 + \dots + x_n$ 
  - compute sum modulo  $b$

- ☒ This may not be best function ...
- ☒ Read Knuth Vol. 3 if you really need to select a good function.

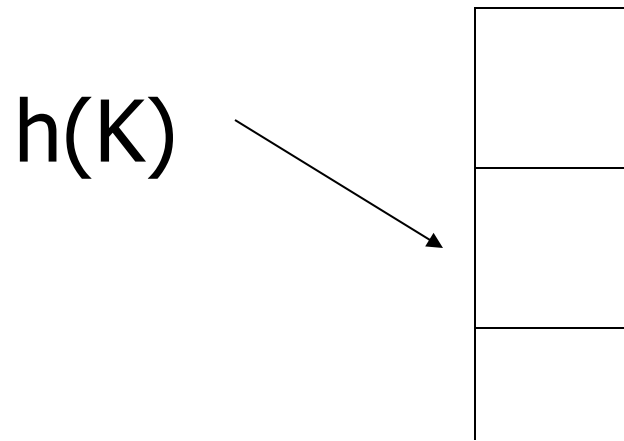
Good hash function:       Expected number of **keys/bucket** is the same for all buckets



## Within a bucket:

- Do we keep keys sorted?
- Yes, if CPU time critical  
& Inserts/Deletes not too frequent

Next: example to illustrate  
inserts, overflows, deletes



## EXAMPLE 2 records/bucket

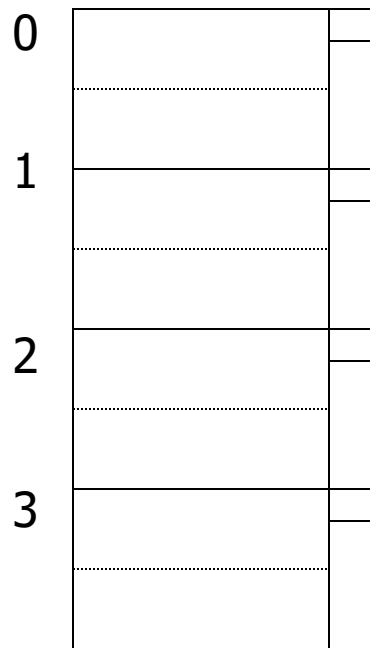
INSERT:

$$h(a) = 1$$

$$h(b) = 2$$

$$h(c) = 1$$

$$h(d) = 0$$



## EXAMPLE 2 records/bucket

INSERT:

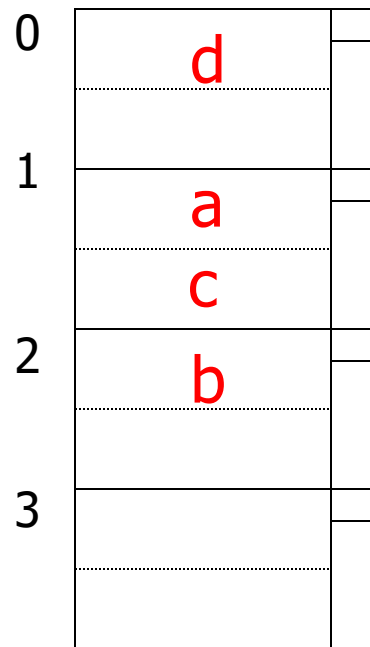
$h(a) = 1$

$h(b) = 2$

$h(c) = 1$

$h(d) = 0$

$h(e) = 1$



## EXAMPLE 2 records/bucket

INSERT:

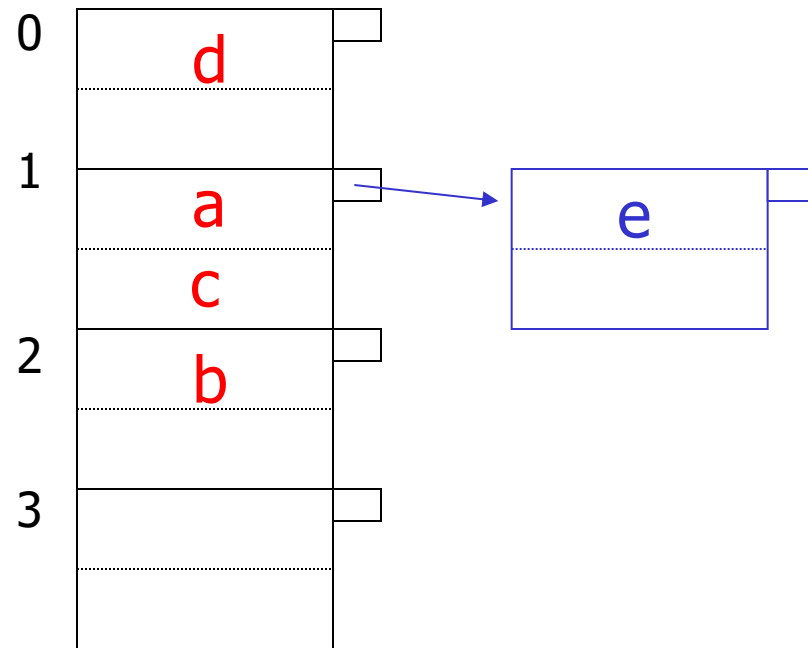
$h(a) = 1$

$h(b) = 2$

$h(c) = 1$

$h(d) = 0$

$h(e) = 1$

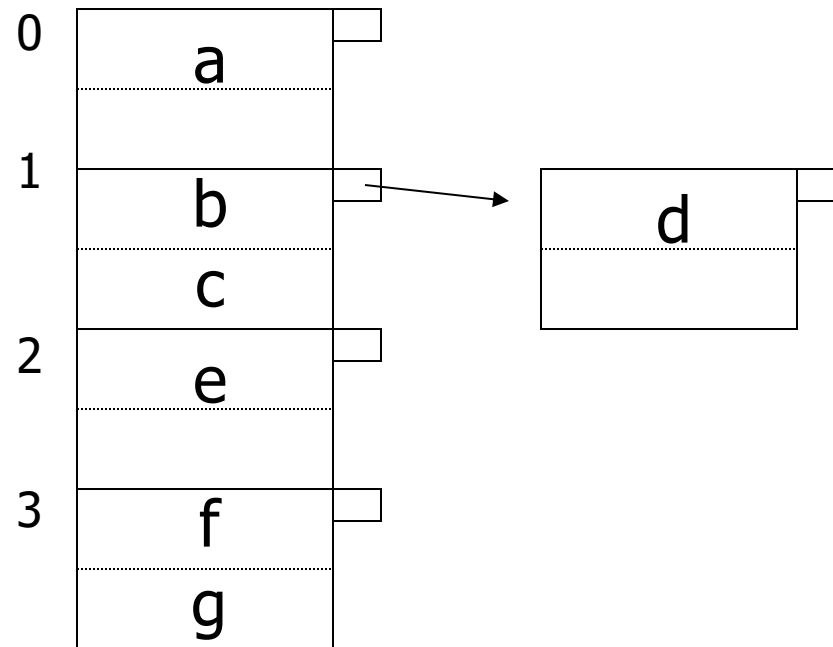


# EXAMPLE: deletion

Delete:

e

f



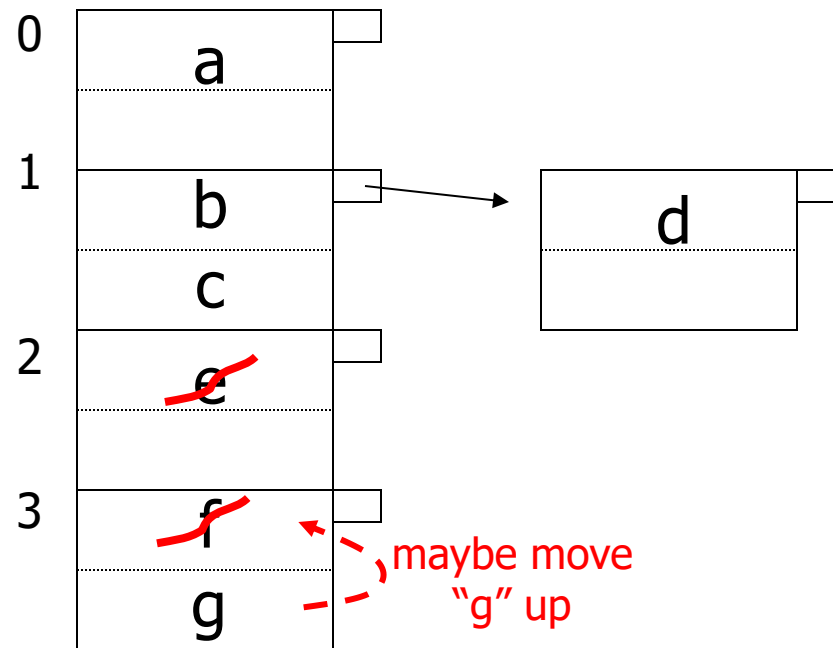
# EXAMPLE: deletion

Delete:

e

f

c



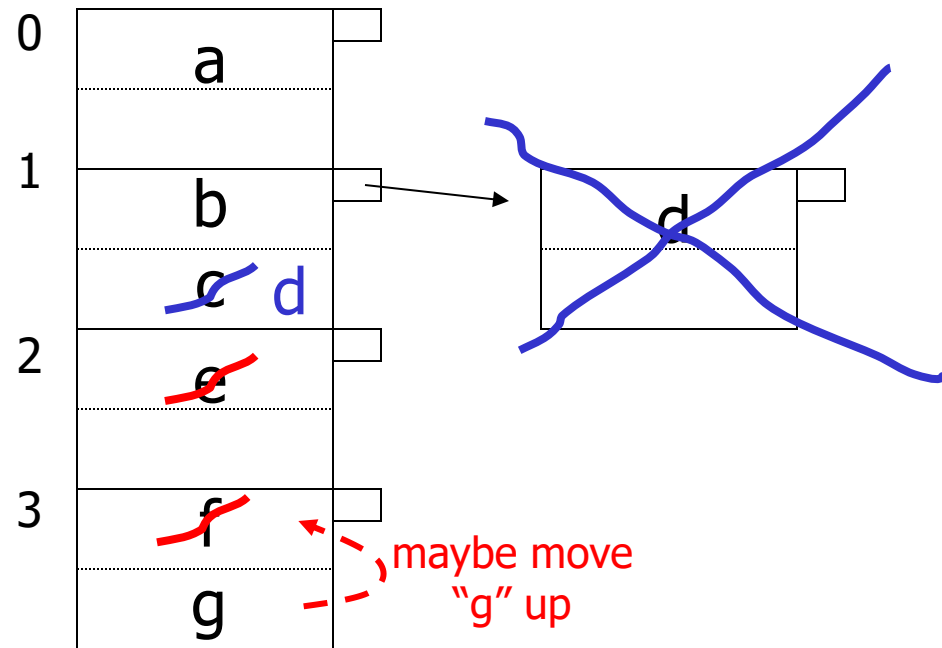
# EXAMPLE: deletion

Delete:

e

f

c





## Rule of thumb:

- Try to keep space utilization between 50% and 80%

$$\text{Utilization} = \frac{\text{\# keys used}}{\text{total \# keys that fit}}$$

