CS 525: Advanced Database Organization

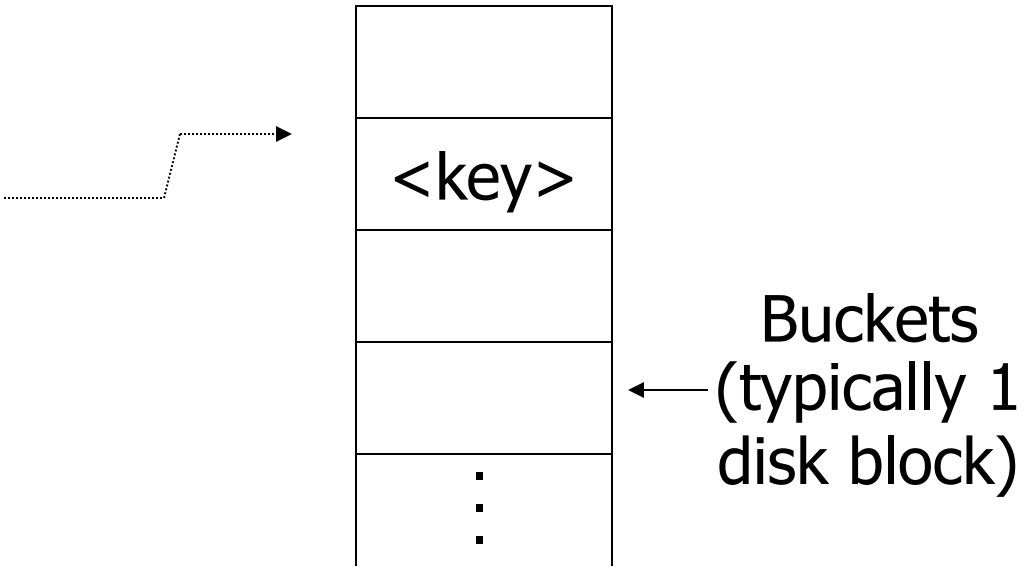
05: Hashing and More

Boris Glavic

Slides: adapted from a course taught by Hector Garcia-Molina, Stanford InfoLab

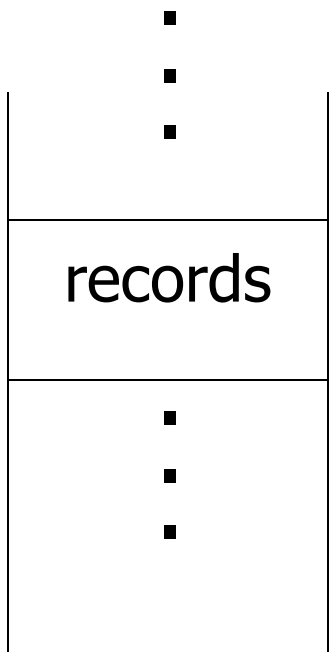
Hashing

key → h(key)



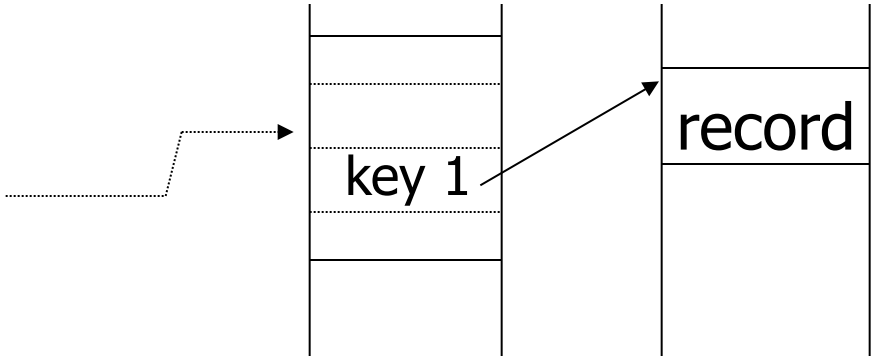
Two alternatives

(1) key → h(key)



Two alternatives

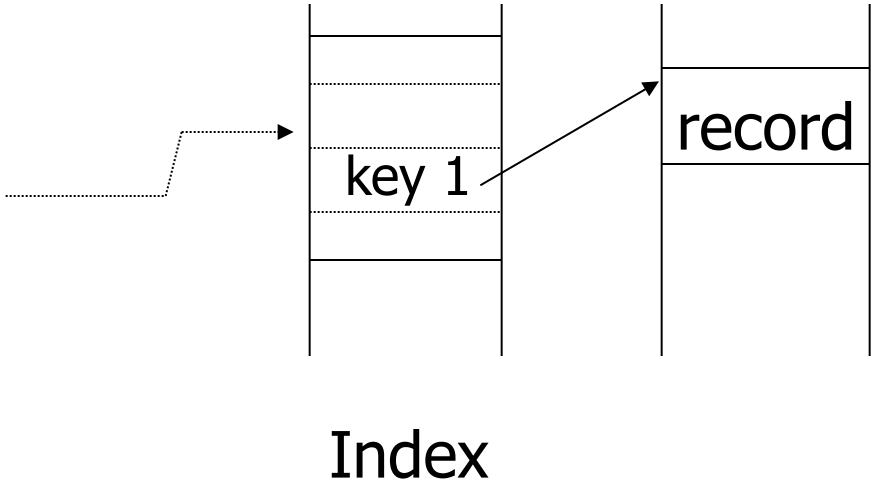
(2) key → h(key)



Index

Two alternatives

(2) key → h(key)



- Alt (2) for "secondary" search key

Example hash function

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

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

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

Good hash function:

