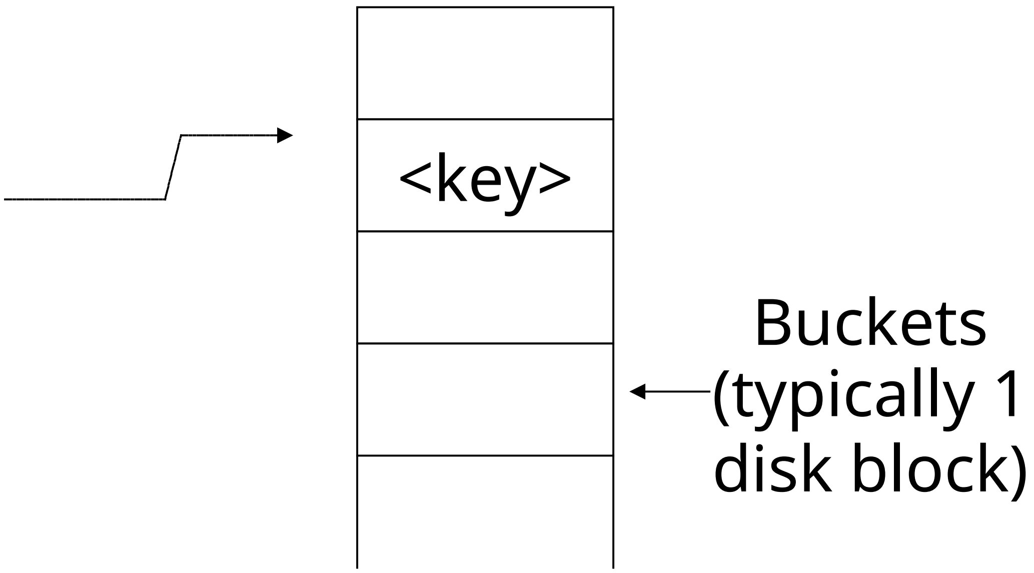
Ullman et al. :
Database System Principles

Notes 5: Hashing and More

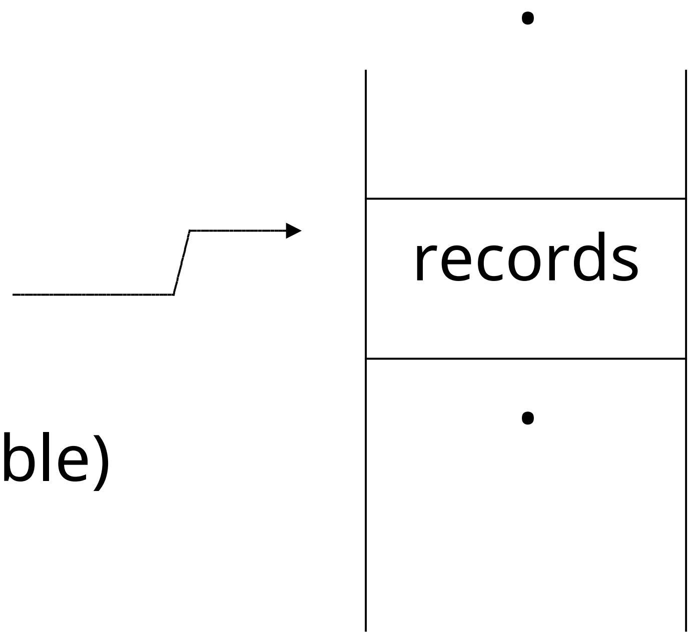
Hashing

key → h(key)



Two alternatives

(1) key → h(key)

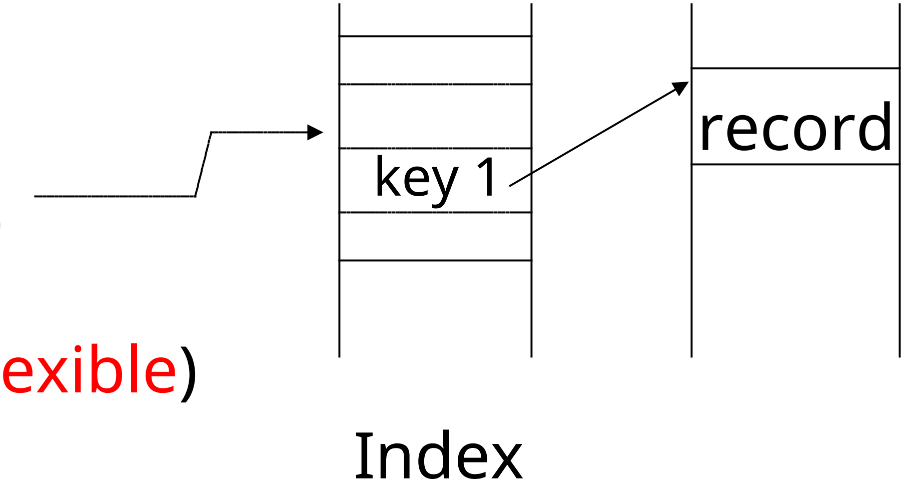


(direct reference, not flexible)

Two alternatives

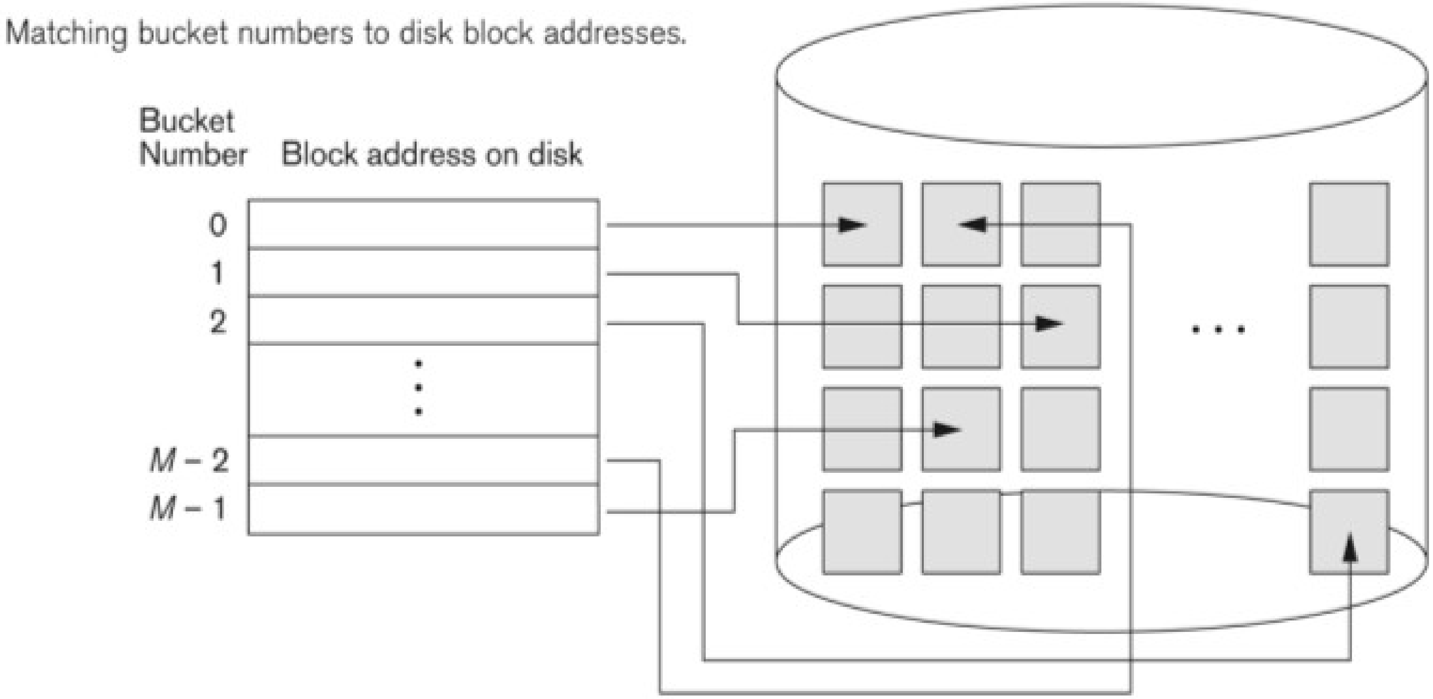
(2) key → h(key)

(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 … x_n’ *n* byte character string
- Have *b* buckets
- h: add $x_1 + x_2 + \ldots\ldots x_n$
  - 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:

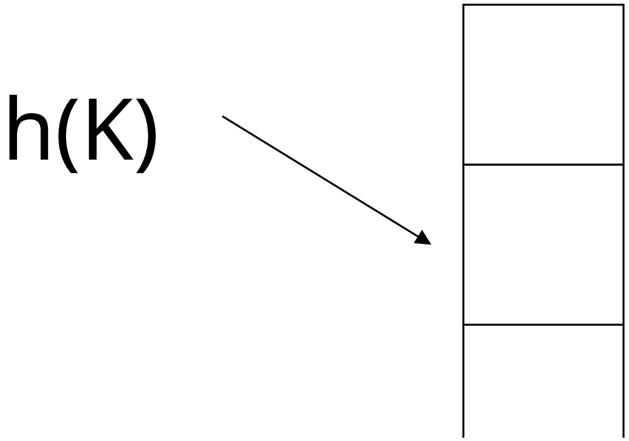
☞ Expected number of keys/bucket is the same for all buckets

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

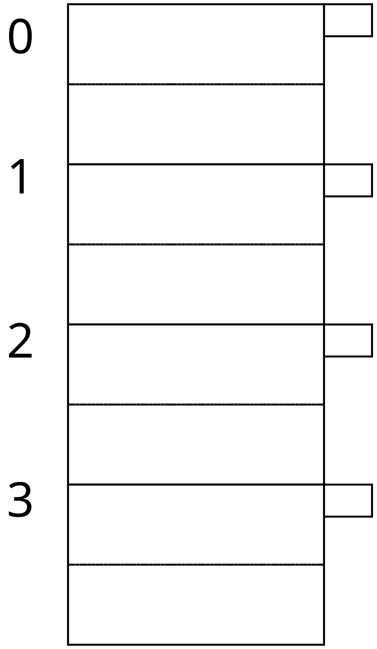
INSERT:

h(a) = 1

h(b) = 2

h(c) = 1

h(d) = 0



EXAMPLE 2 records/bucket

INSERT:

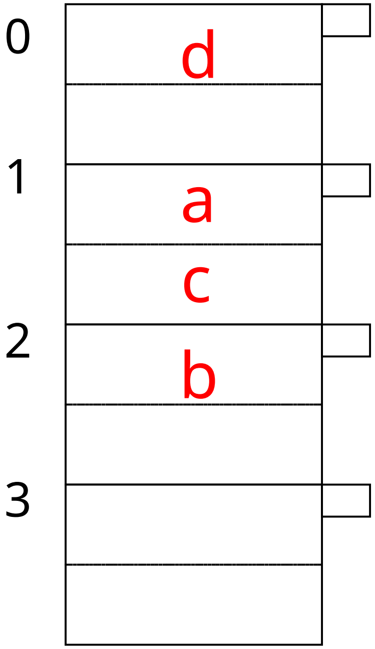
h(a) = 1

h(b) = 2

h(c) = 1

h(d) = 0

h(e) = 1



EXAMPLE 2 records/bucket

INSERT:

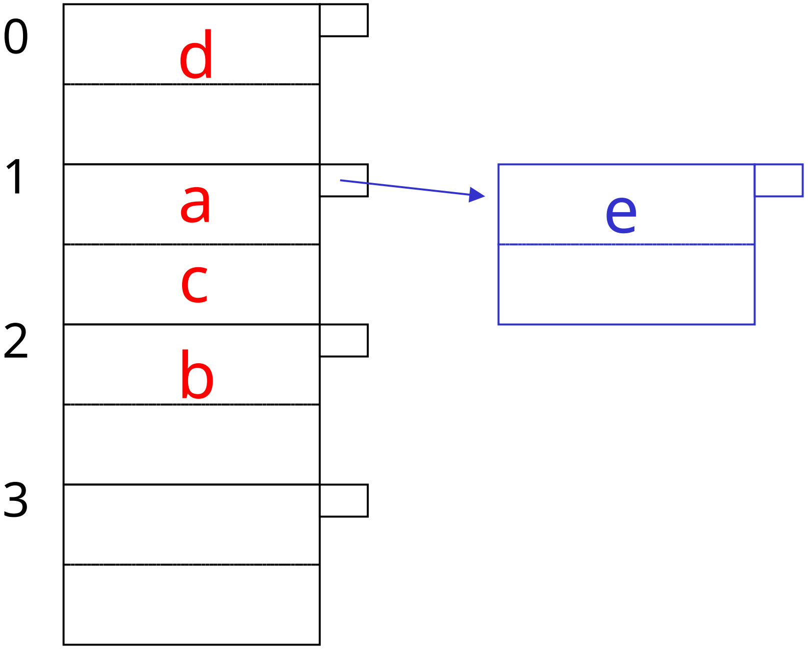
h(a) = 1

h(b) = 2

h(c) = 1

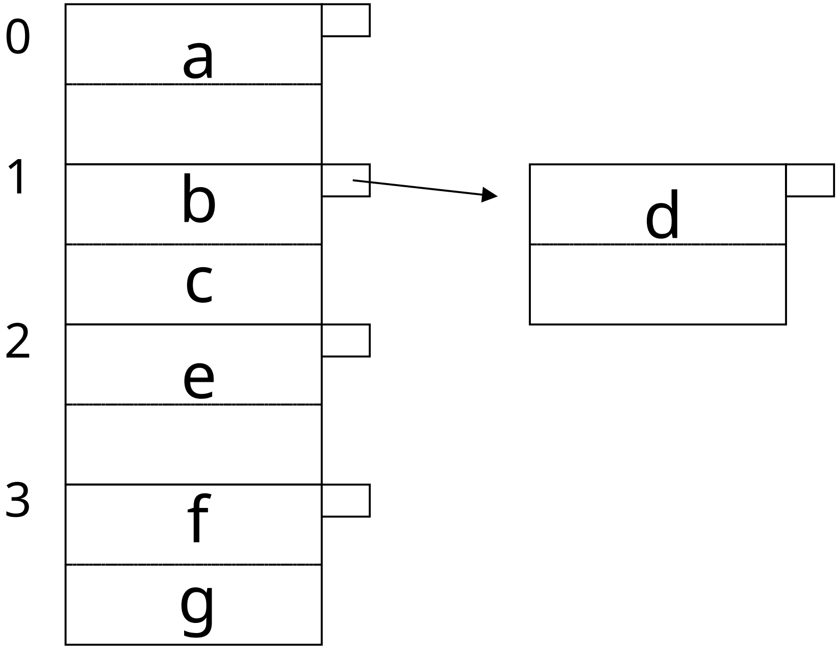
h(d) = 0

h(e) = 1



EXAMPLE: deletion

Delete:
- e
- f



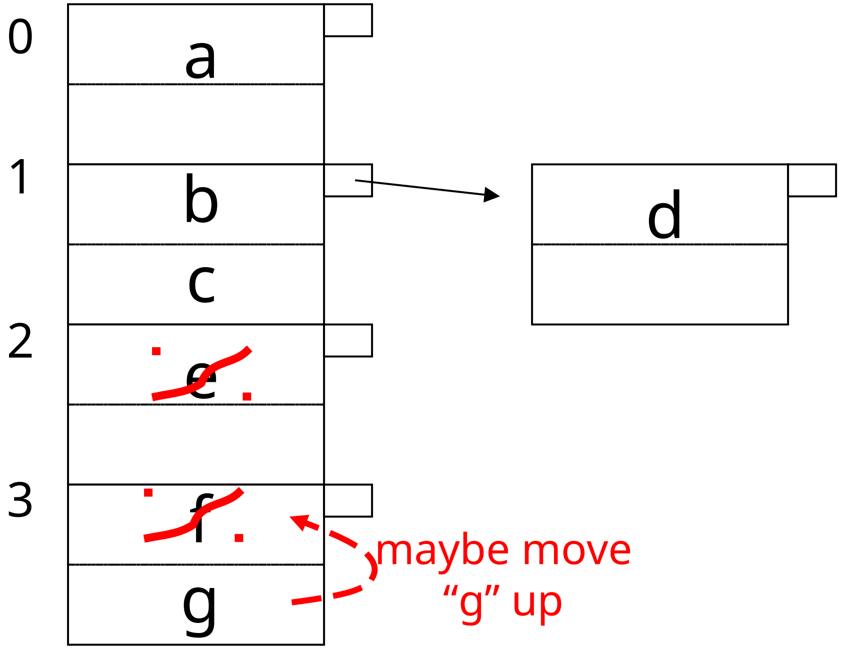
EXAMPLE: deletion

Delete:

e

f

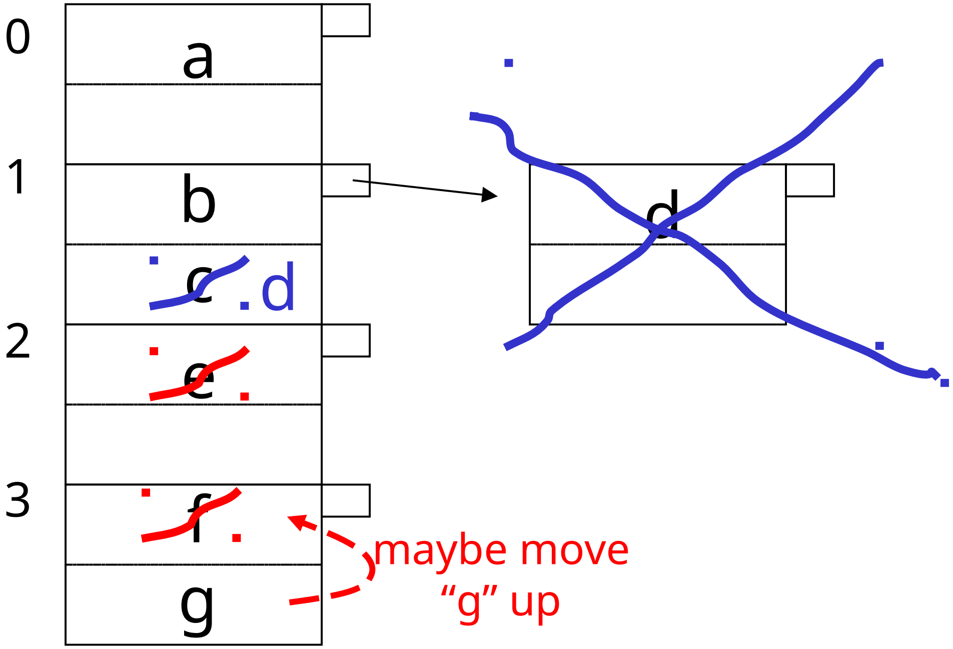
c



EXAMPLE: deletion

Delete:

- e
- f
- c



Rule of thumb:

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

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

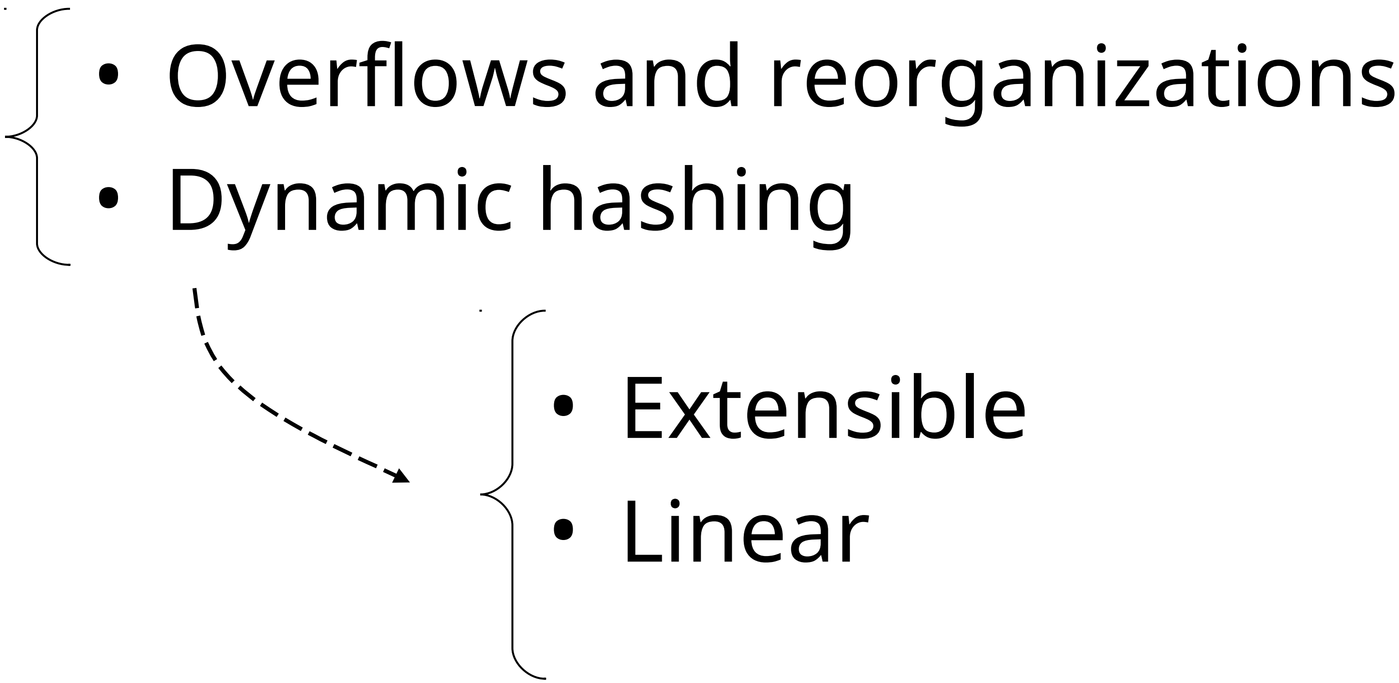
- If < 50%, wasting space
- If > 80%, overflows significant

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?

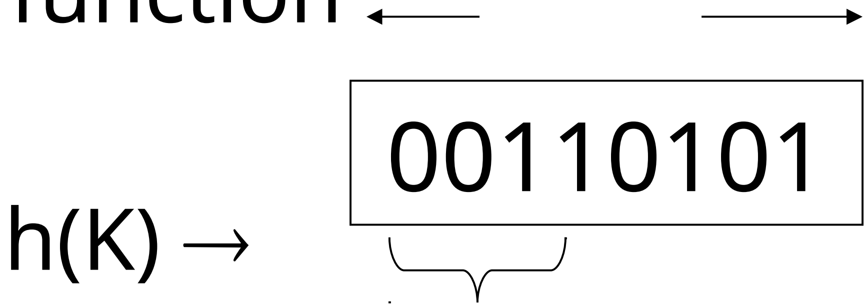


- Overflows and reorganizations
- Dynamic hashing

- Extensible
- Linear

Extensible hashing: two ideas

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

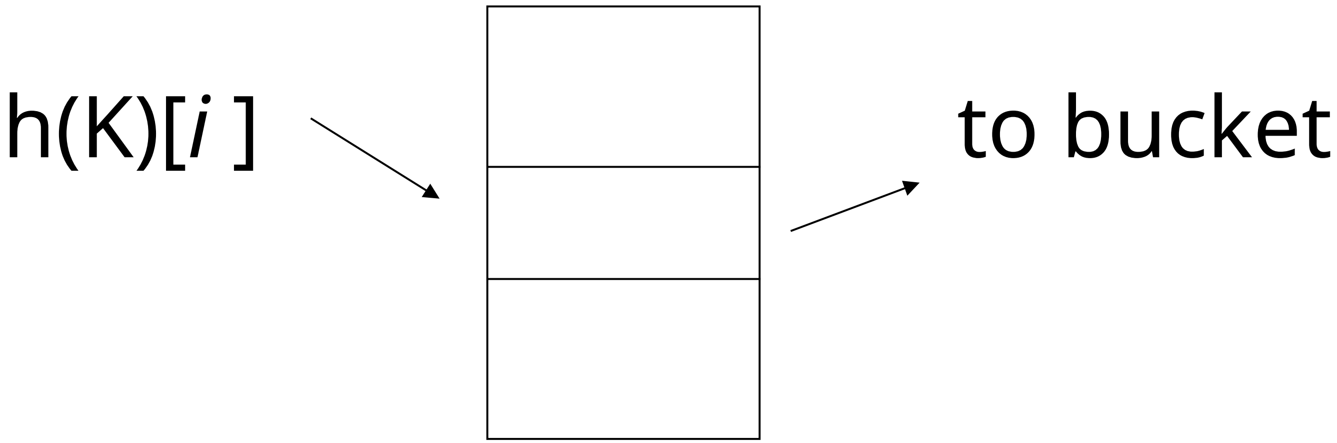


h(K) →

00110101

use *i* → grows over time….