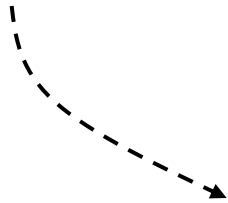
- If  $< 50\%$ , wasting space
- If  $> 80\%$ , overflows significant  
    ↖ depends on how good hash function is & on # keys/bucket

# How do we cope with growth?

- Overflows and reorganizations
- Dynamic hashing

# How do we cope with growth?

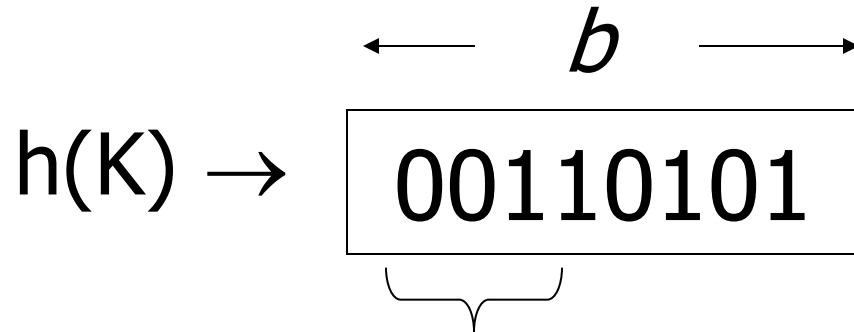
- Overflows and reorganizations
- **Dynamic hashing**



- Extensible
- Linear

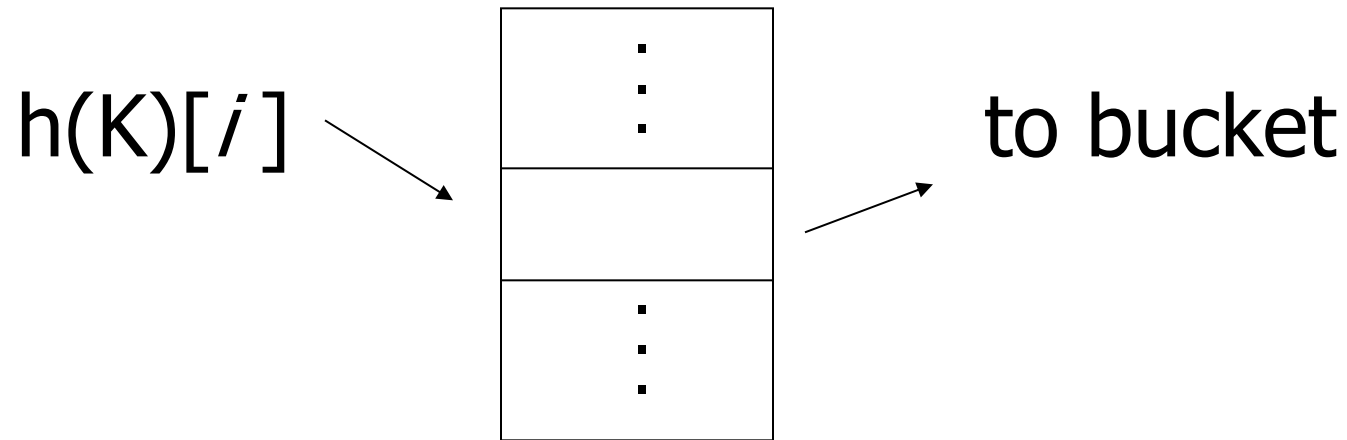
## Extensible hashing: two ideas

(a) Use  $i$  of  $b$  bits output by hash function



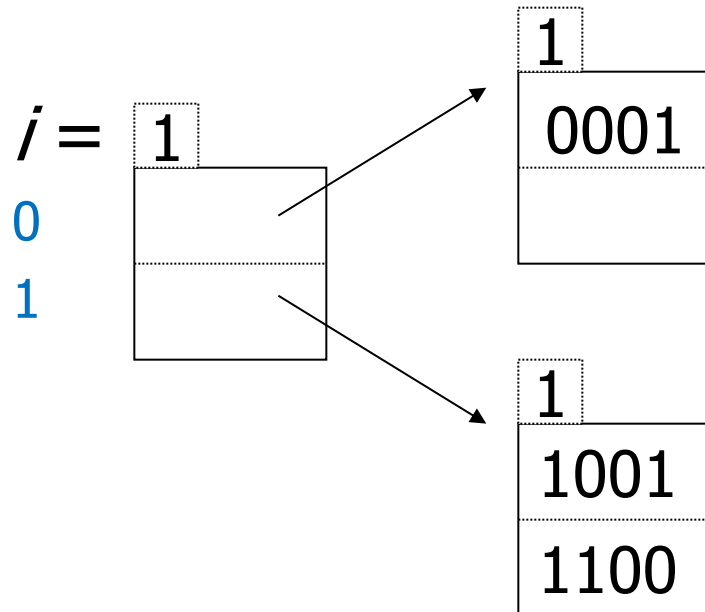
use  $i \rightarrow$  grows over time....

## (b) Use directory



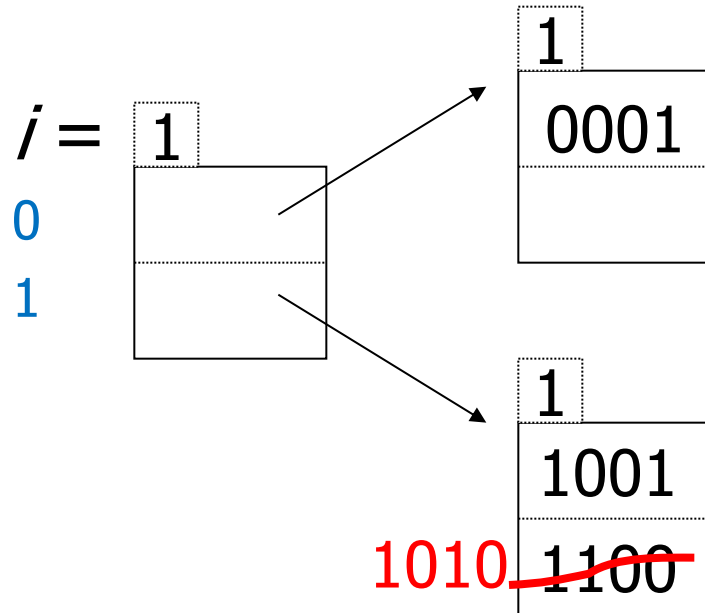
$h(K)[i]$ : means the first  $i$  bits of the output by hash function

Example:  $h(k)$  is 4 bits; 2 keys/bucket

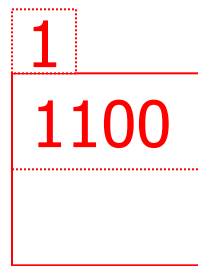


Insert 1010

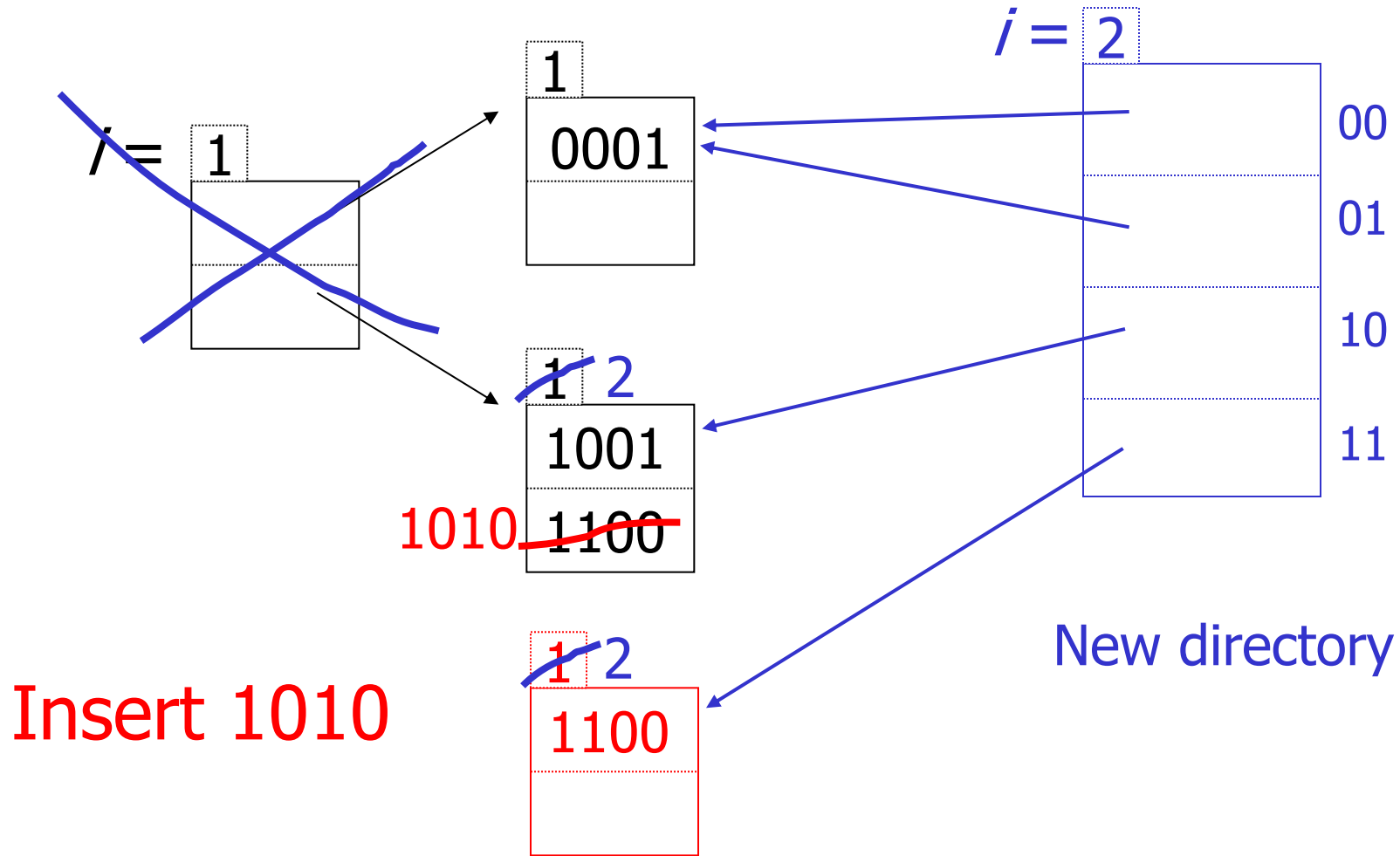
Example:  $h(k)$  is 4 bits; 2 keys/bucket