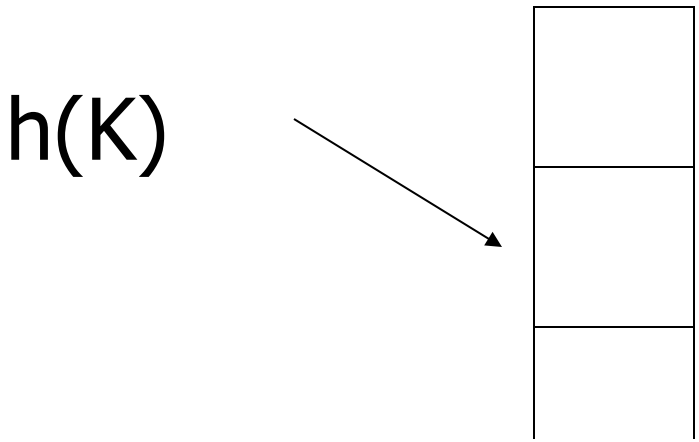
☞ Expected number of keys/bucket is the same for all buckets

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

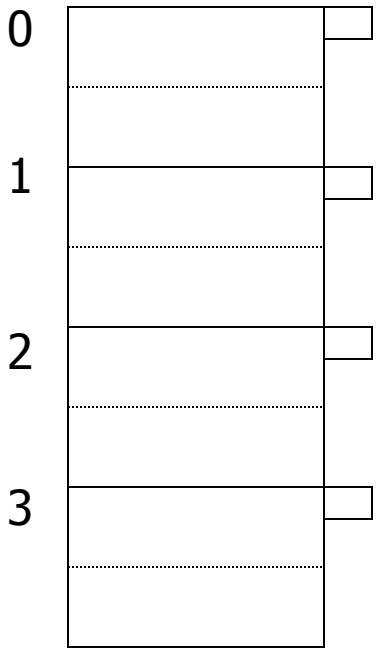
INSERT:

h(a) = 1

h(b) = 2

h(c) = 1

h(d) = 0



EXAMPLE 2 records/bucket

INSERT:

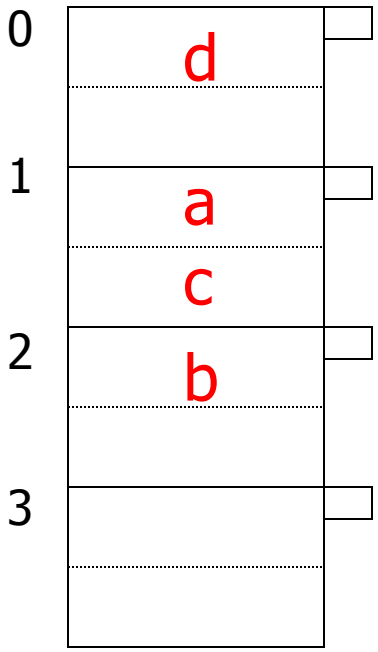
h(a) = 1

h(b) = 2

h(c) = 1

h(d) = 0

h(e) = 1



EXAMPLE 2 records/bucket

INSERT:

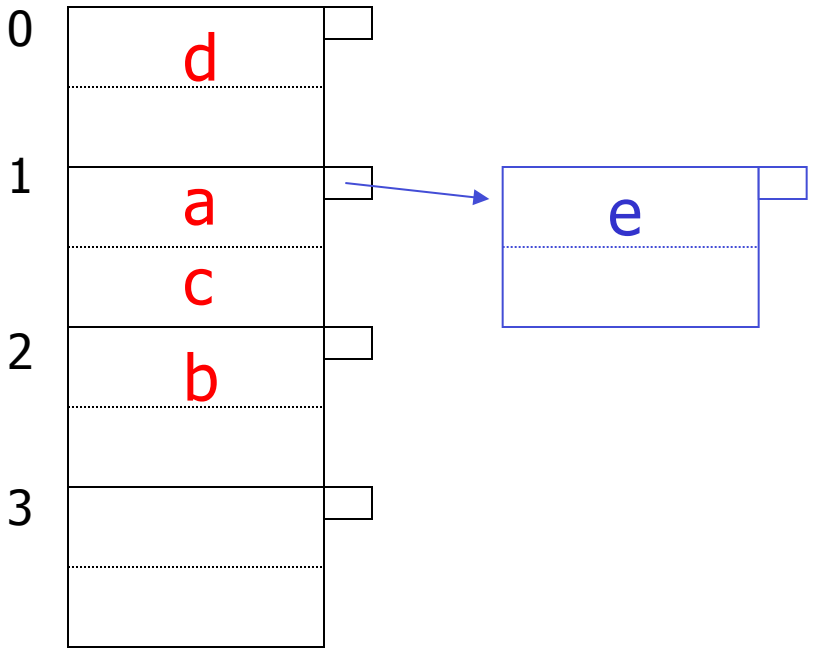
h(a) = 1

h(b) = 2

h(c) = 1

h(d) = 0

h(e) = 1

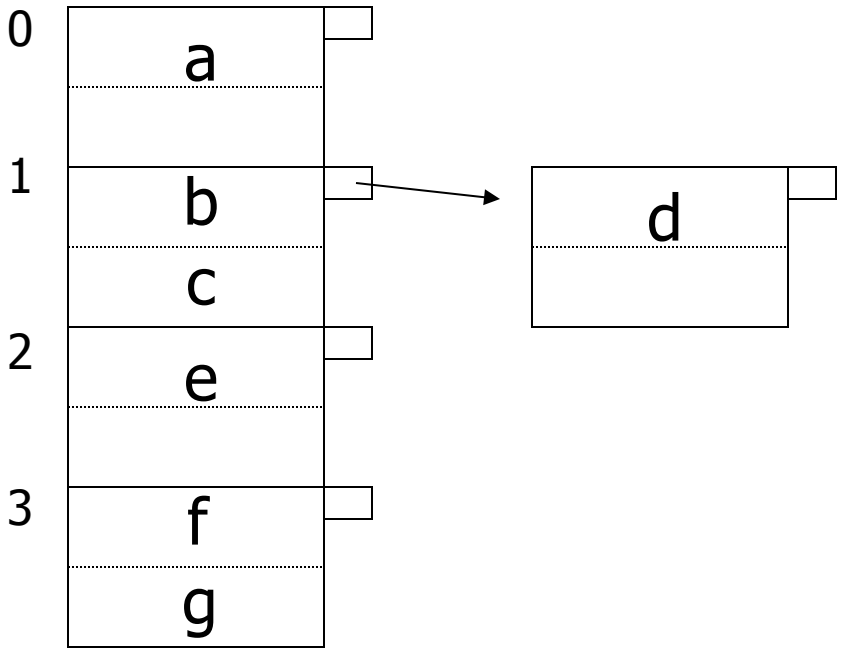


EXAMPLE: deletion

Delete:

e

f



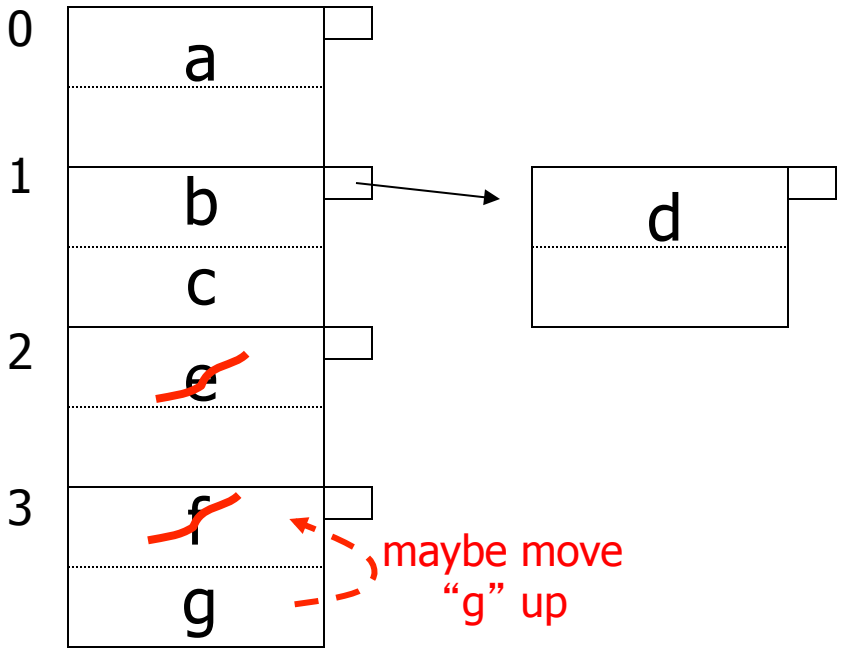
EXAMPLE: deletion

Delete:

e

f

c



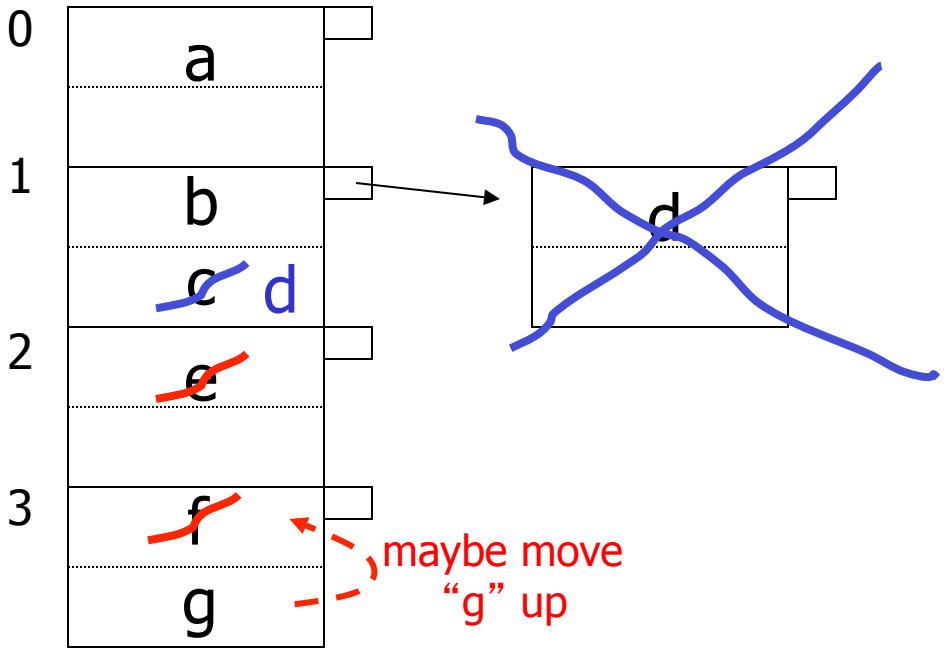
EXAMPLE: deletion

Delete:

e

f

c



Rule of thumb:

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

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

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?

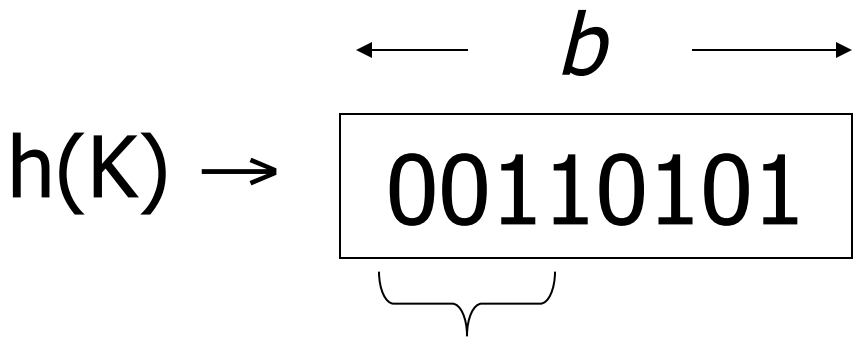
- Overflows and reorganizations
- Dynamic hashing

How do we cope with growth?