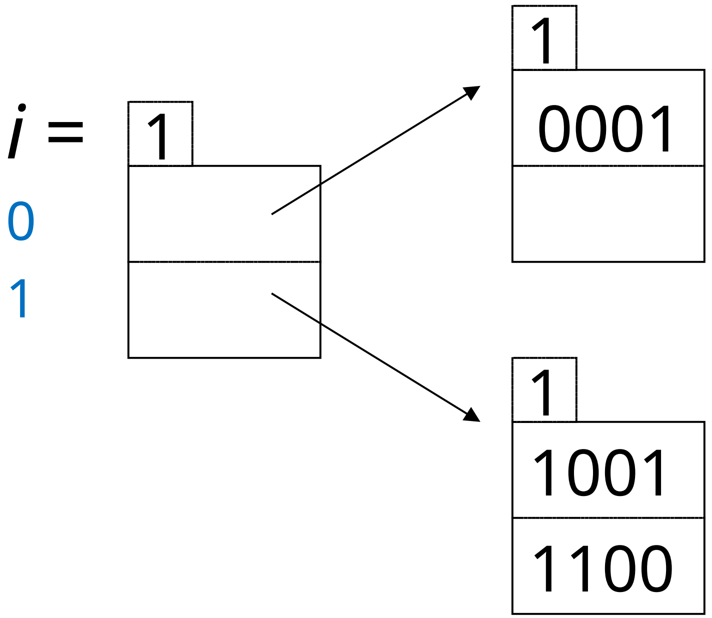
(b) Use directory



h(K)[*i*]

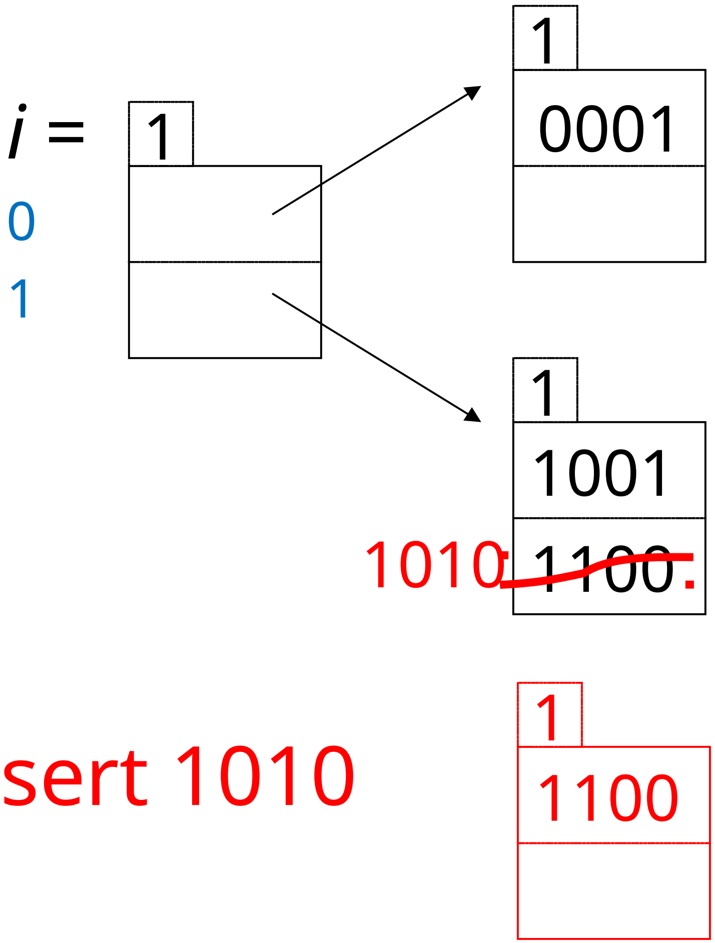
to bucket

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



Insert 1010

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



i = 1

0

1

1

0001

1

1001

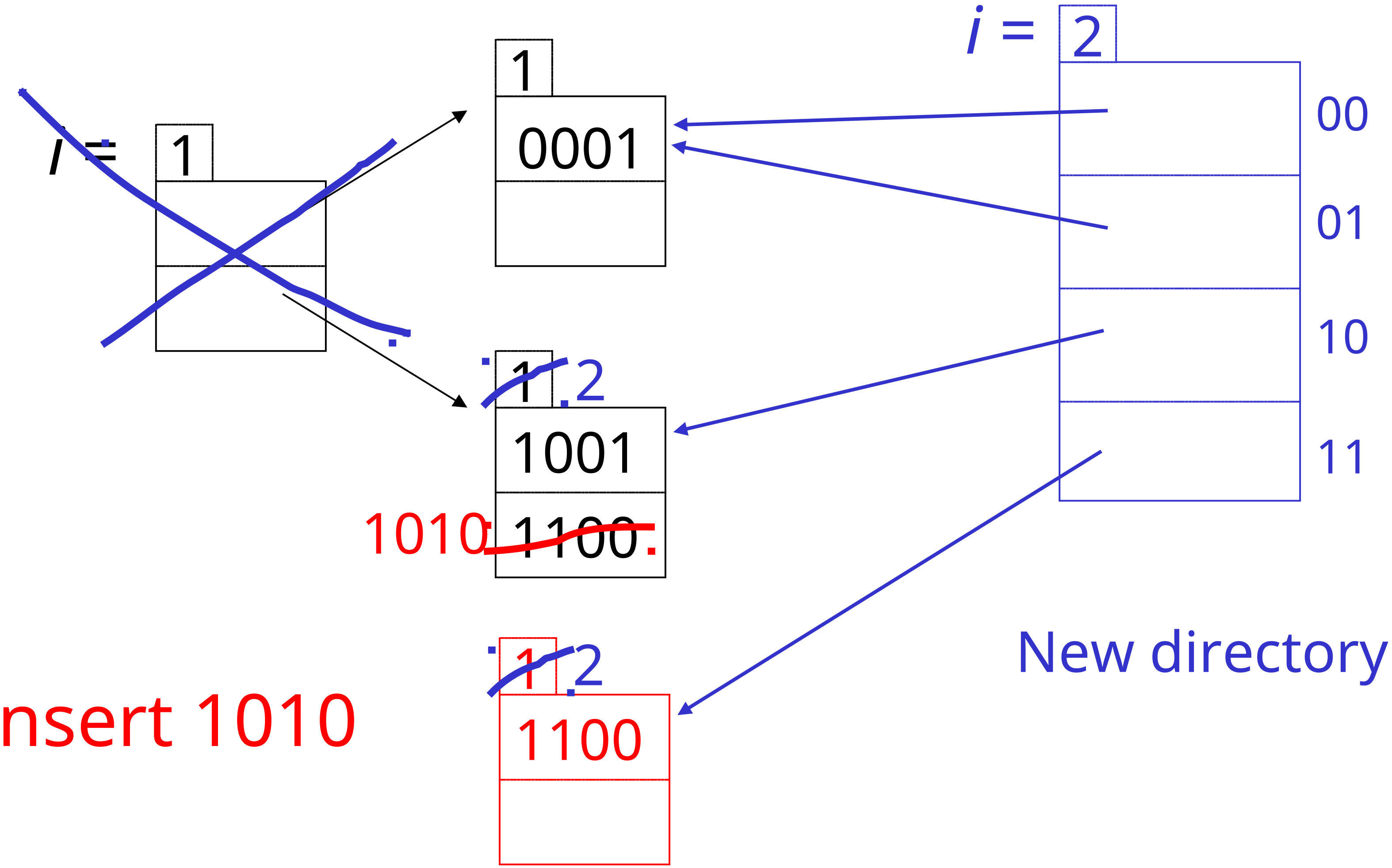
1010 ~~1100~~

1

1100

Insert 1010

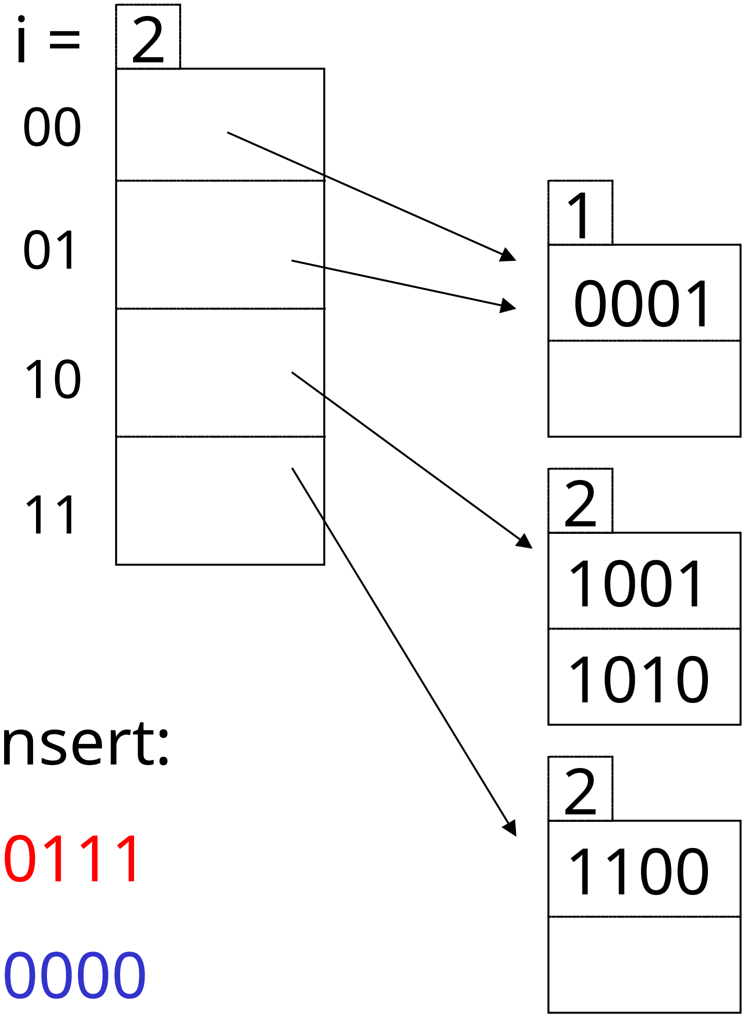
Example: h(k) is 4 bits; 2 keys/bucket



New directory

Insert 1010

Example continued

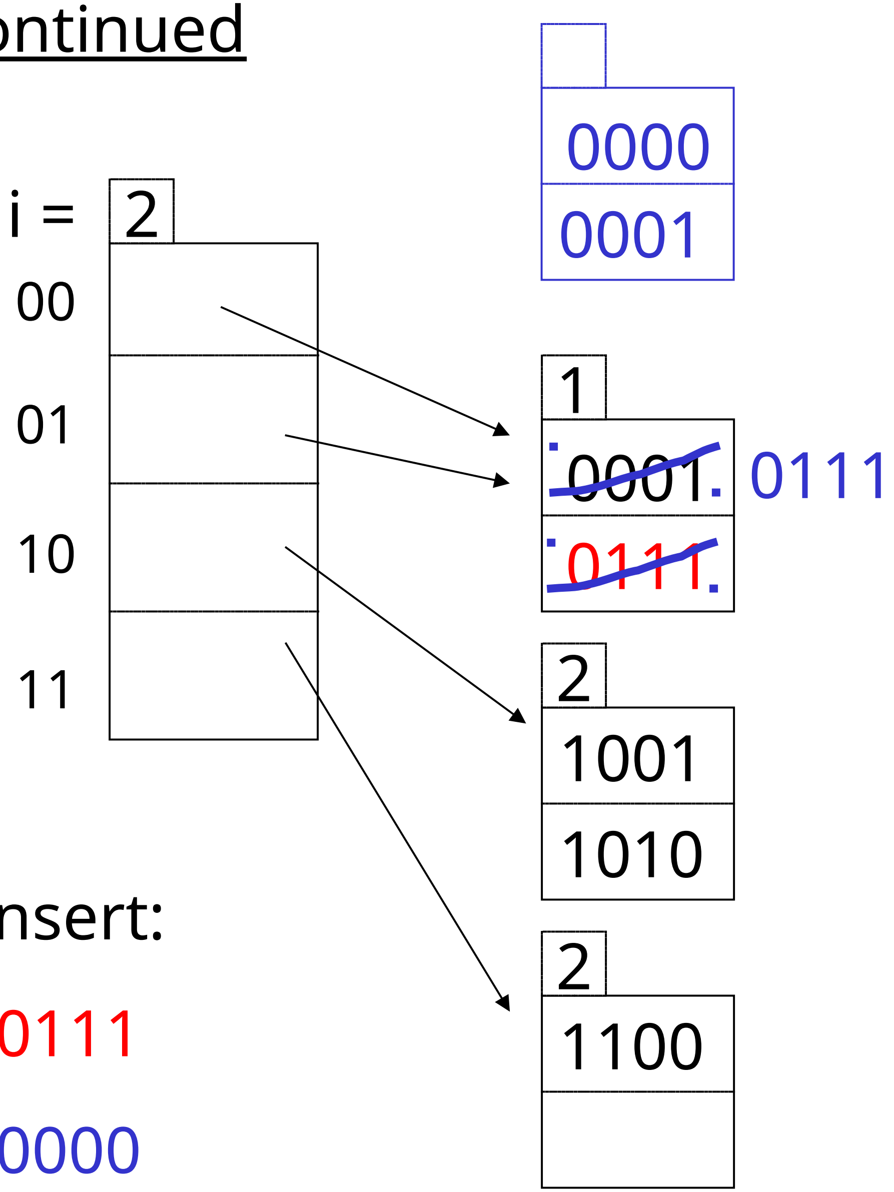


Insert:

0111

0000

Example continued



i = 2

00

01

10

11

0000

0001

1

~~0001~~

~~0111~~

0111

2

1001

1010

2

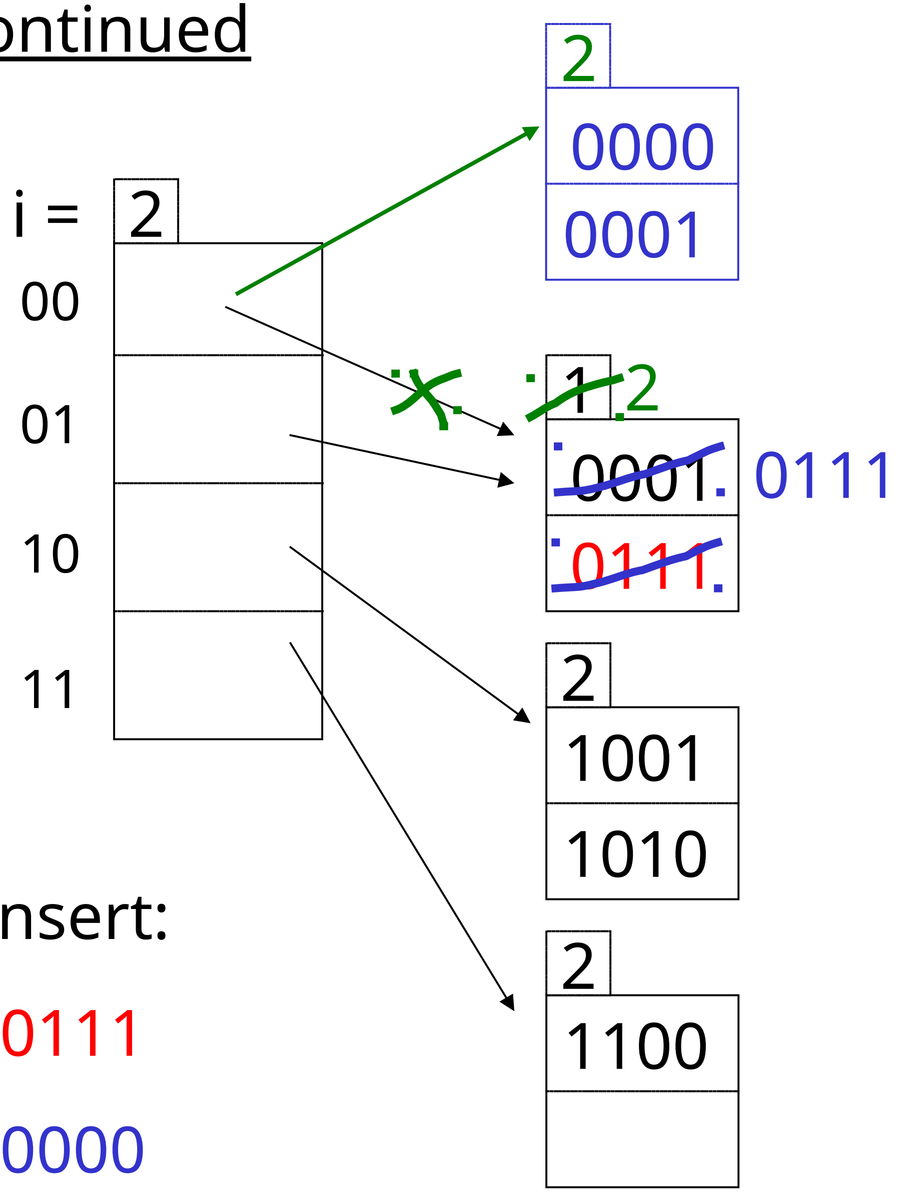
1100

Insert:

0111

0000

Example continued

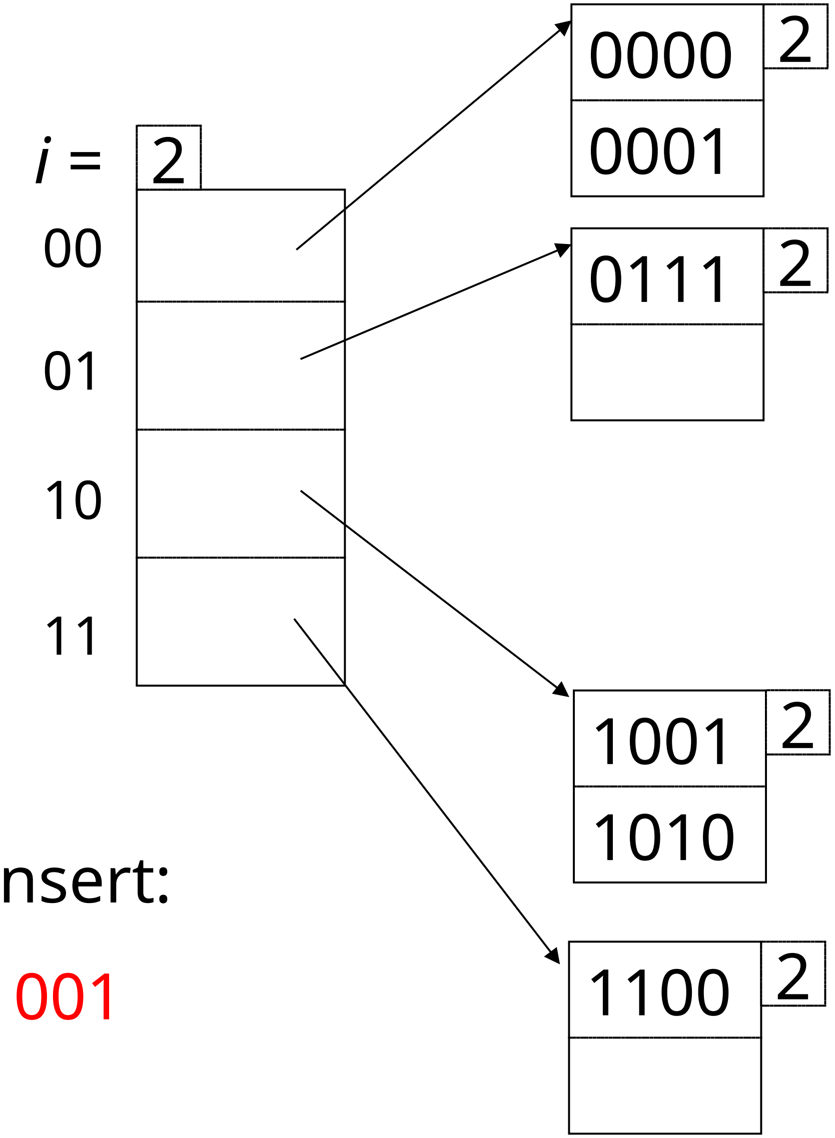


Insert:

0111

0000

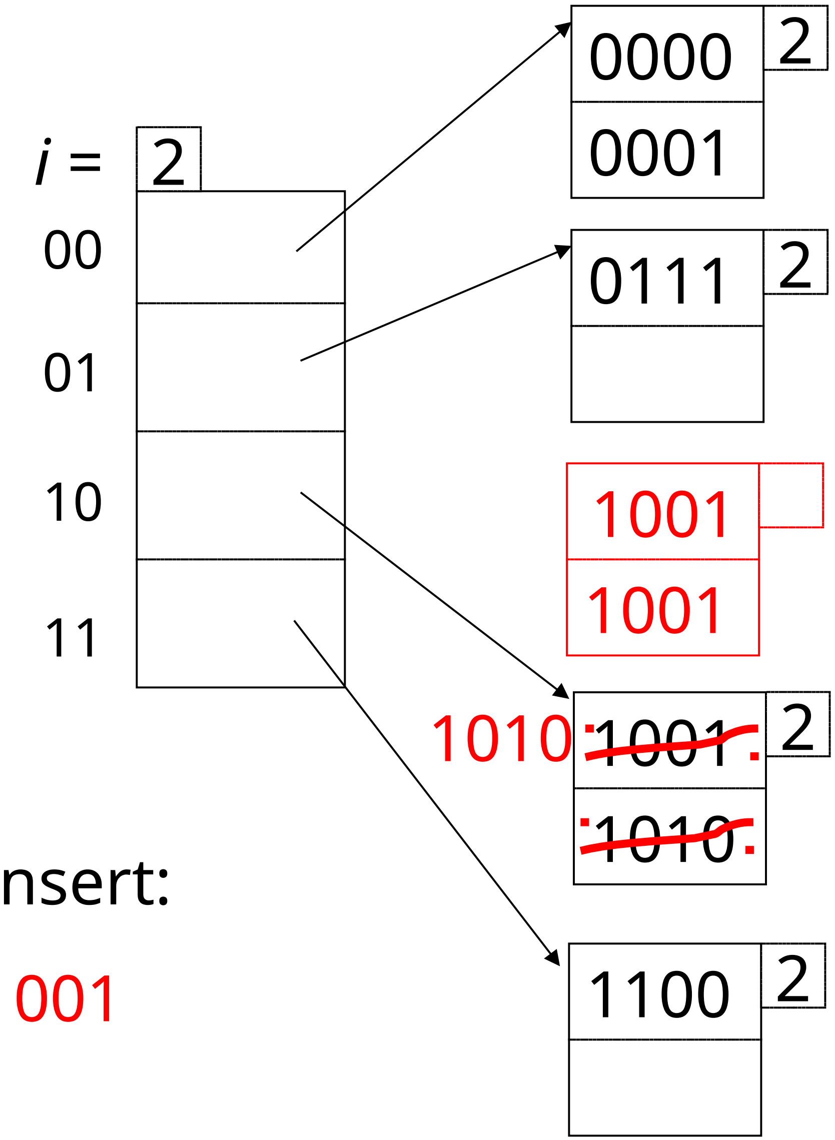
Example continued



Insert:

1001

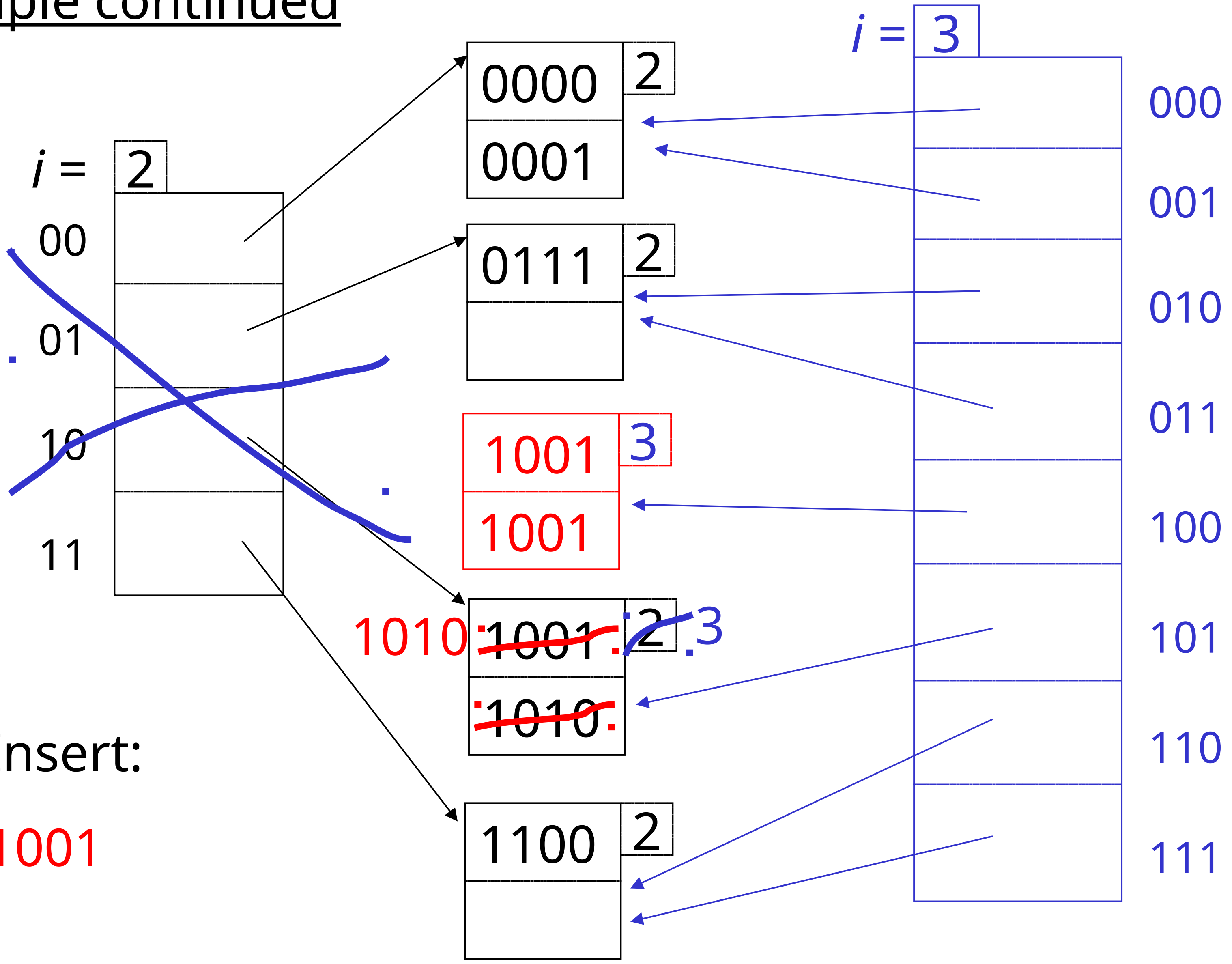
Example continued



Insert:

1001

Example continued



Insert:

1001

Extensible hashing: deletion

- No merging of blocks
- Merge blocks
  and cut directory if possible

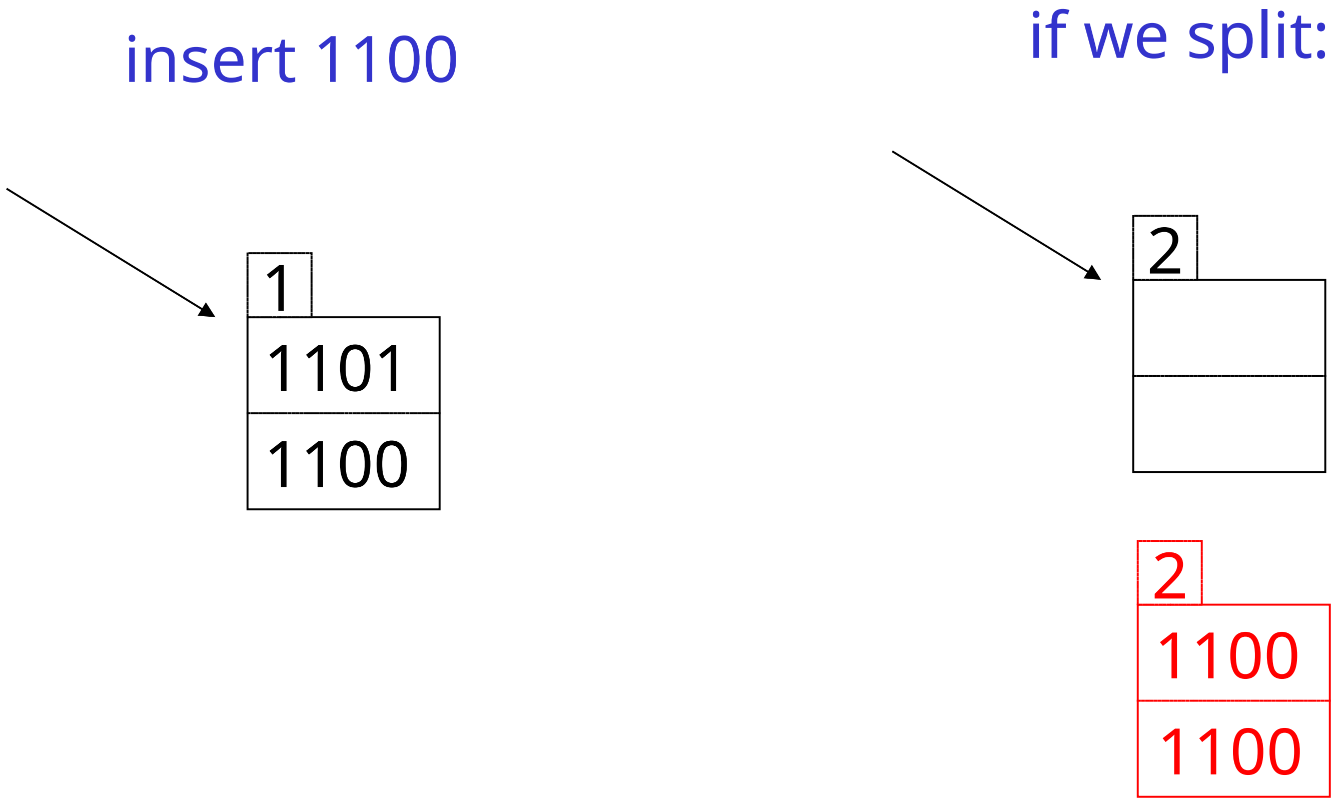
  (Reverse insert procedure)

Deletion example:

- Run thru insert example in reverse!

Note: Still need overflow chains

- Example: many records with duplicate keys



insert 1100

1

1101

1100

if we split:

2

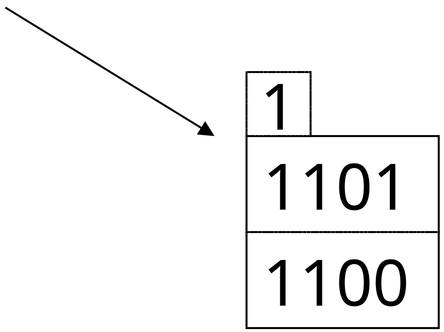
2

1100

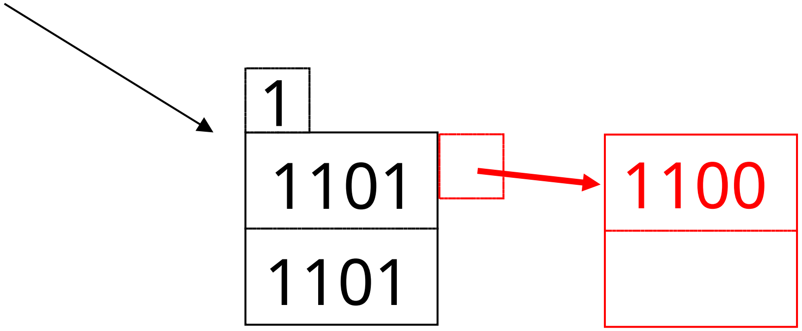
1100

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

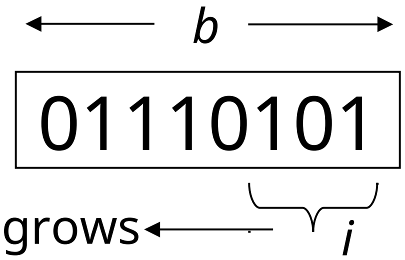
(Now it fits, now it does not)

Linear hashing

- Another dynamic hashing scheme

Two ideas:

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



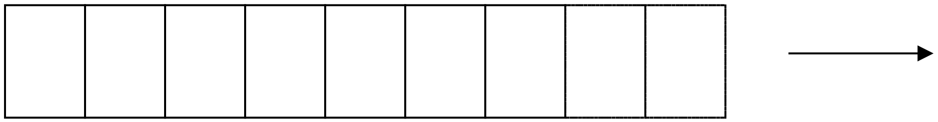
b

01110101

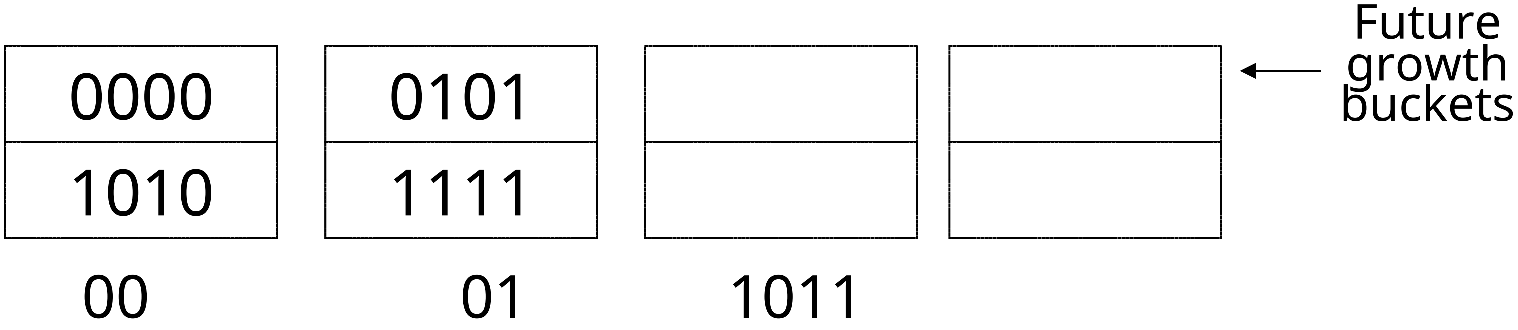
grows

i

(b) File grows linearly



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



0000

1010

0101

1111

Future growth buckets

00

01

1011

m = 01 (max used block)

Rule

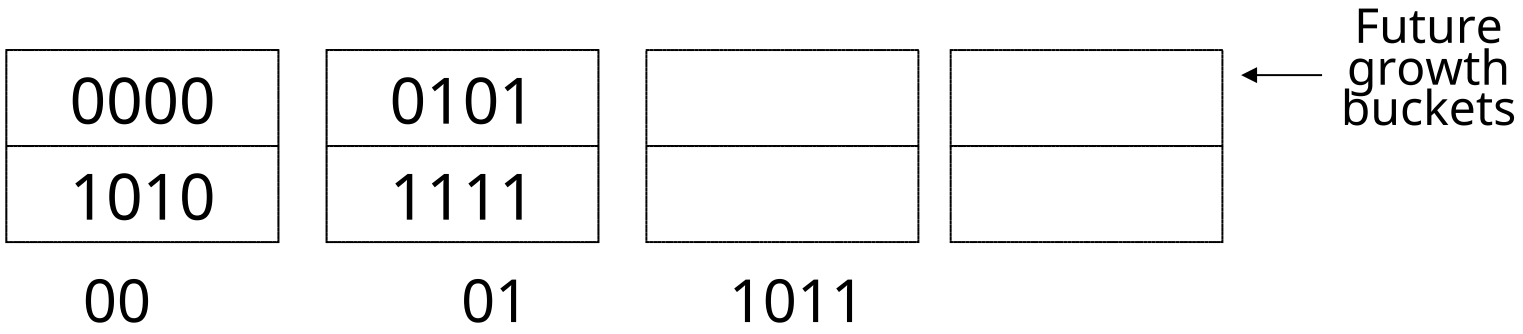
If h(k)[i] $\leq m$, then

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



m = 01 (max used block)