Insert 1010

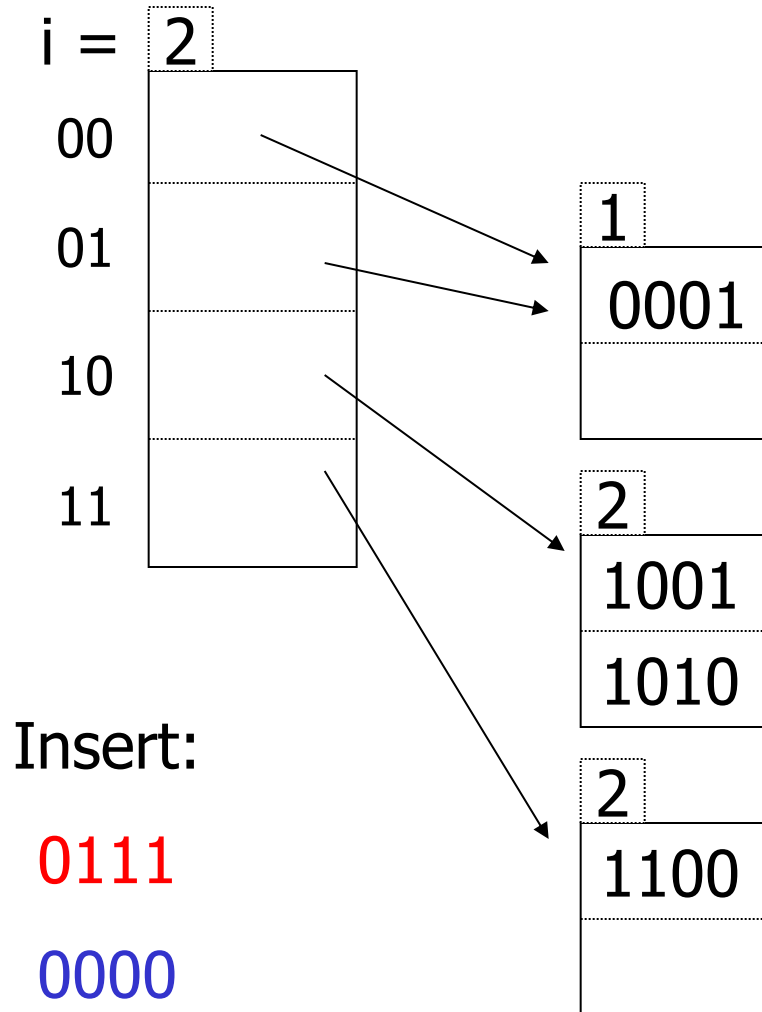


Example:  $h(k)$  is 4 bits; 2 keys/bucket

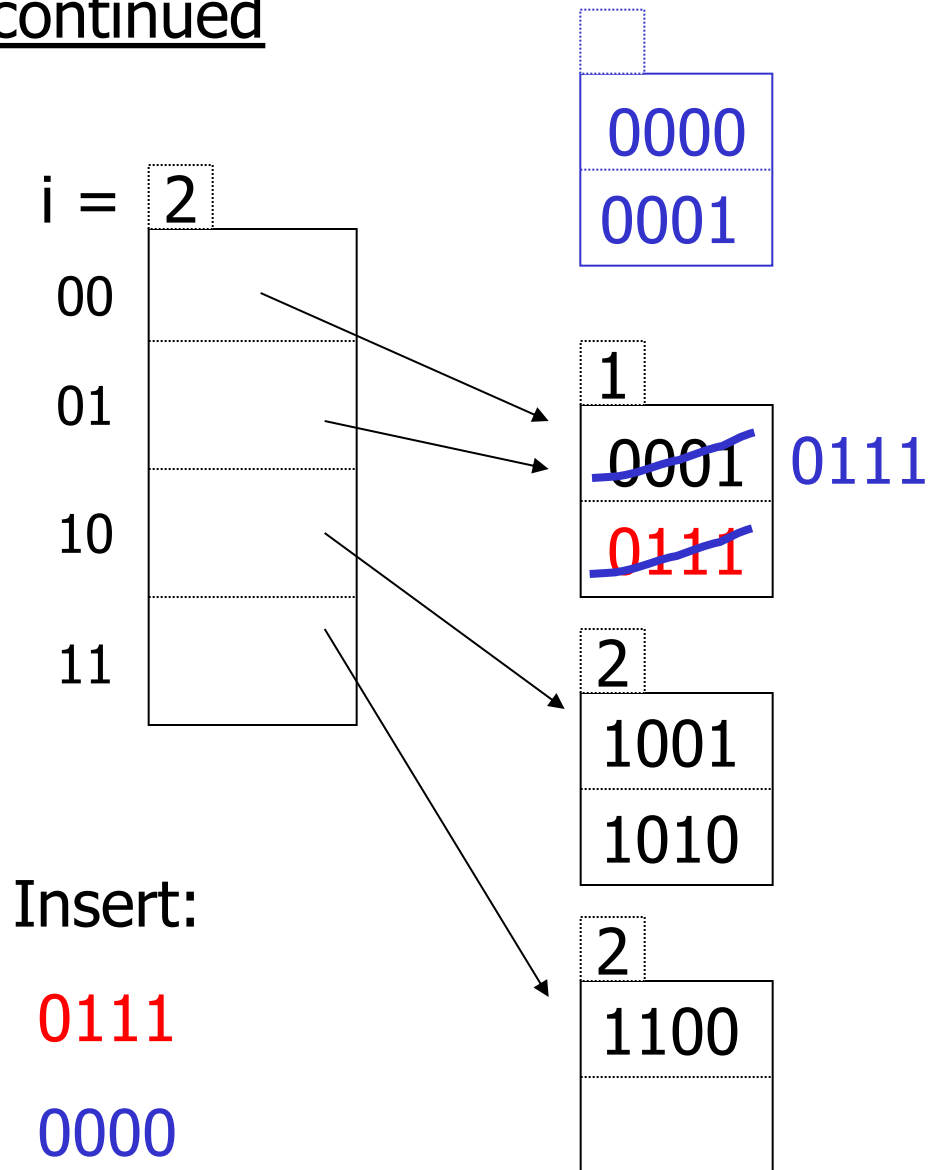




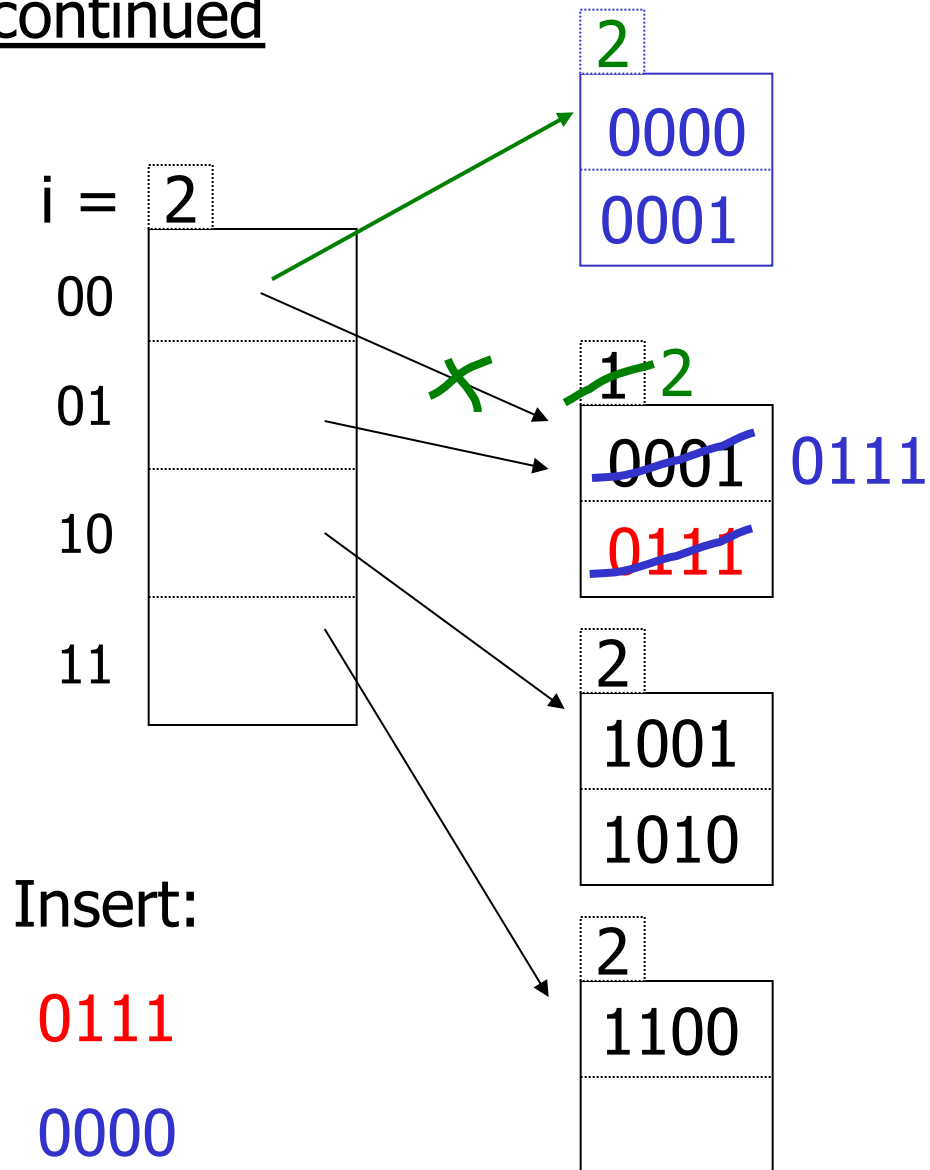
## Example continued



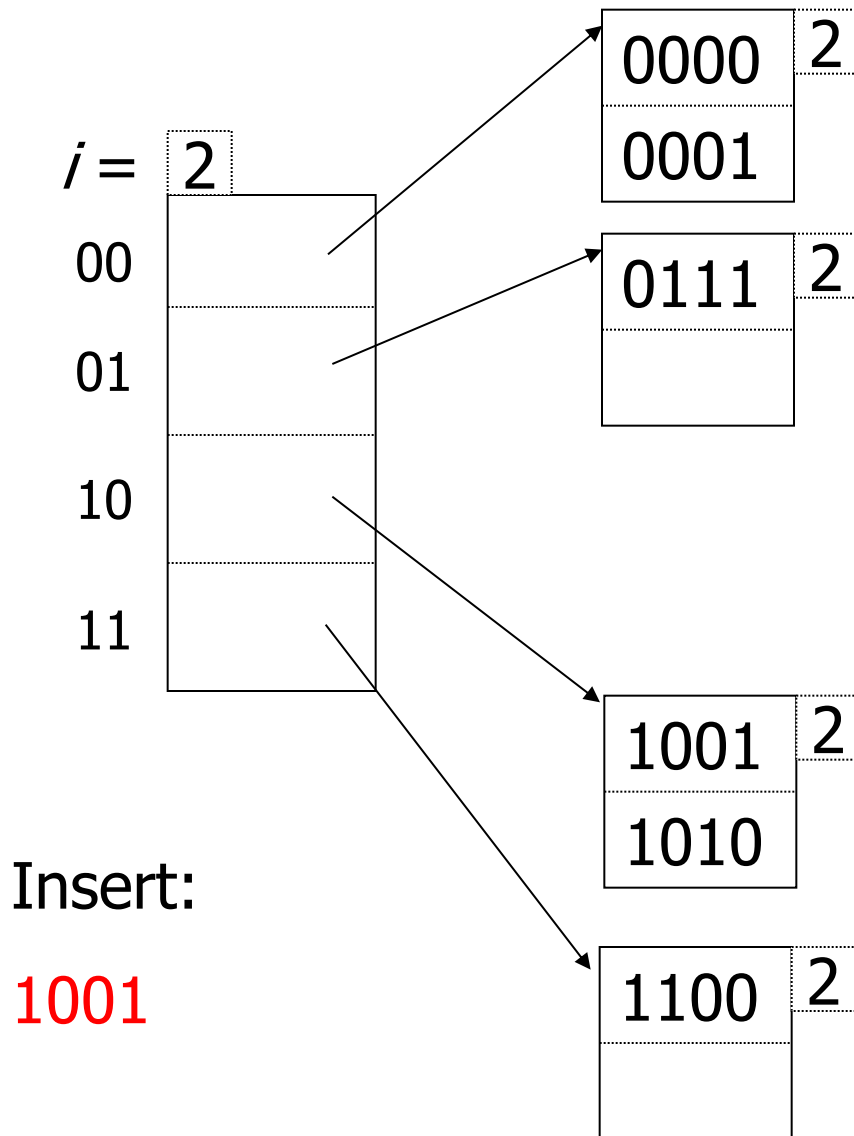
## Example continued



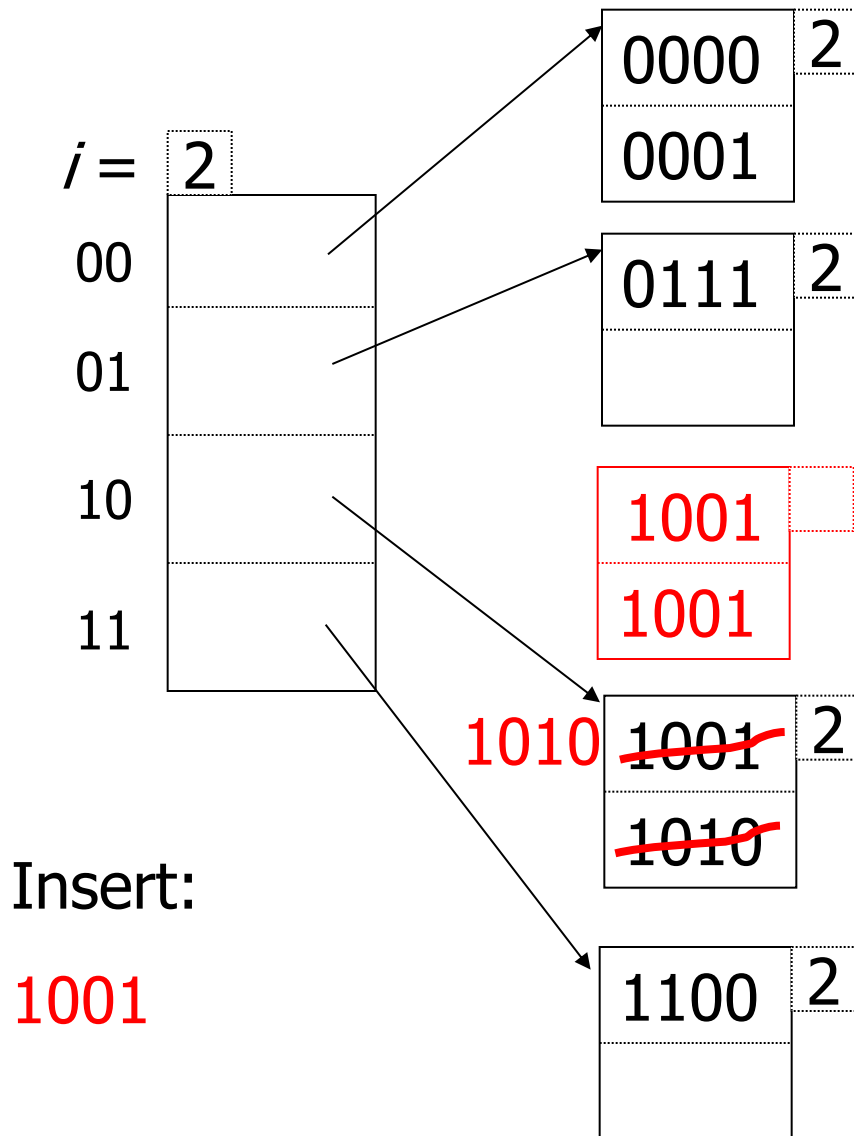