- Overflows and reorganizations
- Dynamic hashing
  - Extensible
  - Linear

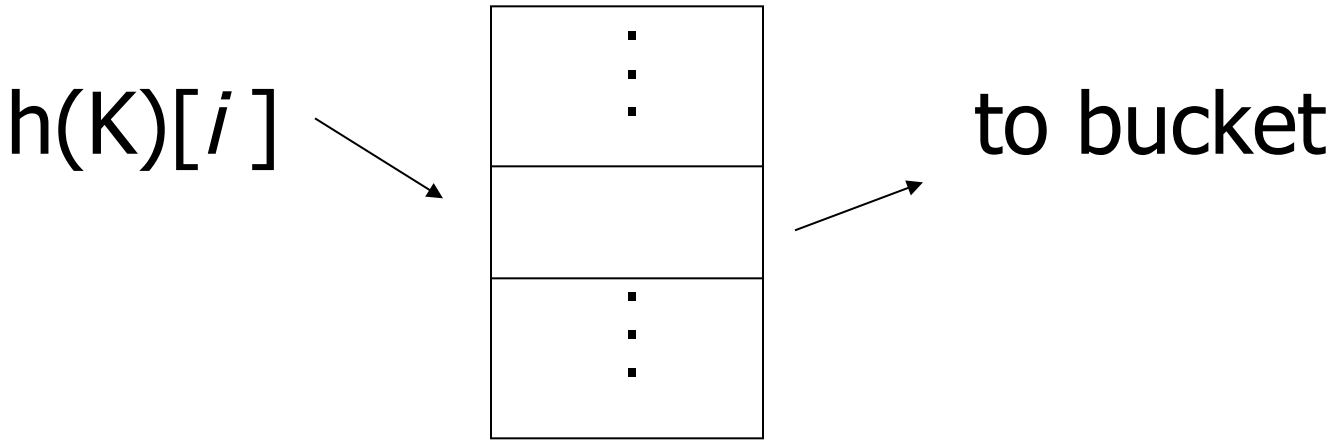
Extensible hashing: two ideas

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

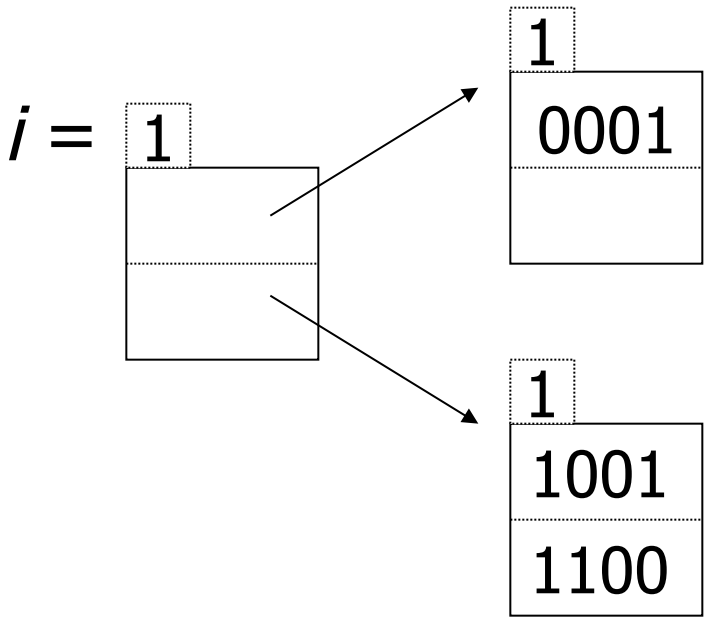


use *i* → grows over time….