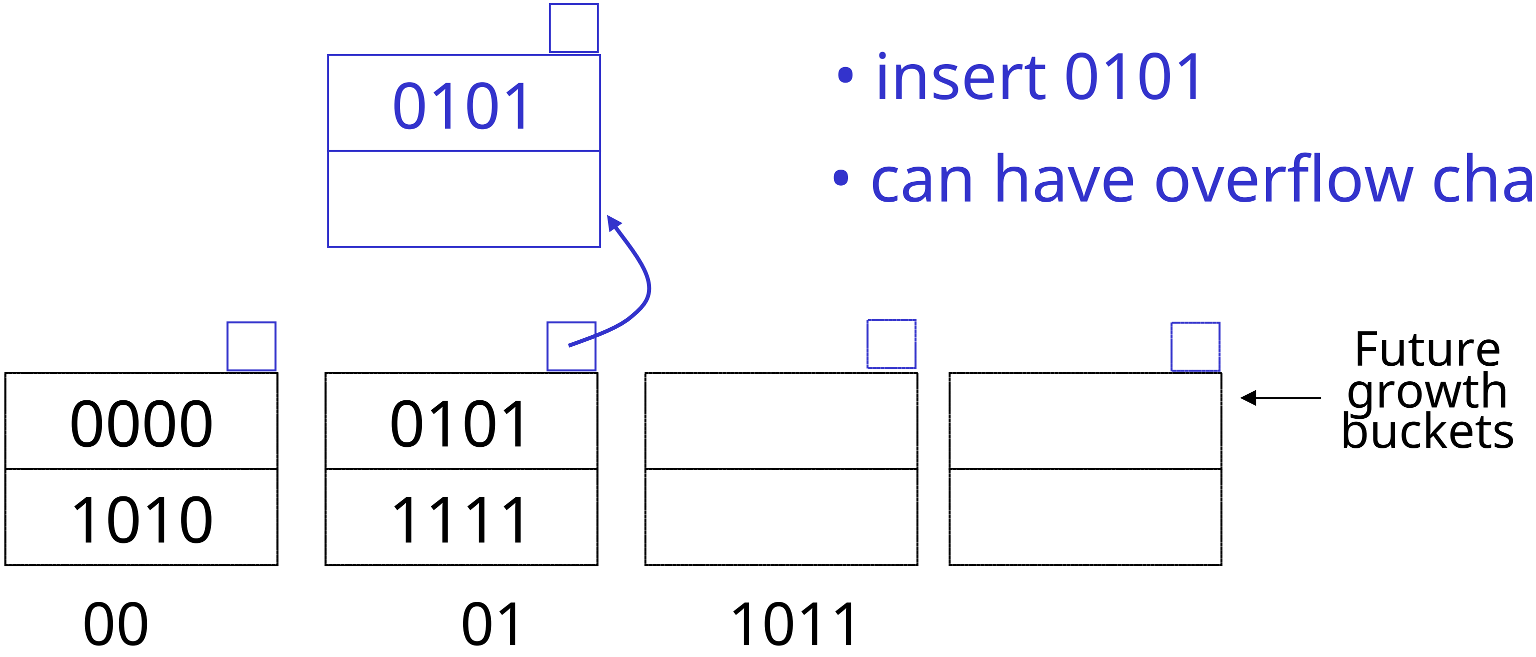
Rule If h(k)[i] ≤ m, then

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
- can have overflow chains!

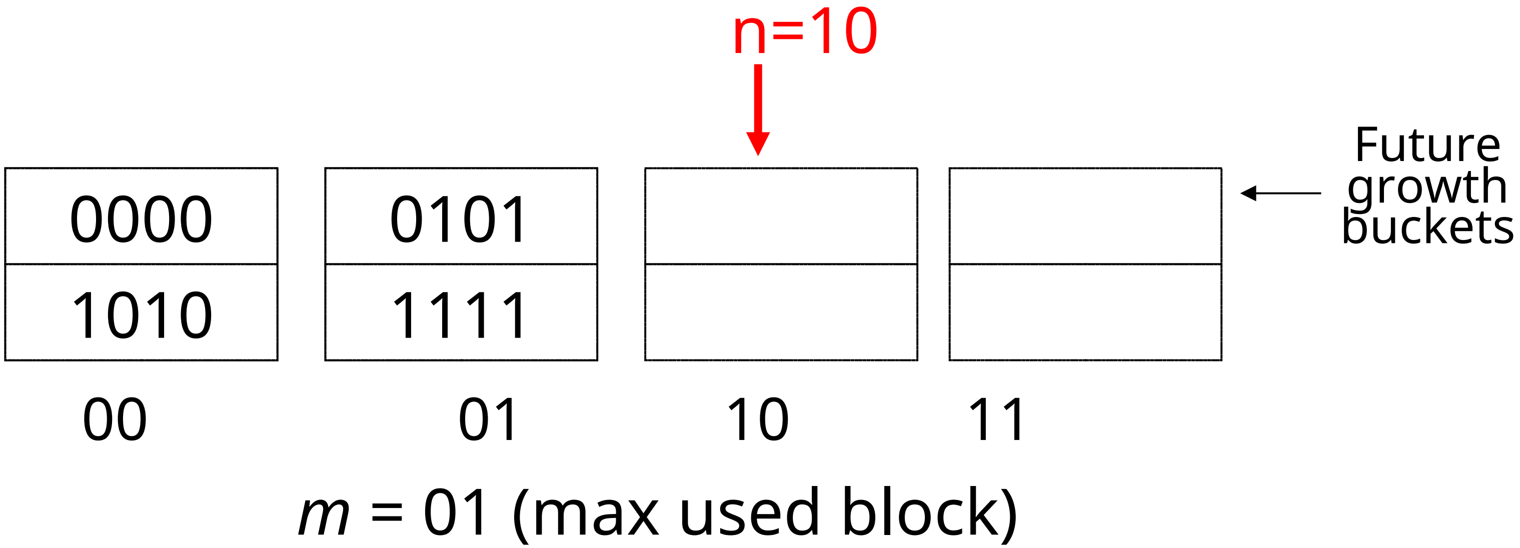


m = 01 (max used block)