## Example continued



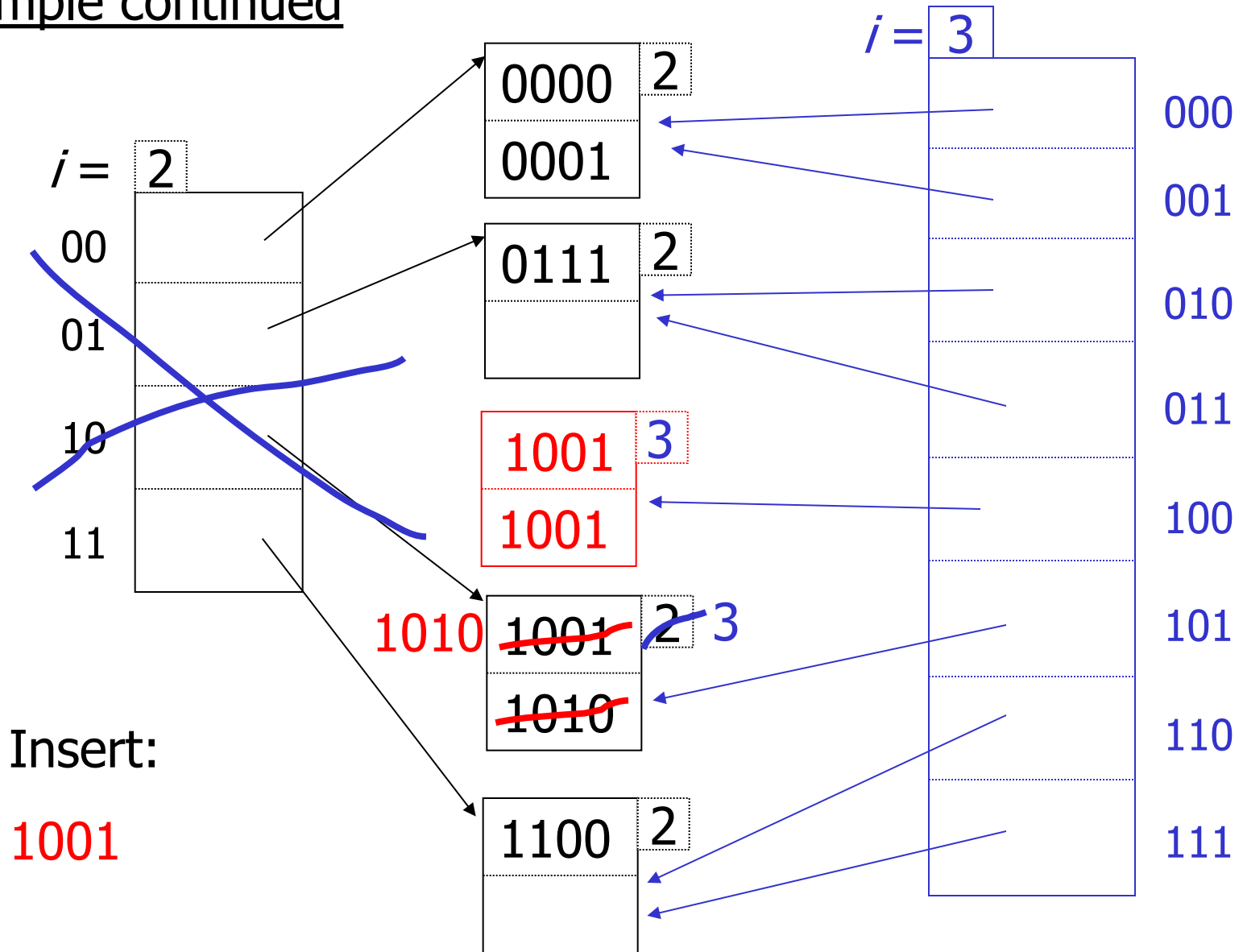
## Example continued




## Example continued



## Example continued



# Extensible hashing: deletion

- 
- No merging of blocks
  - Merge blocks  
and cut directory if possible  
(Reverse insert procedure)

## Deletion example:

- Run thru insert example in reverse!