(b) Use directory

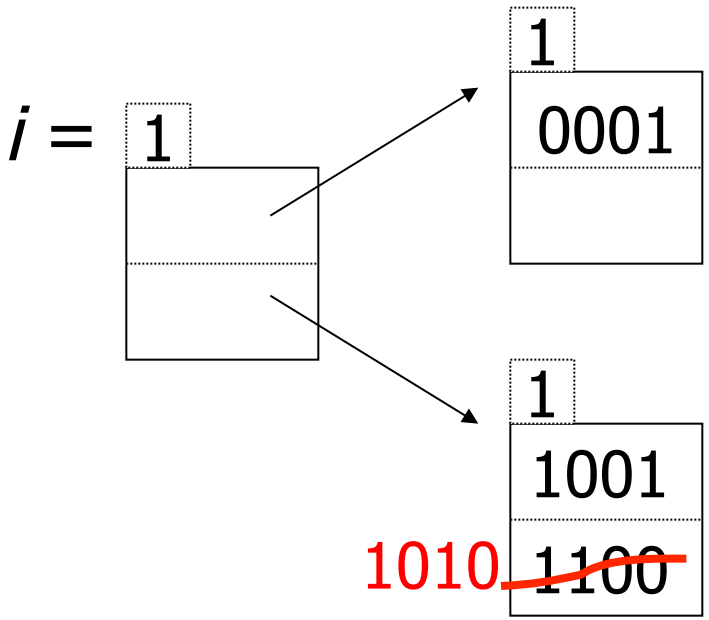


Example: h(k) is 4 bits; 2 keys/bucket

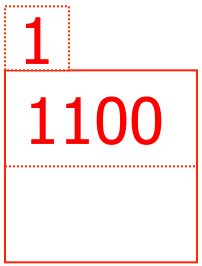


Insert 1010

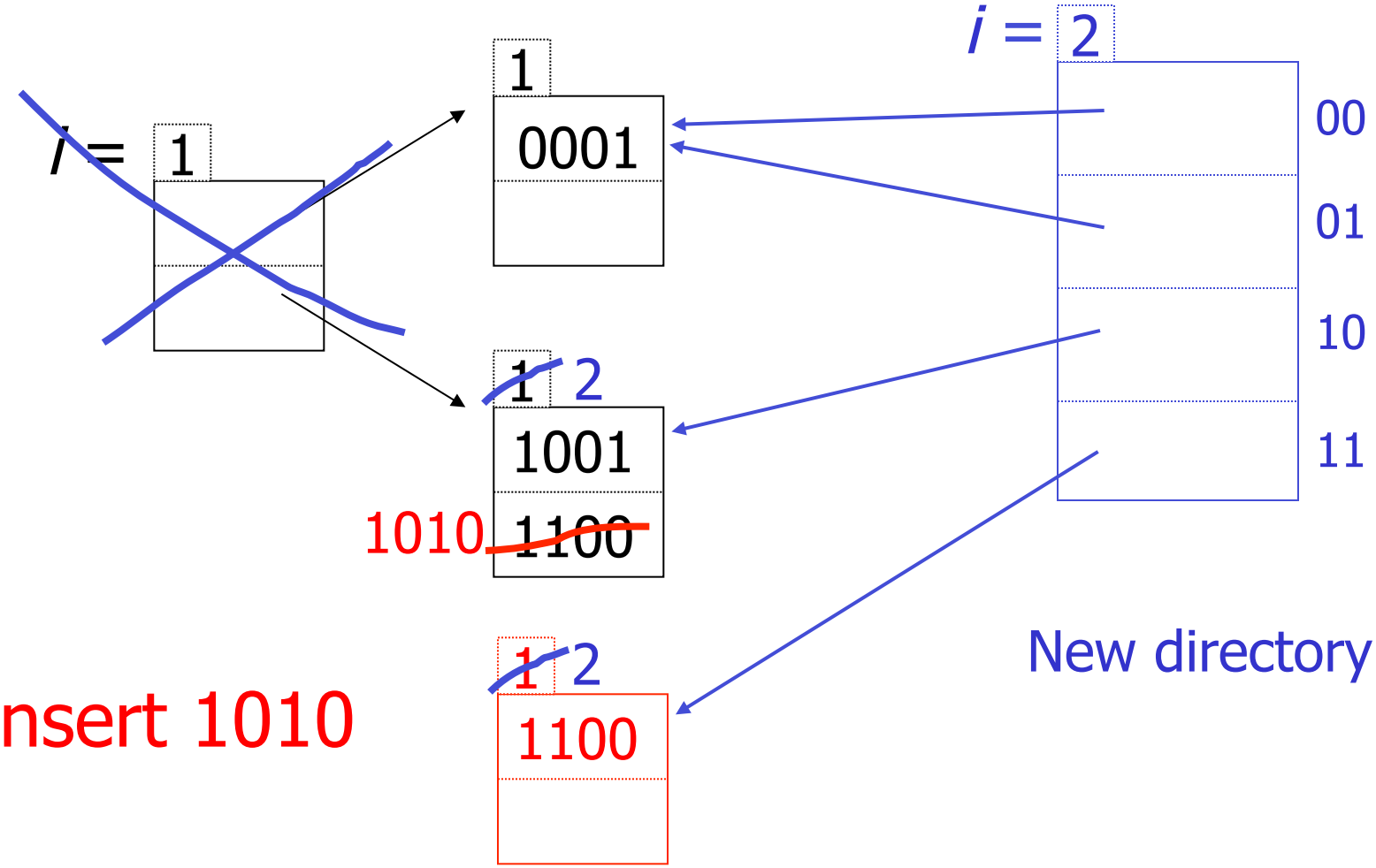