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

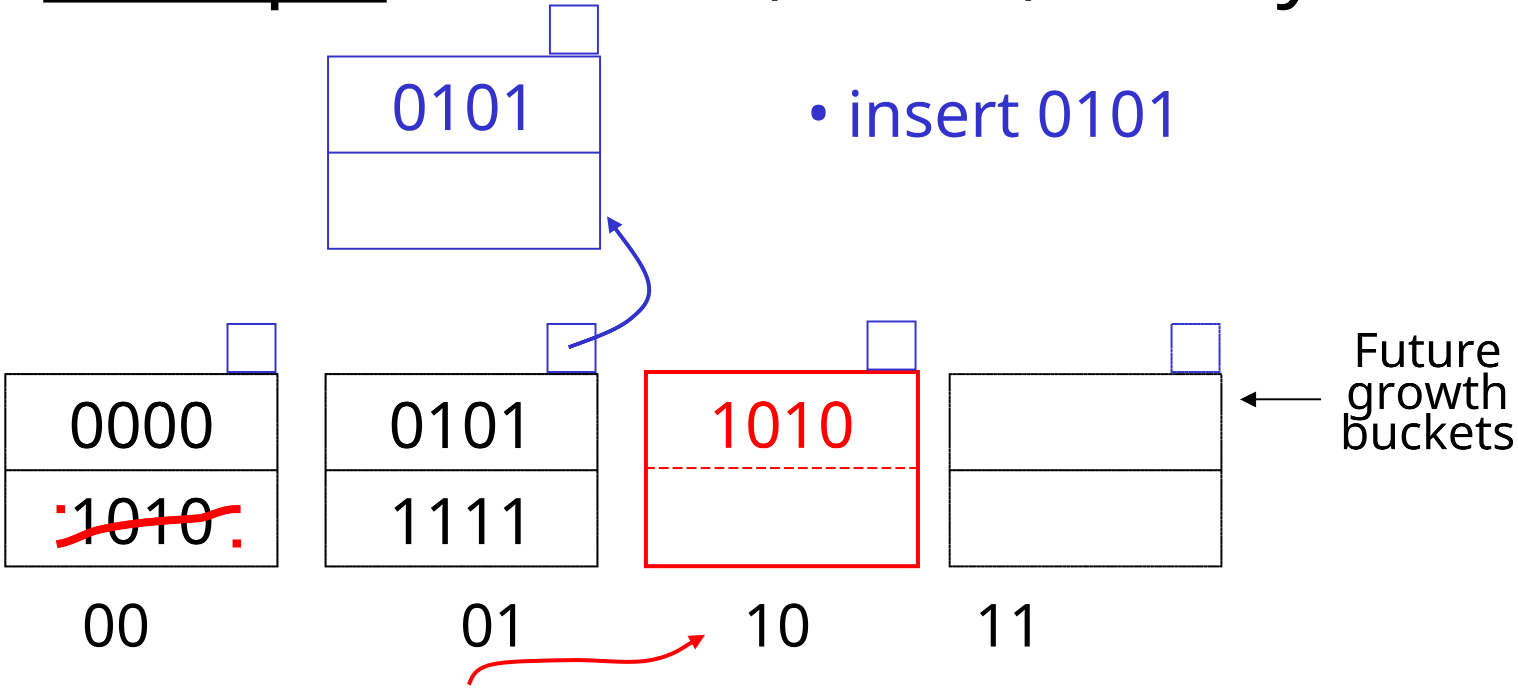
Note

- In textbook, n is used instead of m
- n=m+1



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

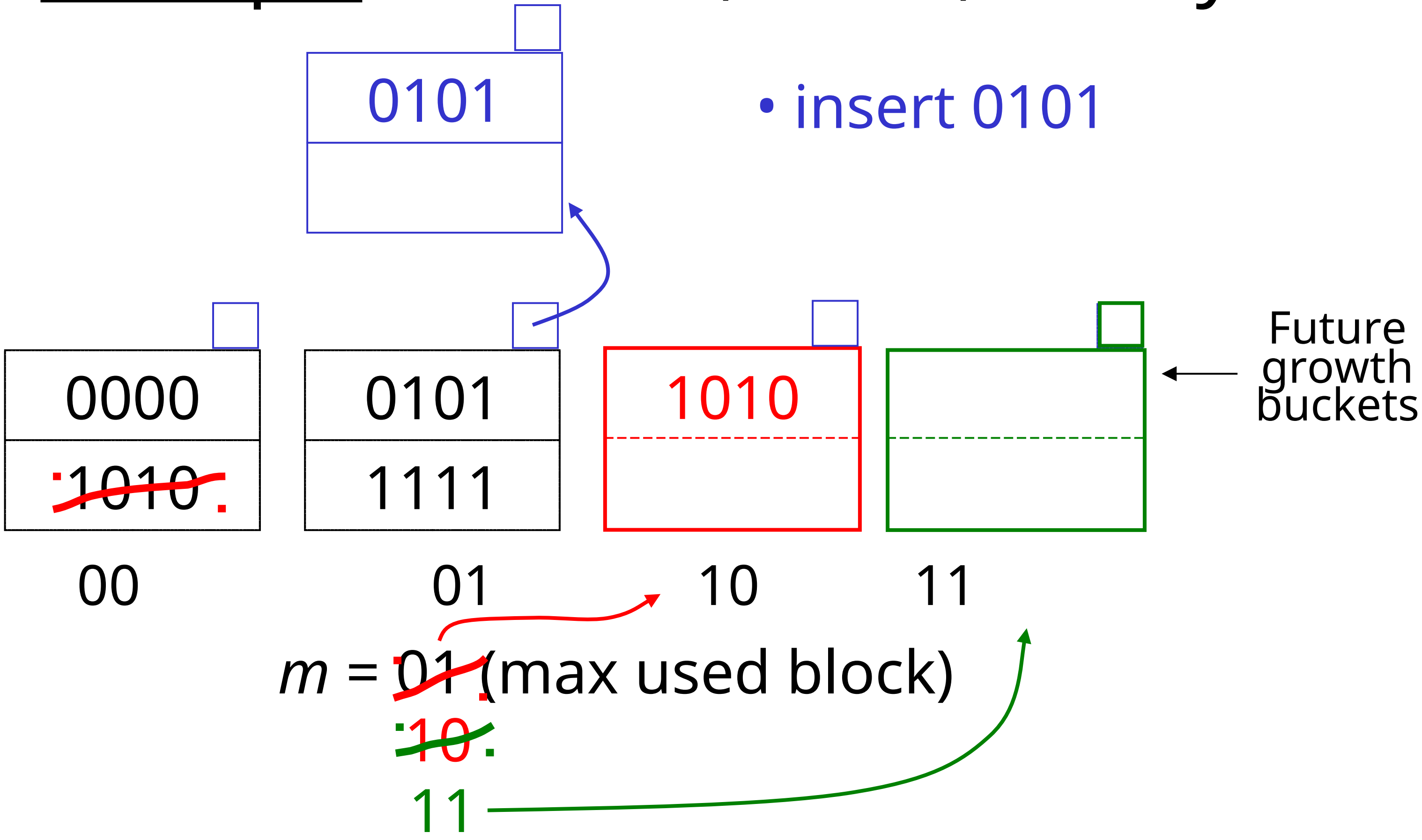
- insert 0101



m = ~~01~~ (max used block)
10

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

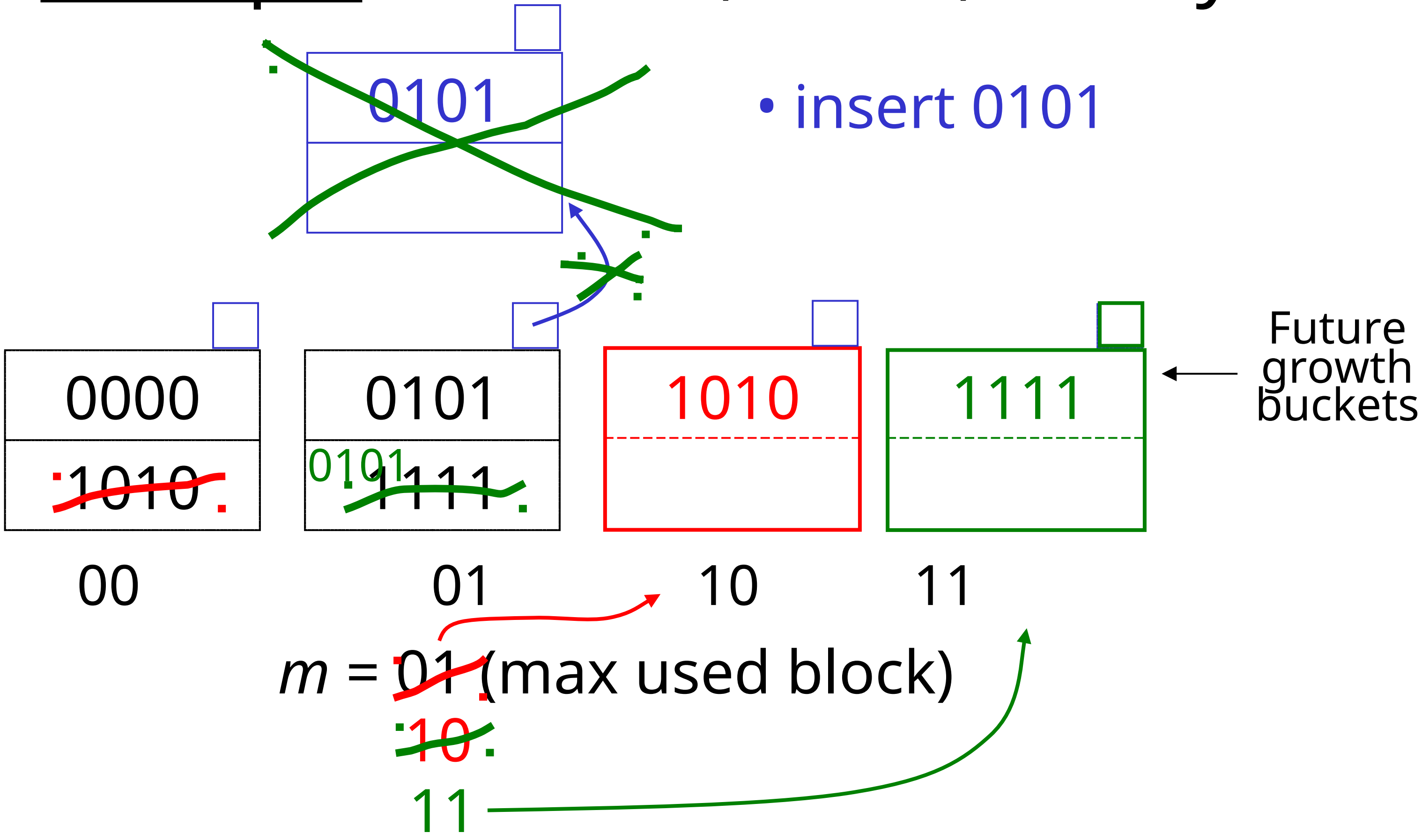
- insert 0101



m = ~~01~~ (max used block)
~~10~~
11

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

- insert 0101



~~0101~~

0000

~~1010~~

0101

0101 ~~1111~~

1010

1111

00 01 10 11

Future growth buckets

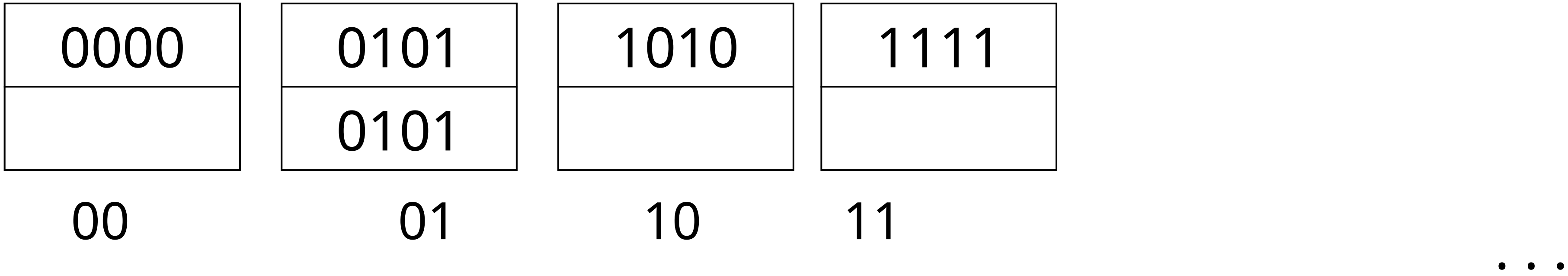
m = ~~01~~ (max used block)

~~10~~

11

Example Continued: How to grow beyond this?

i = 2



0000

0101
0101

1010

1111

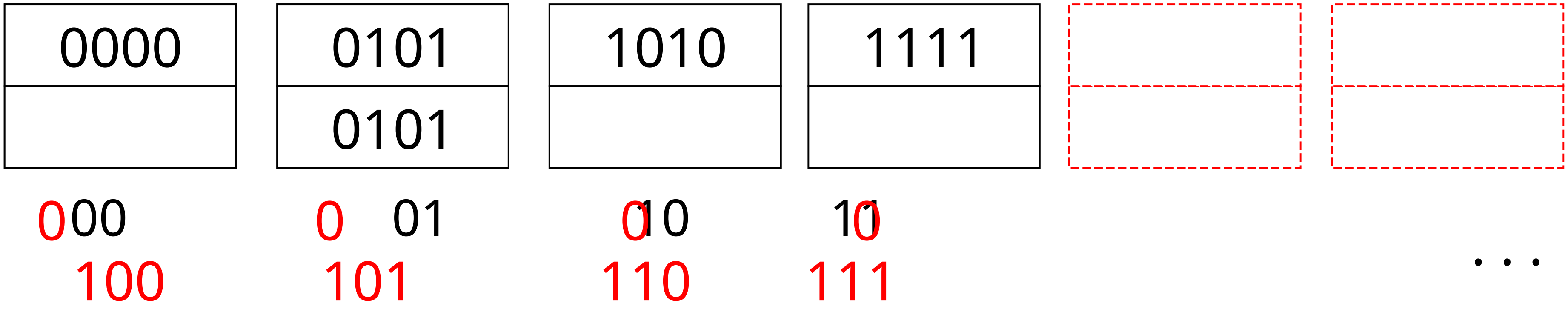
00 01 10 11

. . .

m = 11 (max used block)

Example Continued: How to grow beyond this?

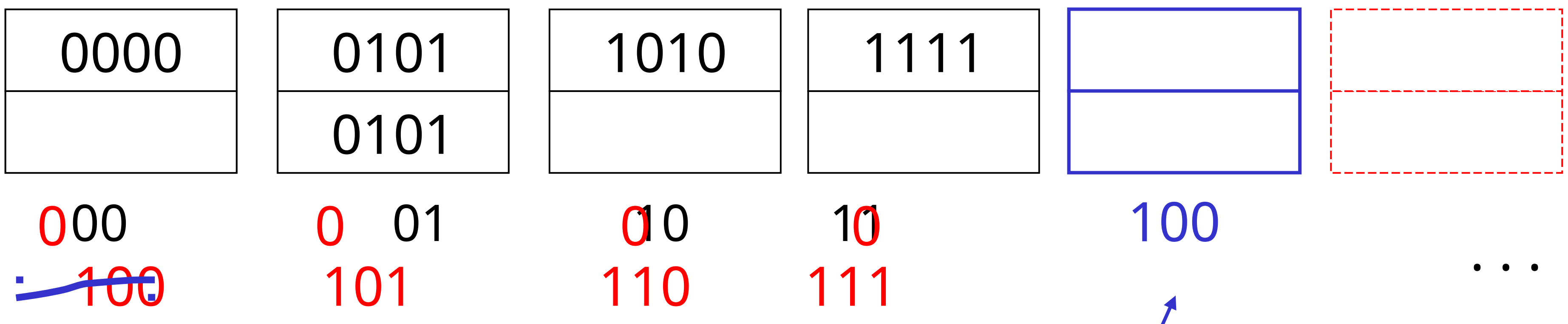
i = ~~2~~ 3



m = 11 (max used block)

Example Continued: How to grow beyond this?

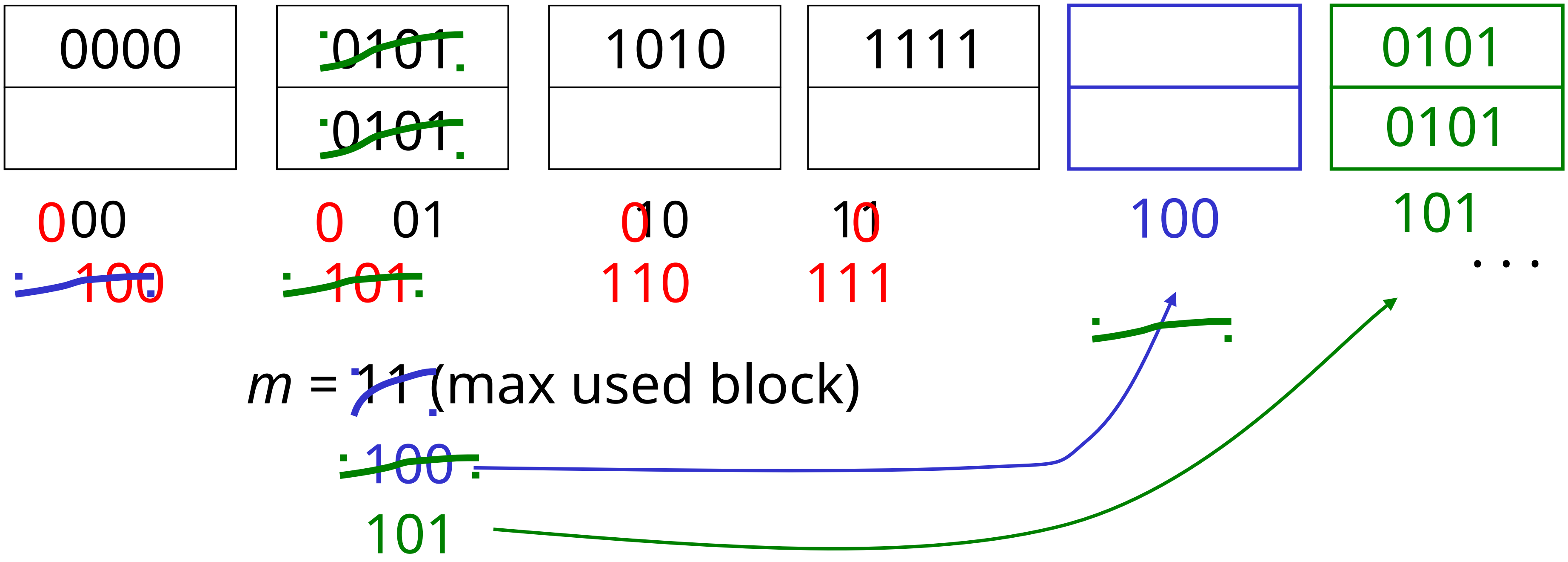
i = ~~2~~ 3



m = ~~11~~ (max used block)
100

Example Continued: How to grow beyond this?

i = ~~2~~ 3



0000

~~0101~~
~~0101~~

1010

1111

0101
0101

000
~~100~~

0 01
~~101~~

010
110

110
111

100

101

. . .

m = ~~11~~ (max used block)

~~100~~

101

☛ When do we expand file?