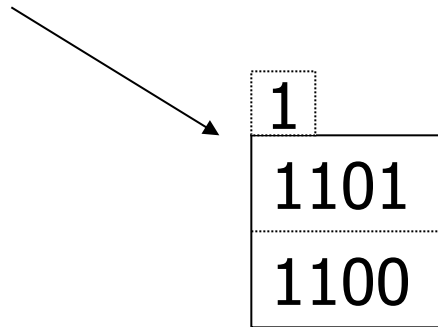
But: Typically **not implemented**



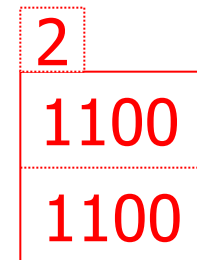
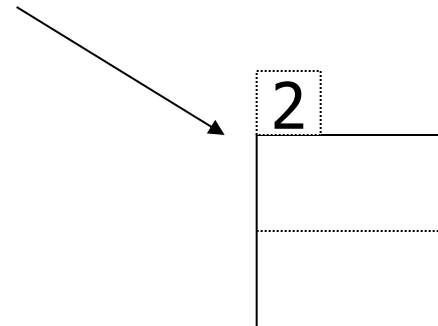
# Note: Still need overflow chains

- Example: many records with duplicate keys

insert 1100

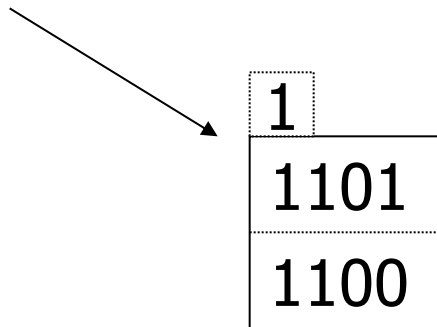


if we split:

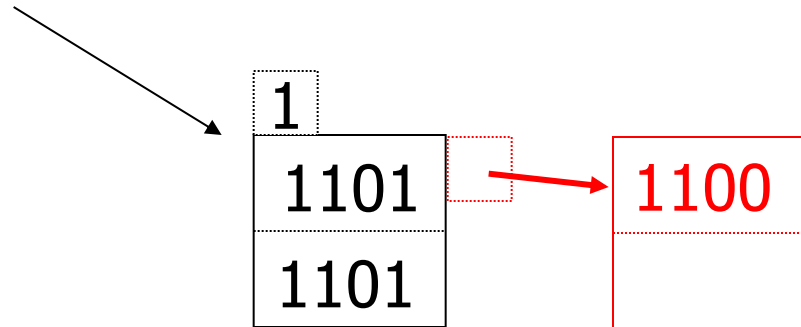


# Solution: overflow chains

insert 1100



add overflow block:



## Summary

## Extensible hashing

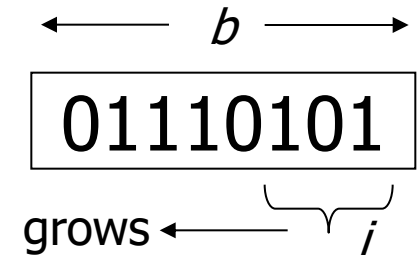
- ⊕ Can handle growing files
  - with less wasted space
  - with no full reorganizations
- ⊖ Indirection
  - (Not bad if directory in memory)
- ⊖ Directory doubles in size
  - (Now it fits, now it does not)

# Linear hashing

- Another dynamic hashing scheme

Two ideas:

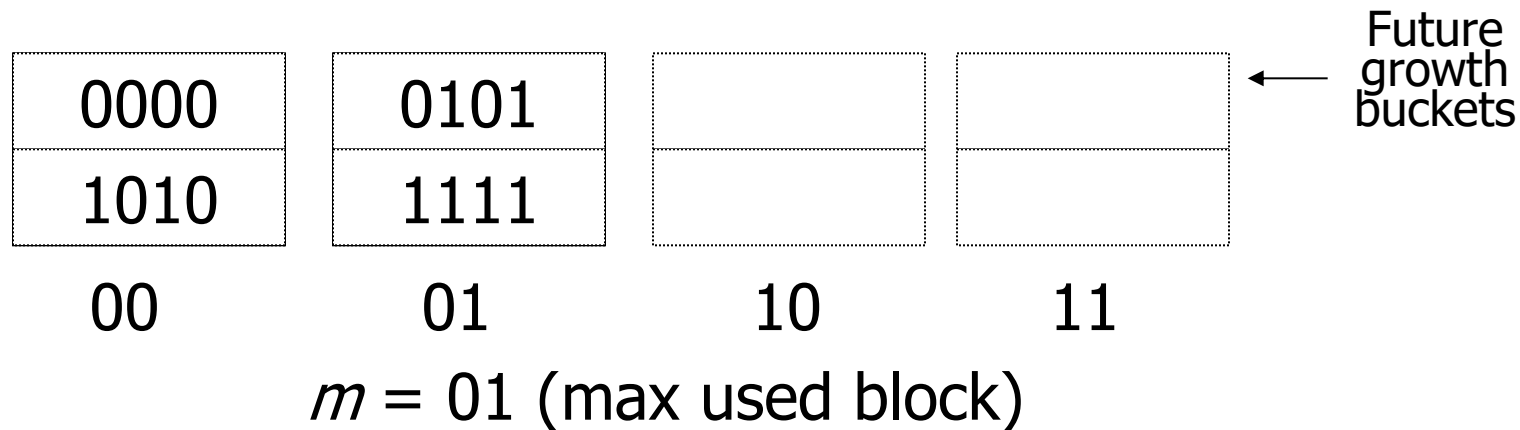
(a) Use  $i$  low order bits of hash



(b) File grows **linearly**



Example  $b=4$  bits,  $i=2$ , 2 keys/bucket



**Rule**

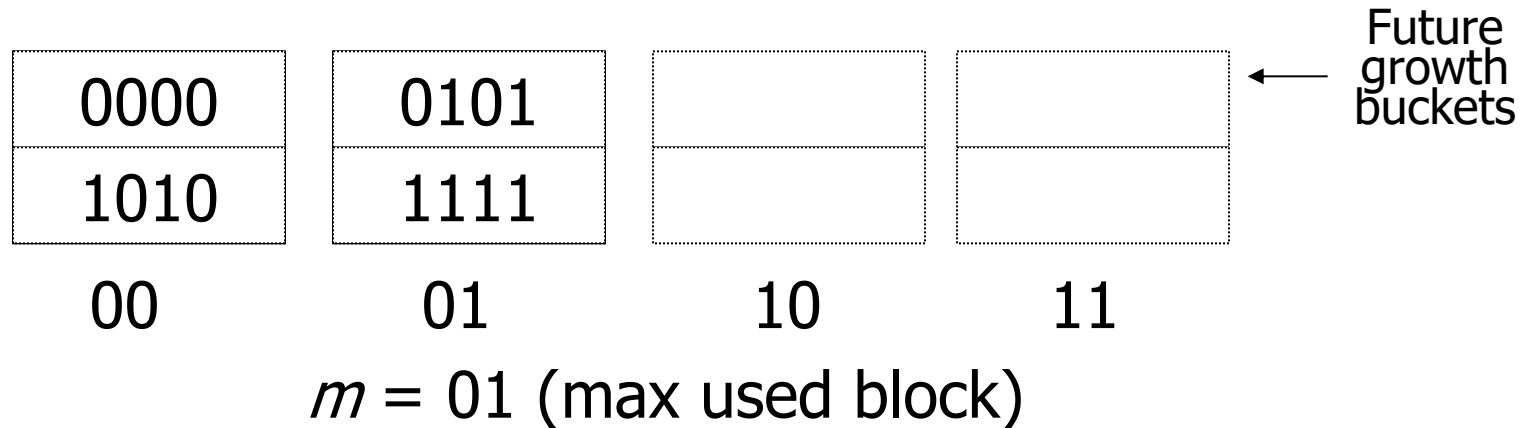
If  $h(k)[i] \leq m$ , then

look at bucket  $h(k)[i]$

else, look at bucket  $h(k)[i] - 2^{i-1}$

Example  $b=4$  bits,  $i=2$ , 2 keys/bucket

- insert 0101



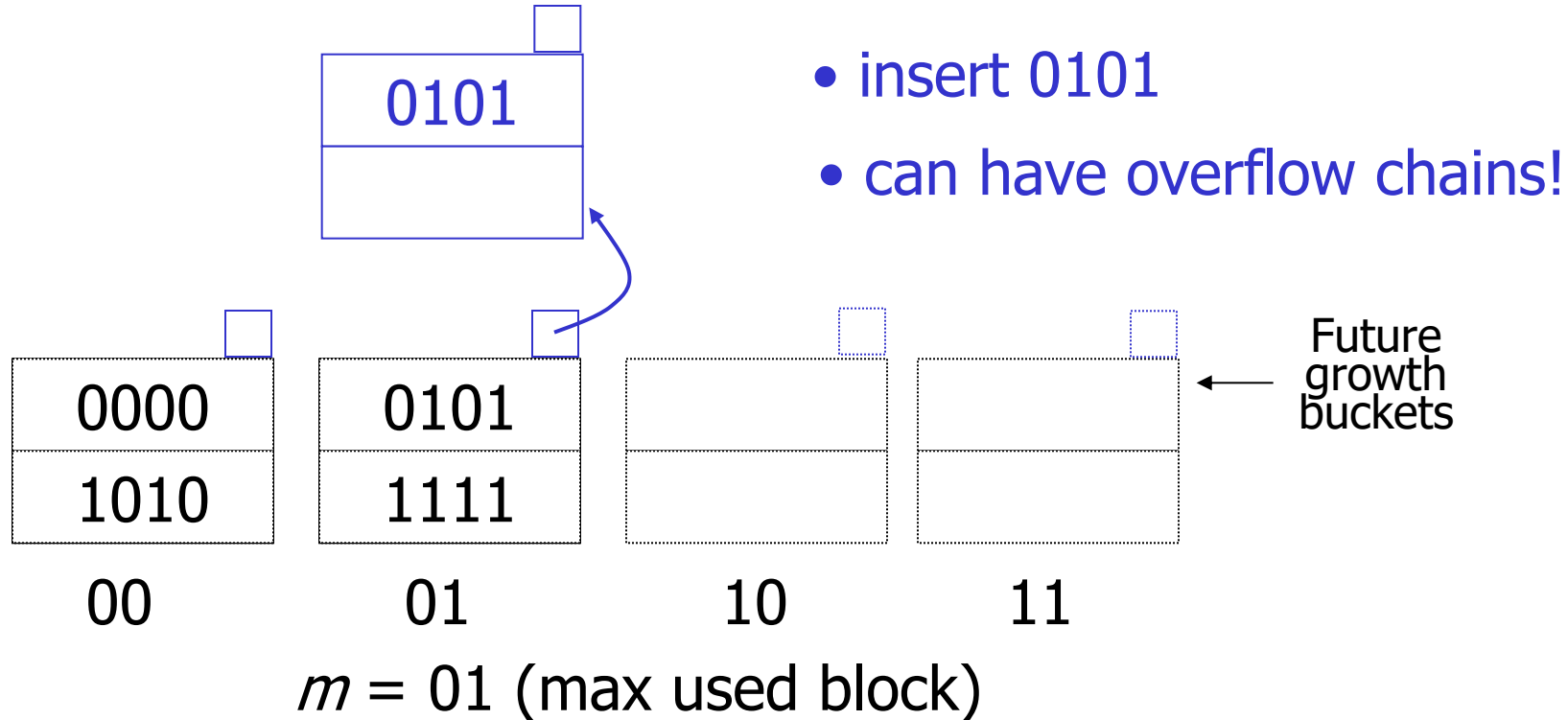
**Rule**

If  $h(k)[i] \leq m$ , then

look at bucket  $h(k)[i]$

else, look at bucket  $h(k)[i] - 2^{i-1}$

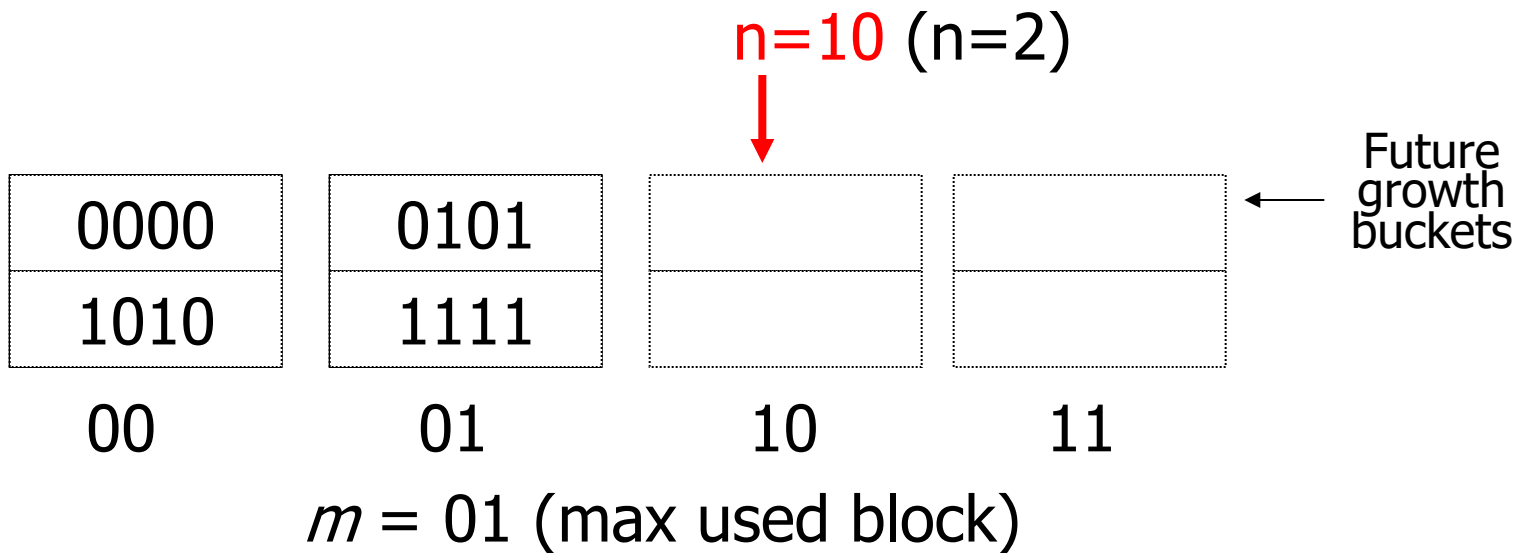
Example  $b=4$  bits,  $i=2$ , 2 keys/bucket



**Rule** If  $h(k)[i] \leq m$ , then  
look at bucket  $h(k)[i]$   
else, look at bucket  $h(k)[i] - 2^{i-1}$

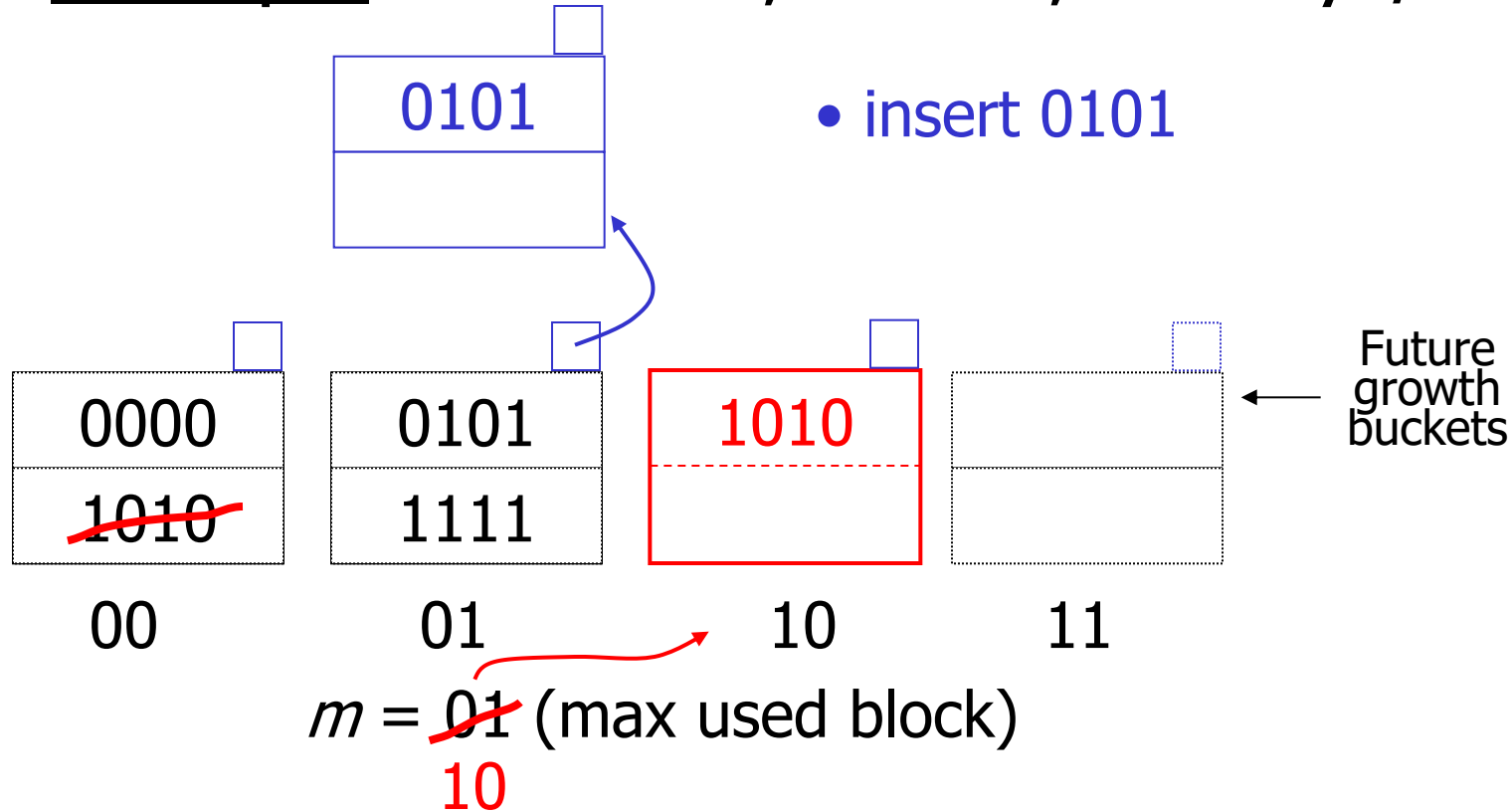
# Note

- In textbook, **n** is used **instead of m**
- $n = m + 1$  ( $n$  = number of used blocks)