- Keep track of: $\dfrac{\text{\# used slots}}{\text{total \# of slots}} = U$

- If U > threshold then increase *m*

  (and maybe *i*)

Summary   Linear Hashing

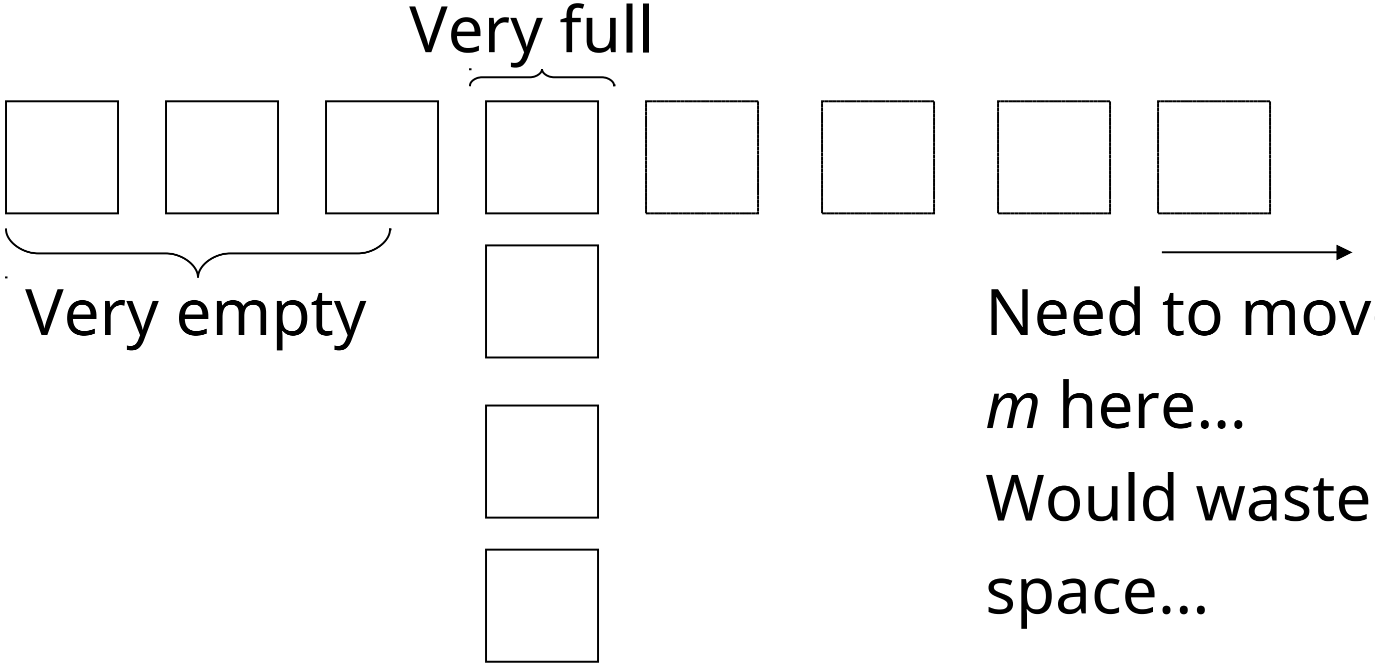
⊕ Can handle growing files

- with less wasted space
- with no full reorganizations

⊕ No indirection like extensible hashing

⊖ Can still have overflow chains

Example: BAD CASE



Very full

Very empty

Need to move *m* here…

Would waste space…

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

NoteNNOT 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 <u>LIST</u> ⇒ MULTIKEY INDEX

(next)

e.g., <u>CREATE</u> <u>INDEX</u> foo <u>ON</u> R(A,B,C)

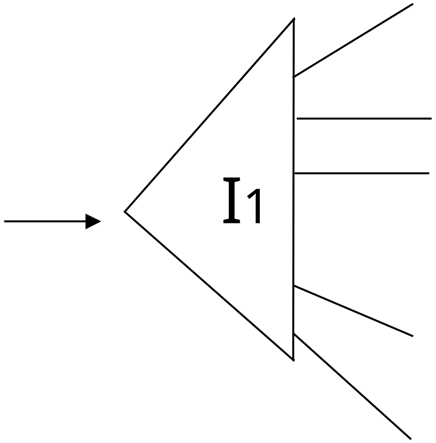
Multi-key Index

Motivation: Find records where

DEPT = “Toy” AND SAL > 50k

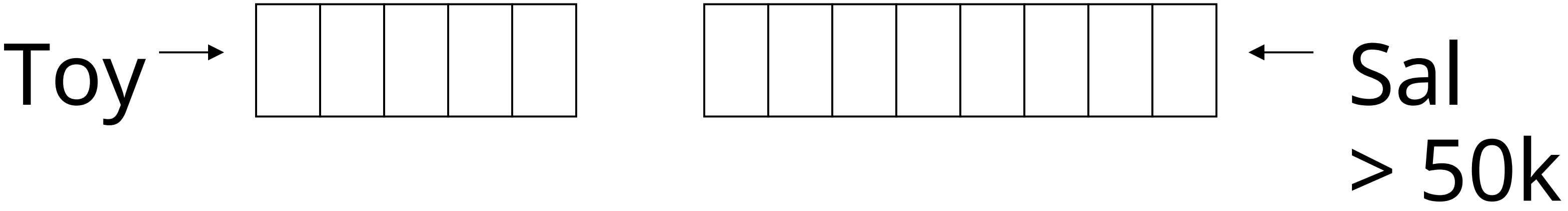
Strategy I:

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



Strategy II:

- Use 2 Indexes; Manipulate Pointers



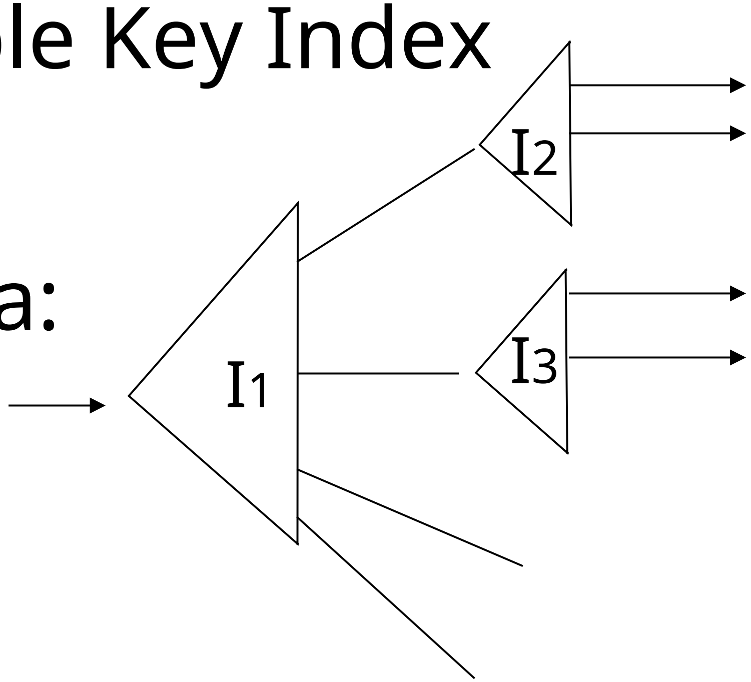
Toy

Sal
> 50k

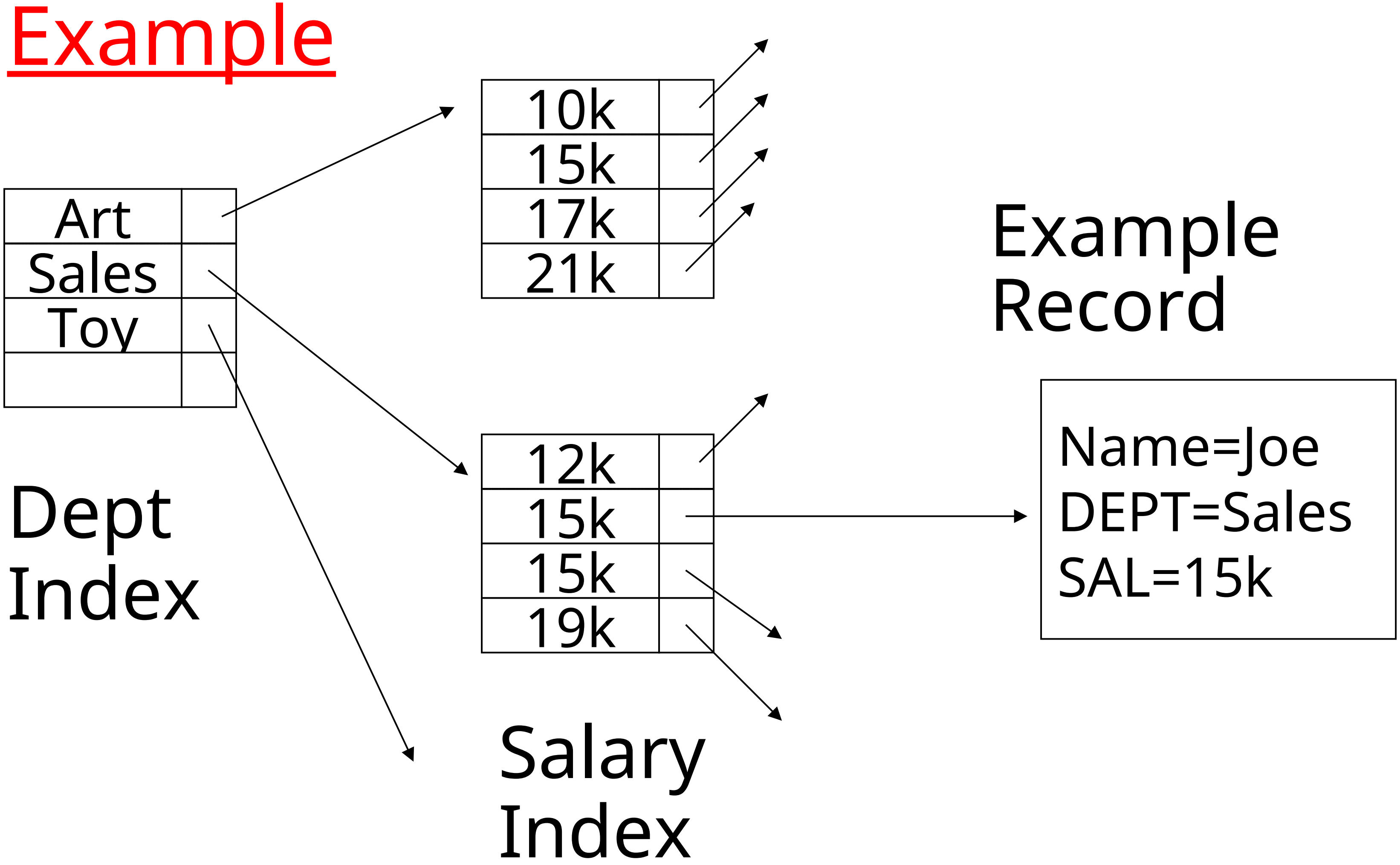
Strategy III:

- Multiple Key Index

One idea:



Example



Art
Sales
Toy

Dept Index

10k
15k
17k
21k

12k
15k
15k
19k

Salary Index

Example Record

Name=Joe
DEPT=Sales
SAL=15k

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