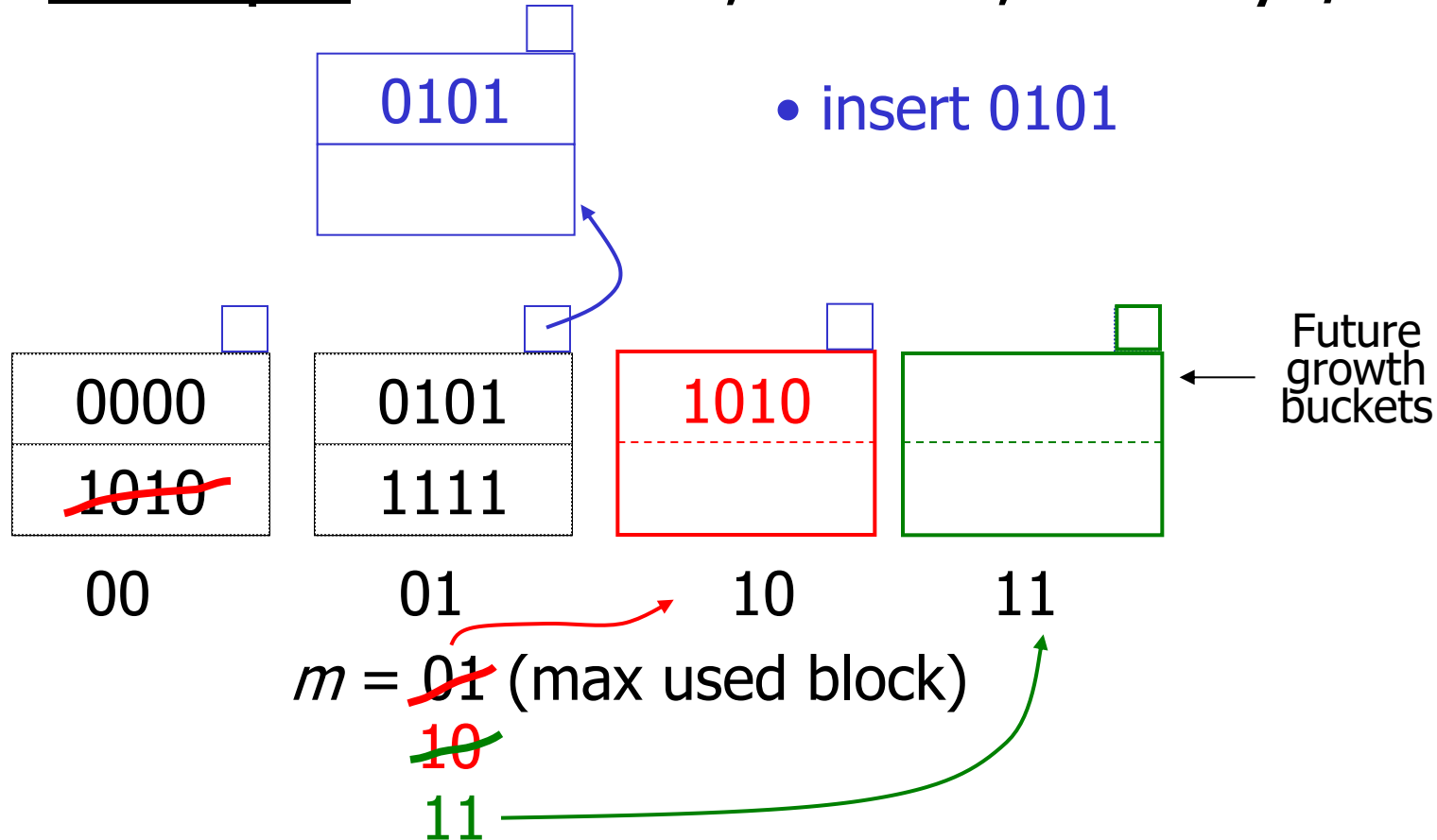




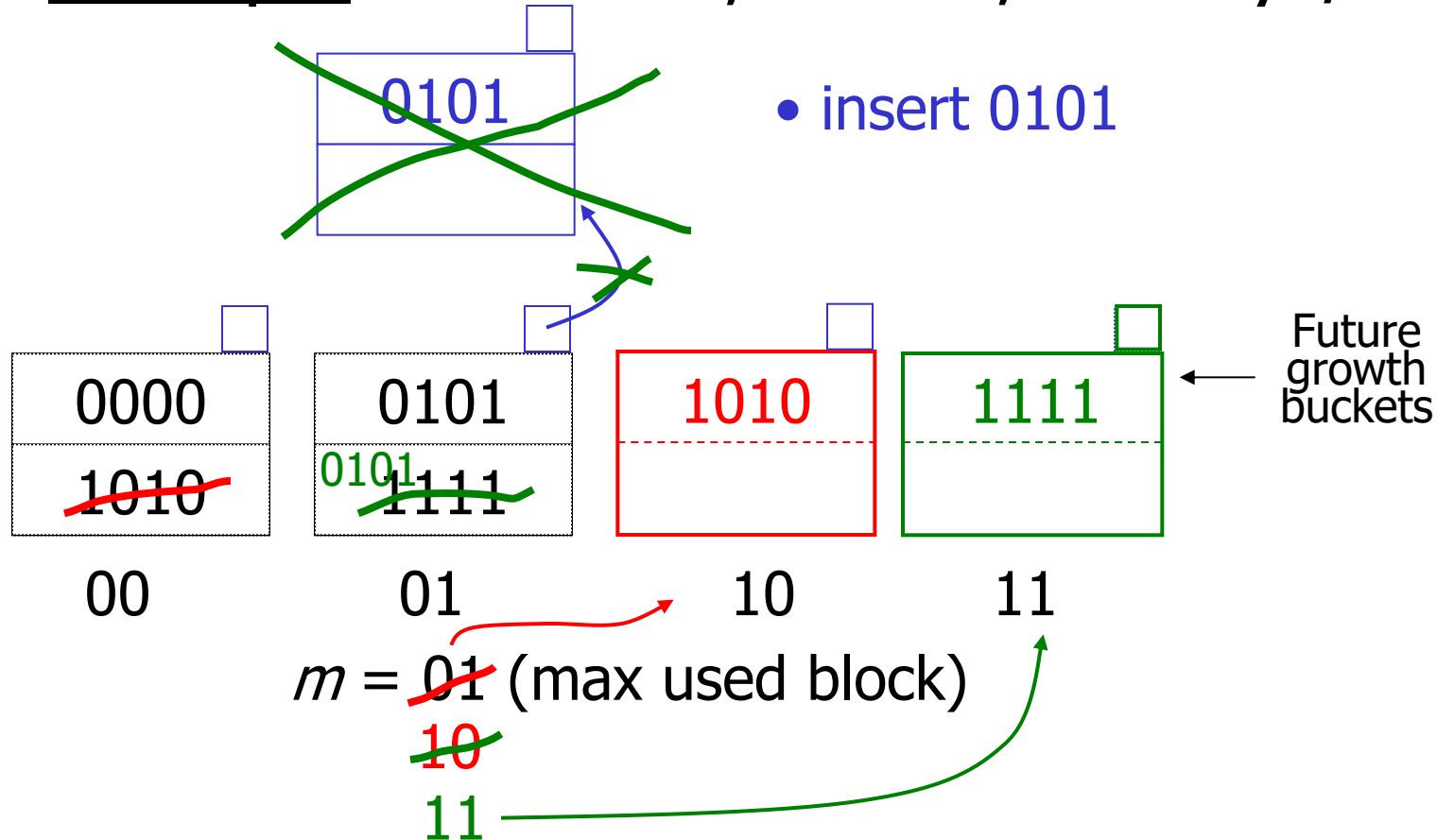
Example  $b=4$  bits,  $i=2$ , 2 keys/bucket



Example  $b=4$  bits,  $i=2$ , 2 keys/bucket



Example  $b=4$  bits,  $i=2$ , 2 keys/bucket



## Example Continued: How to grow beyond this?

$i = 2$

0000	0101	1010	1111
	0101		
00	01	10	11

...

$m = 11$  (max used block)

# Example Continued: How to grow beyond this?

$i =$ ~~2~~ 3

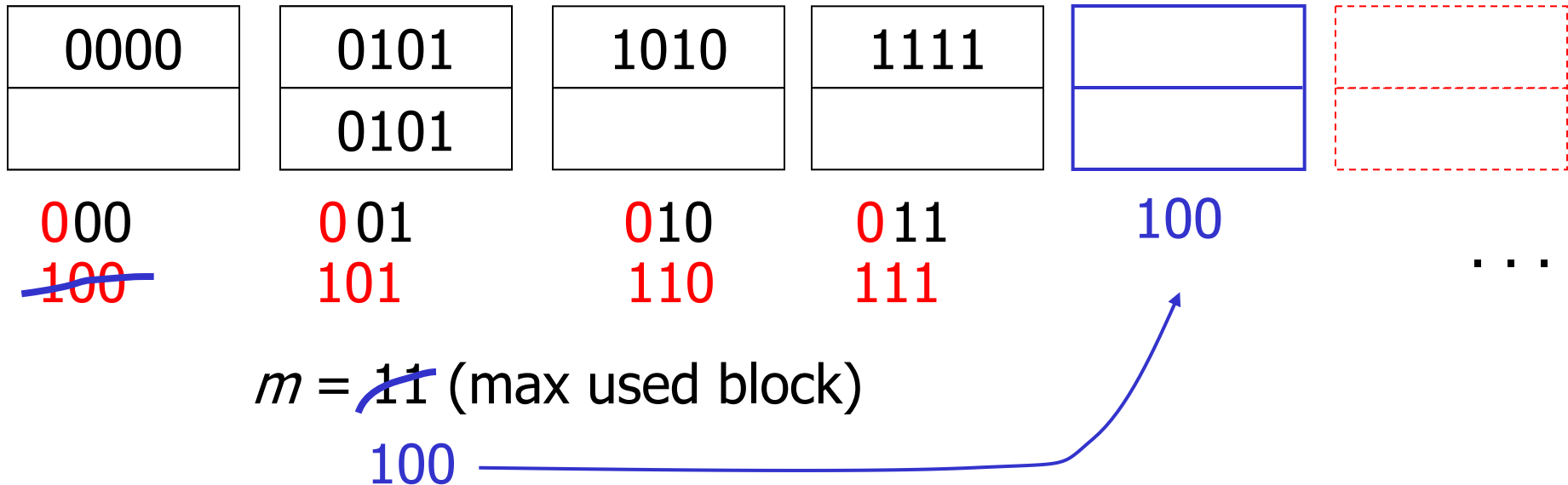
0000	0101	1010	1111		
	0101				
000	001	010	011		
100	101	110	111		

...

$m = 11$  (max used block)

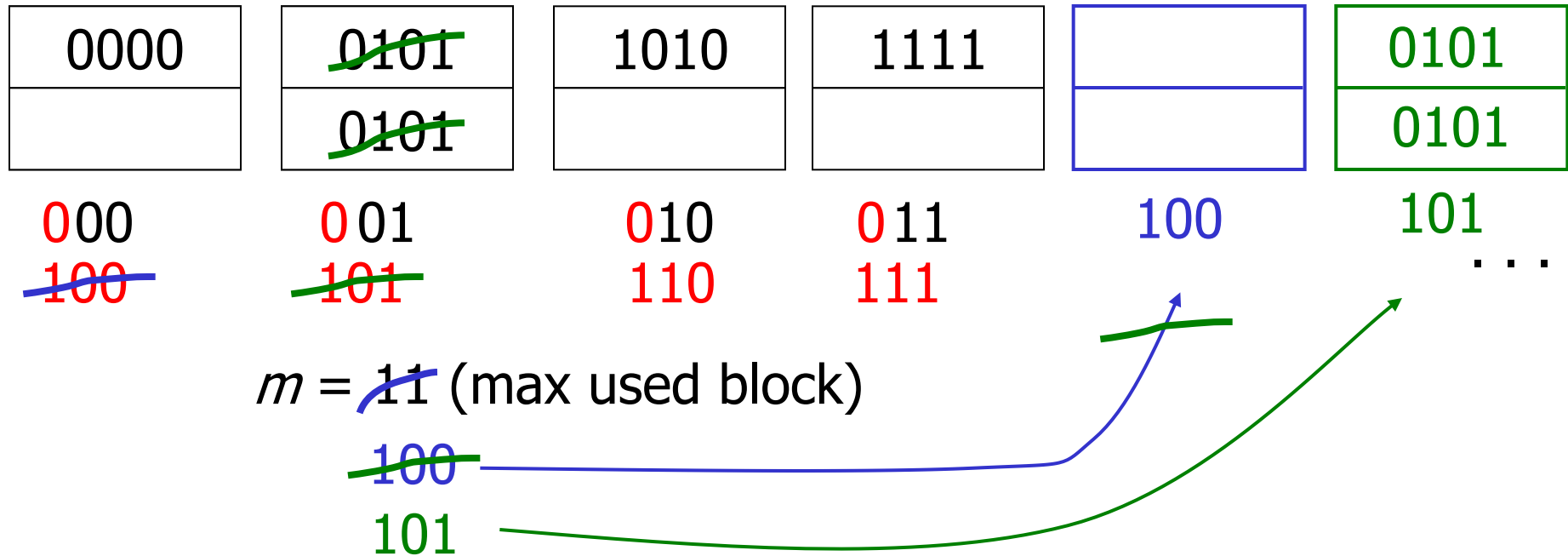
# Example Continued: How to grow beyond this?

$i =$ ~~2~~ 3



# Example Continued: How to grow beyond this?

$i =$ ~~2~~ 3



## When do we expand file?

- Keep track of: 
$$\frac{\text{\# used slots}}{\text{total \# of slots}} = U$$
- If  $U > \text{threshold}$  then increase  $m$   
(and maybe  $i$ )

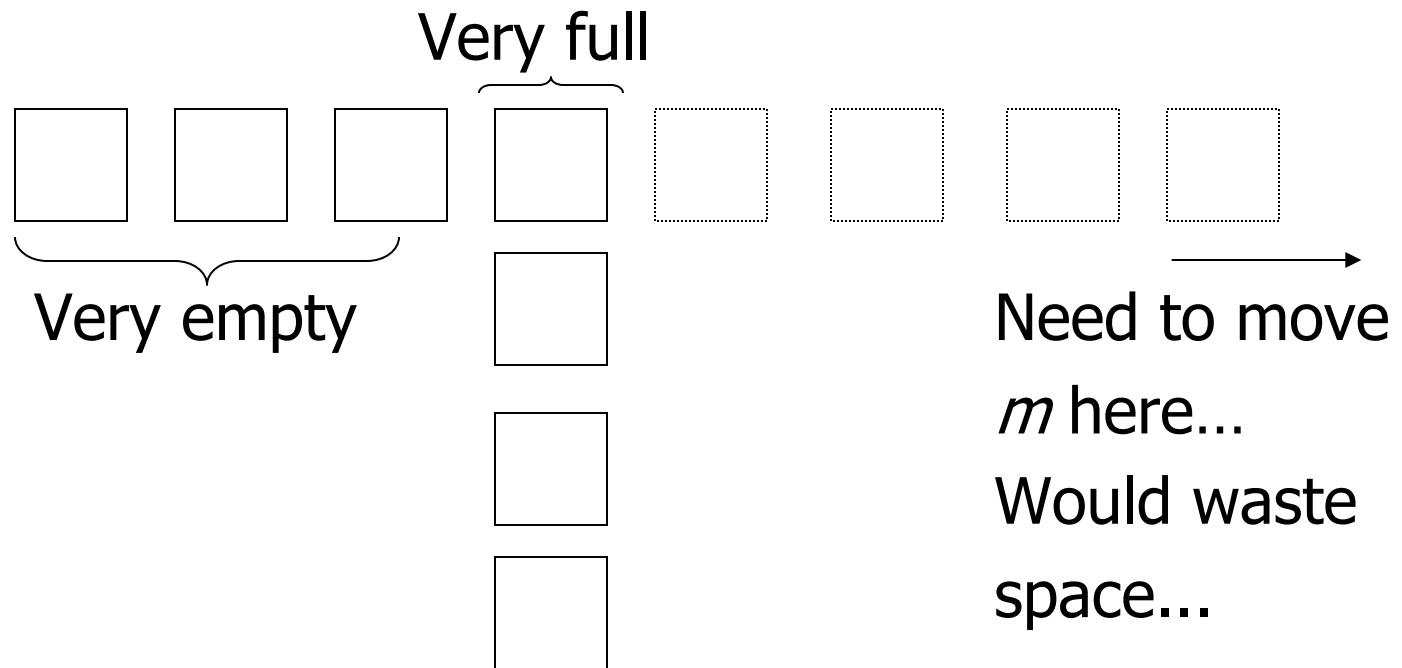


## Summary

## Linear Hashing

- ⊕ Can handle growing files
  - with less wasted space
  - with no full reorganizations
- ⊕ No indirection like extensible hashing
- ⊖ Can still have overflow chains

# Example: BAD CASE



# Summary

## Hashing

- How it works
- Dynamic hashing
  - Extensible
  - Linear

## Next:

- Indexing vs Hashing
- Index definition in SQL
- Multiple key access

# Indexing vs Hashing

- Hashing good for probes given key

e.g.,           SELECT ...  
                  FROM R  
                  WHERE R.A = 5

# Indexing vs Hashing

- **INDEXING** (Including B Trees) good for  
**Range Searches:**

e.g.,

SELECT  
FROM R

WHERE R.A > 5

## Index definition in SQL

- Create index name on rel (attr)
- Create unique index name on rel (attr)

└──────────→ defines candidate key

- Drop INDEX name

Note

CANNOT SPECIFY TYPE OF INDEX

(e.g. B-tree, Hashing, ...)

OR PARAMETERS

(e.g. Load Factor, Size of Hash,...)

... at least in SQL...

In Oracle you can !



Note ATTRIBUTE LIST  $\Rightarrow$  MULTIKEY INDEX  
(next)

e.g., CREATE INDEX foo ON R(**A,B,C**)

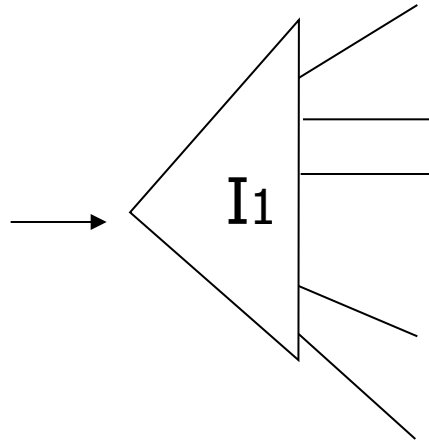
# Multi-key Index

Motivation: Find records where

DEPT = "Toy" AND SAL > 50k

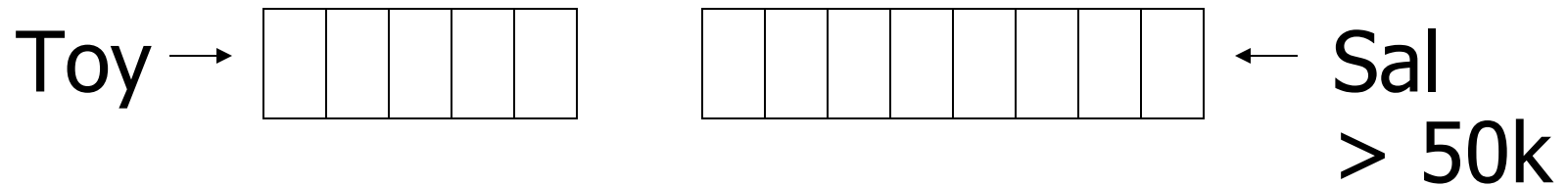
## Strategy I:

- Use one index, say Dept.
- Get all Dept = "Toy" records and check their salary



## Strategy II:

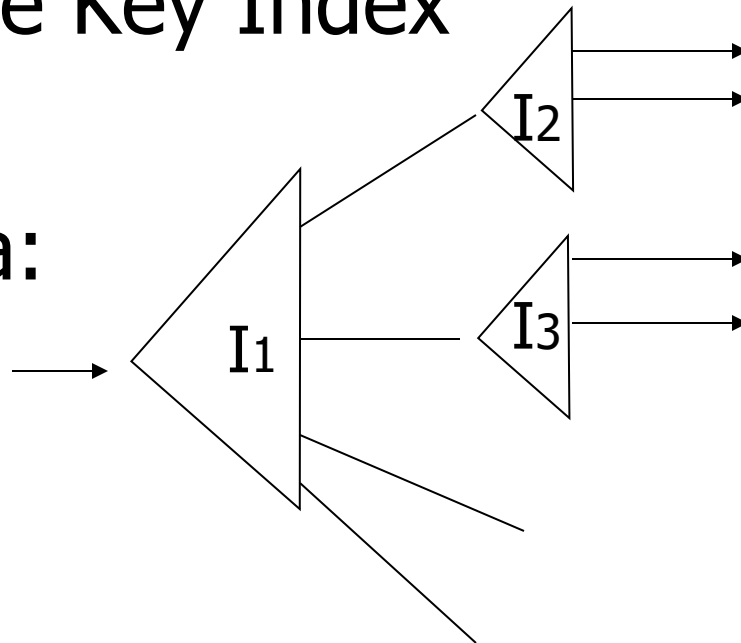
- Use 2 Indexes; Manipulate Pointers



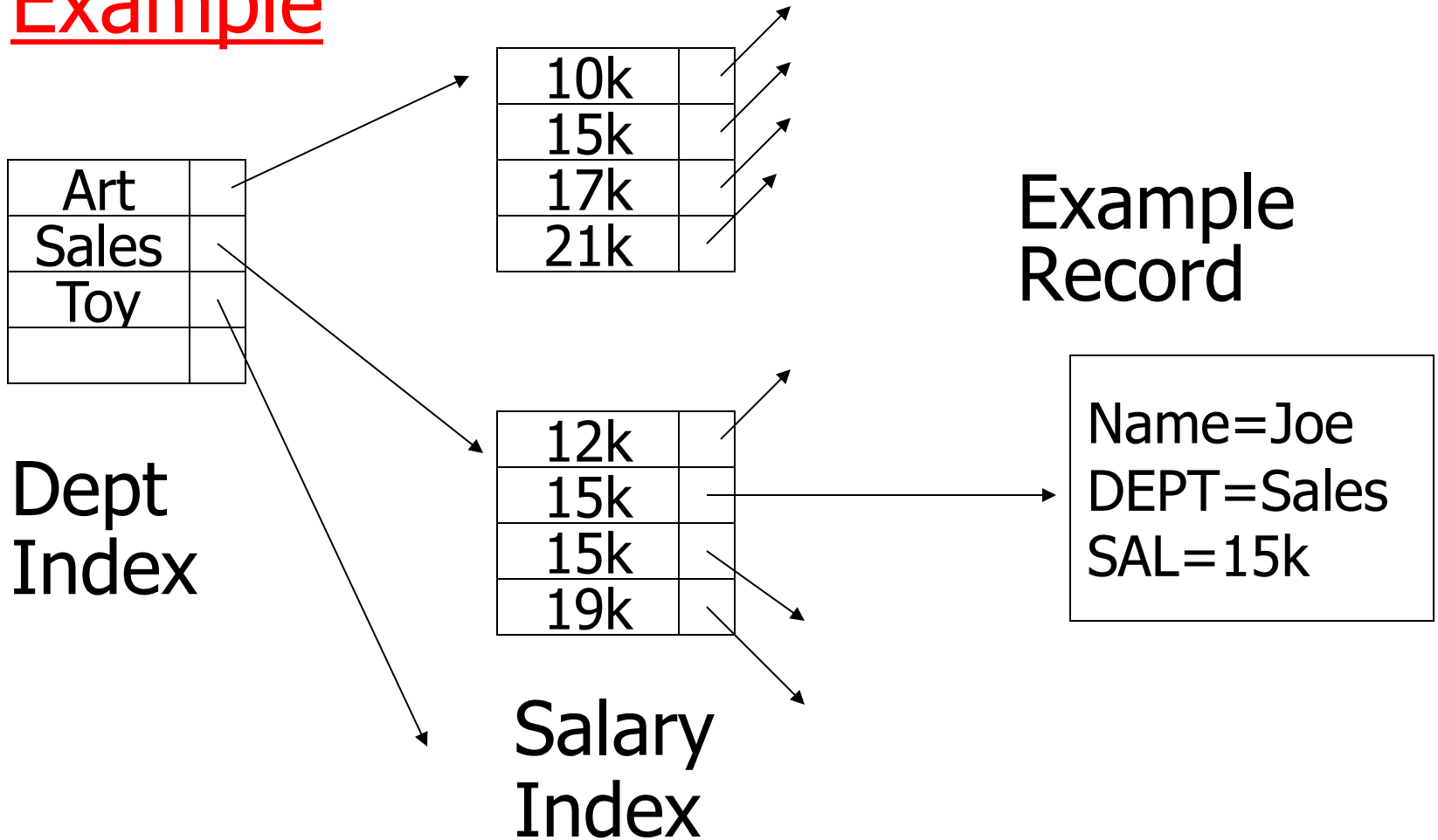
## Strategy III:

- Multiple Key Index

One idea:



# Example



# For which queries is this index good?

- ☐ Find RECs Dept = "Sales"  $\wedge$  SAL=20k
- ☐ Find RECs Dept = "Sales"  $\wedge$  SAL  $\geq$  20k
- ☐ Find RECs Dept = "Sales"
- ☐ Find RECs SAL = 20k