Example: h(k) is 4 bits; 2 keys/bucket



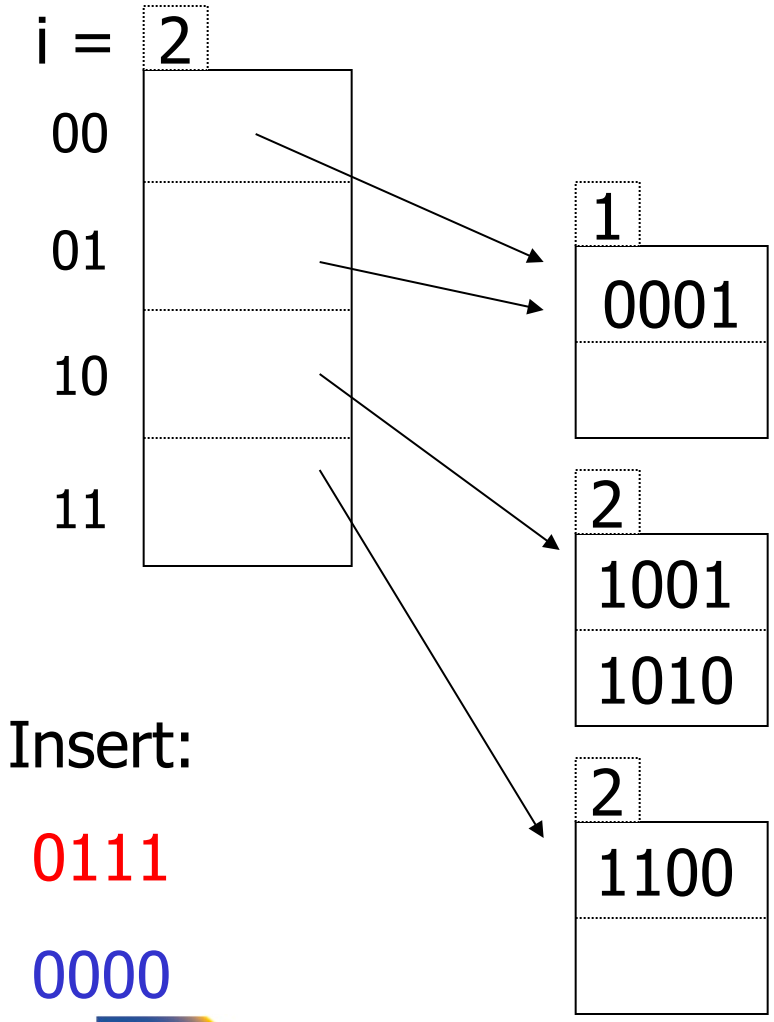
Insert 1010



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



Example continued

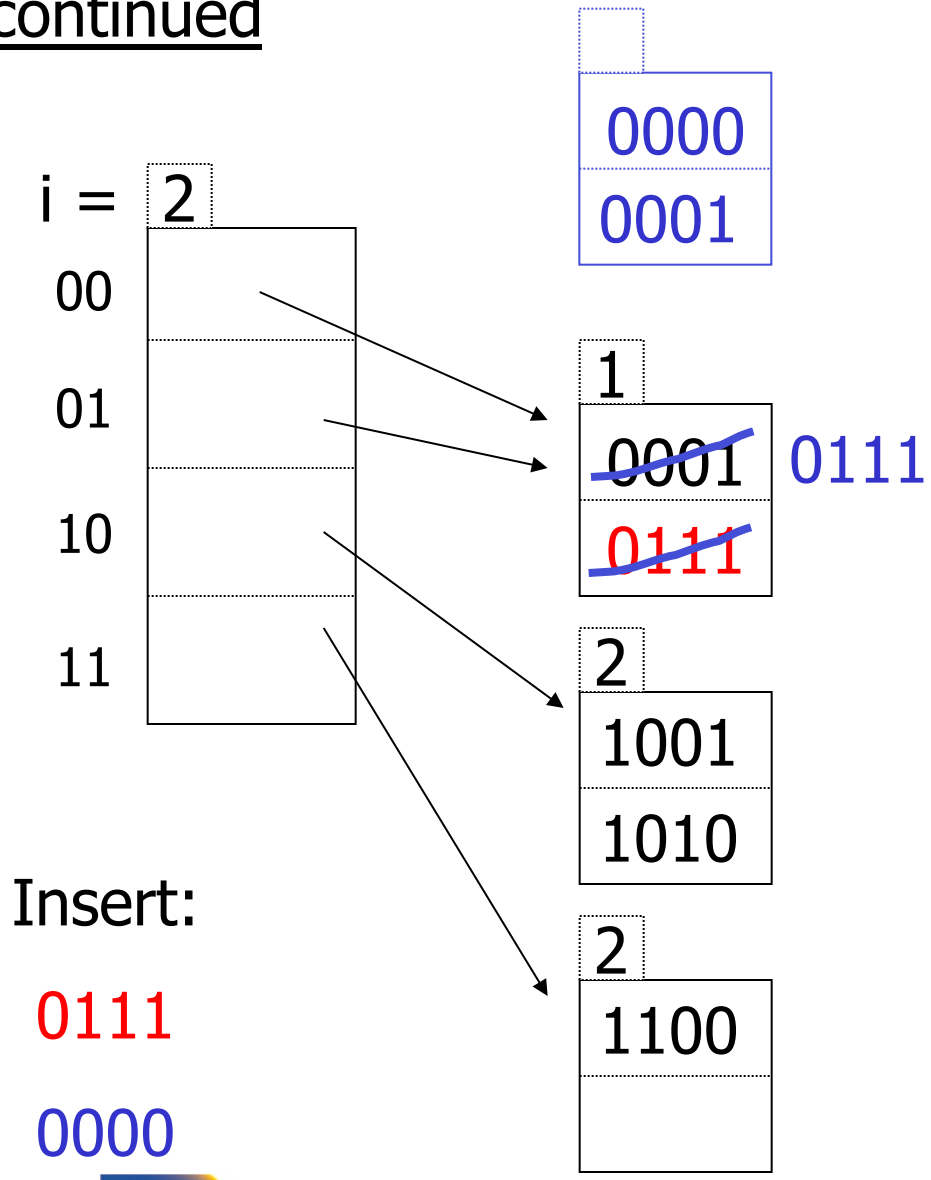


Insert:

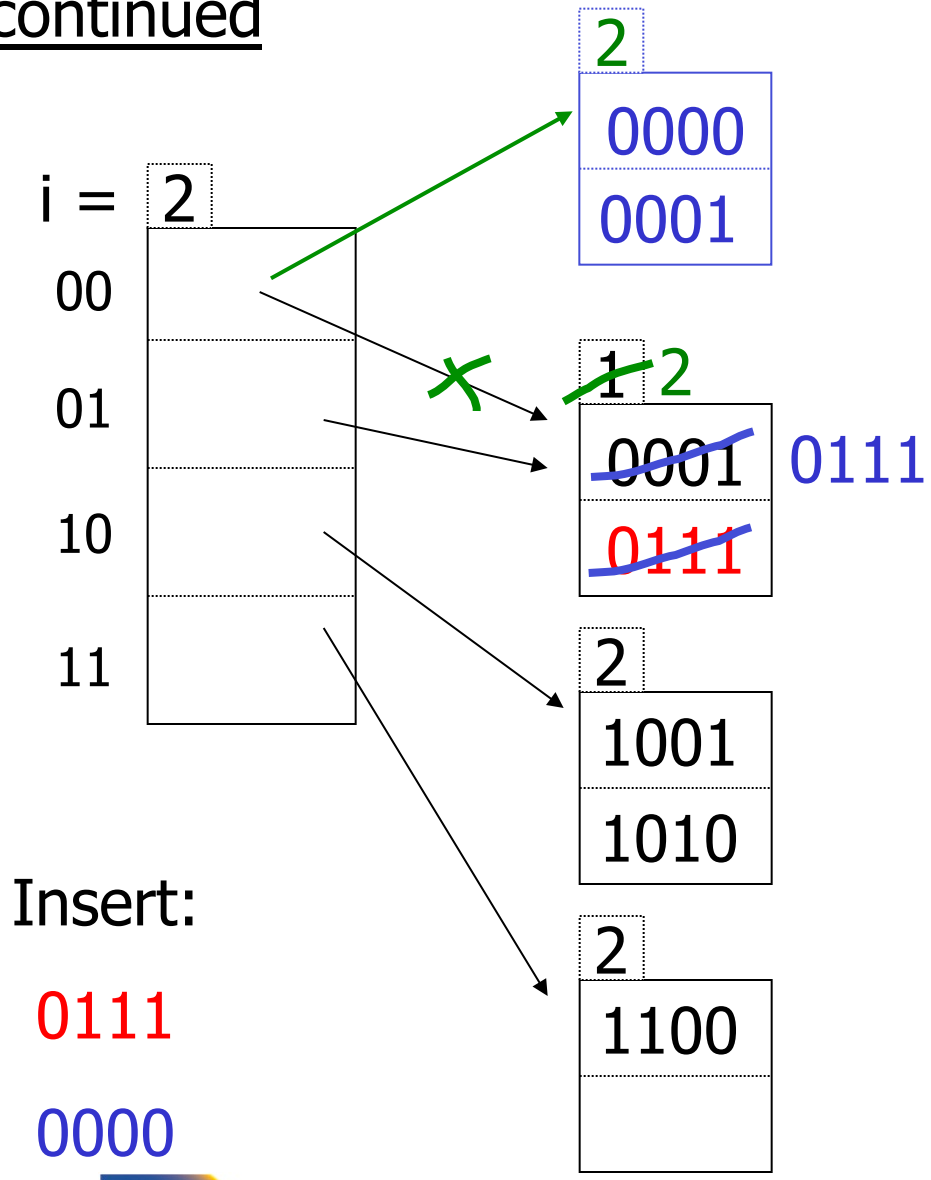
0111

0000

Example continued



Example continued

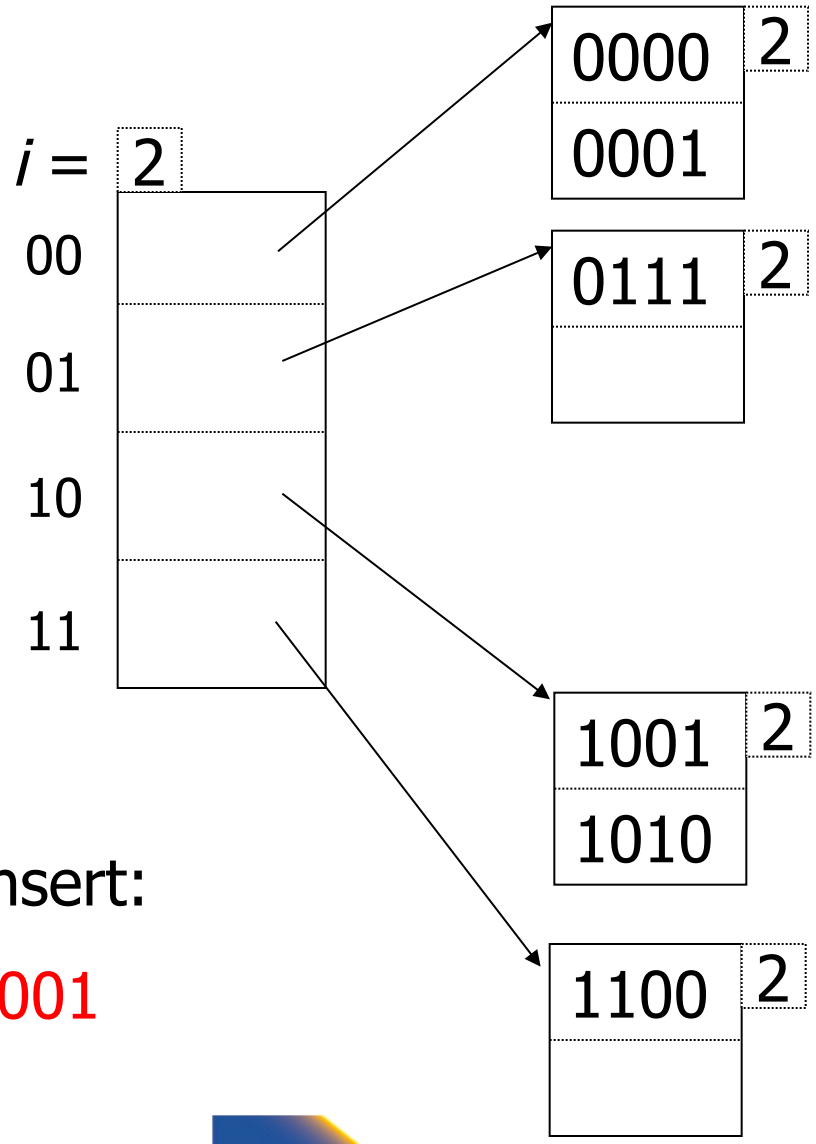


Insert:

0111

0000

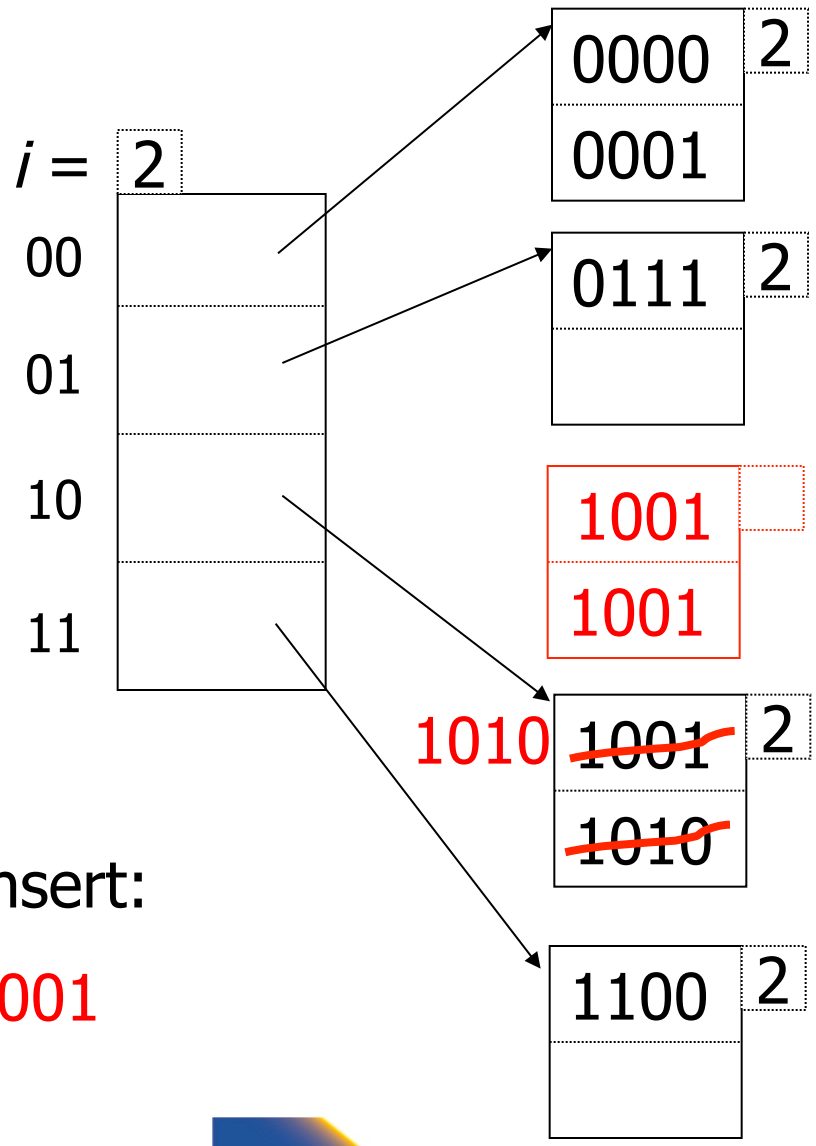
Example continued



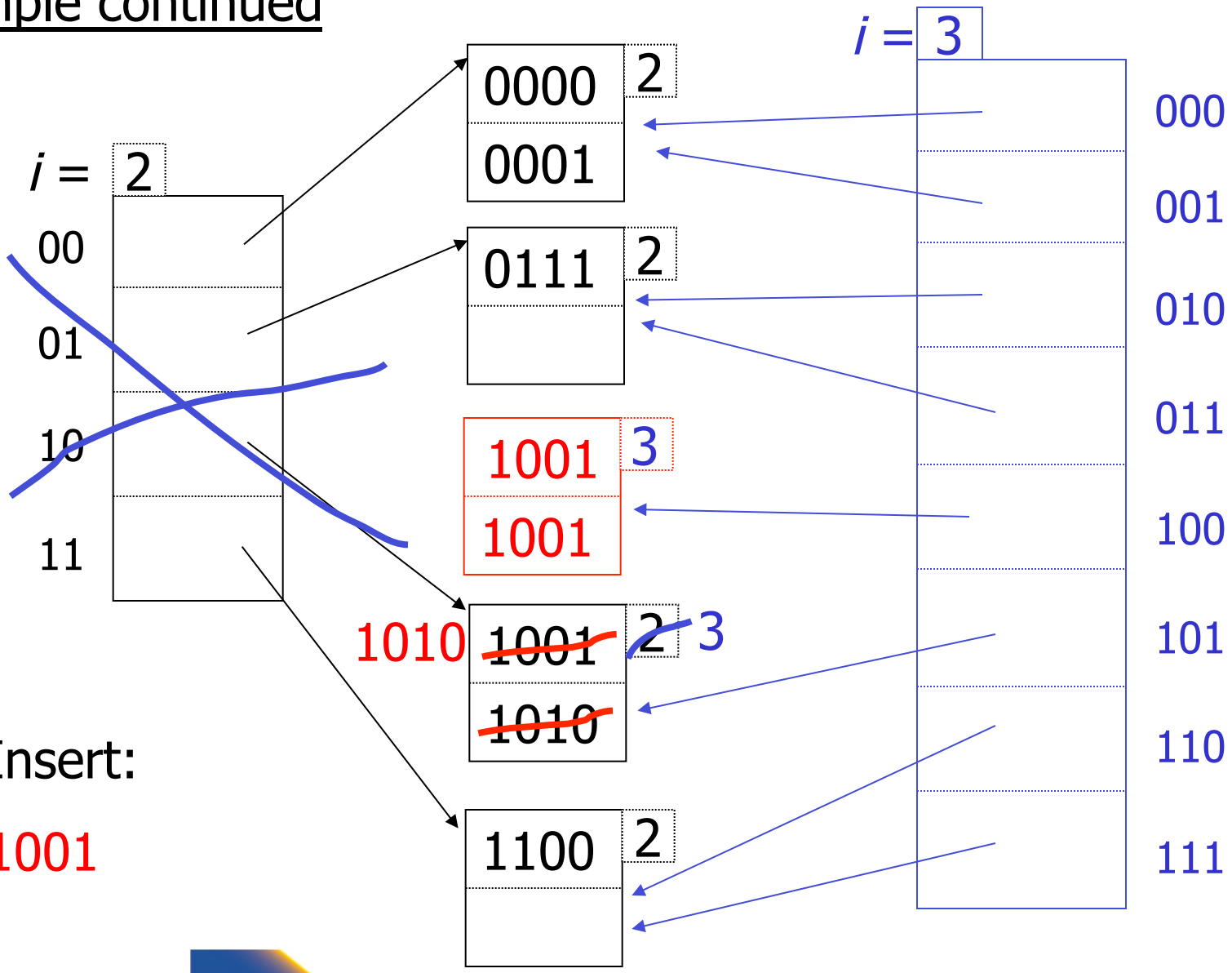
Insert:

1001

Example continued



Example continued



Extensible hashing: deletion

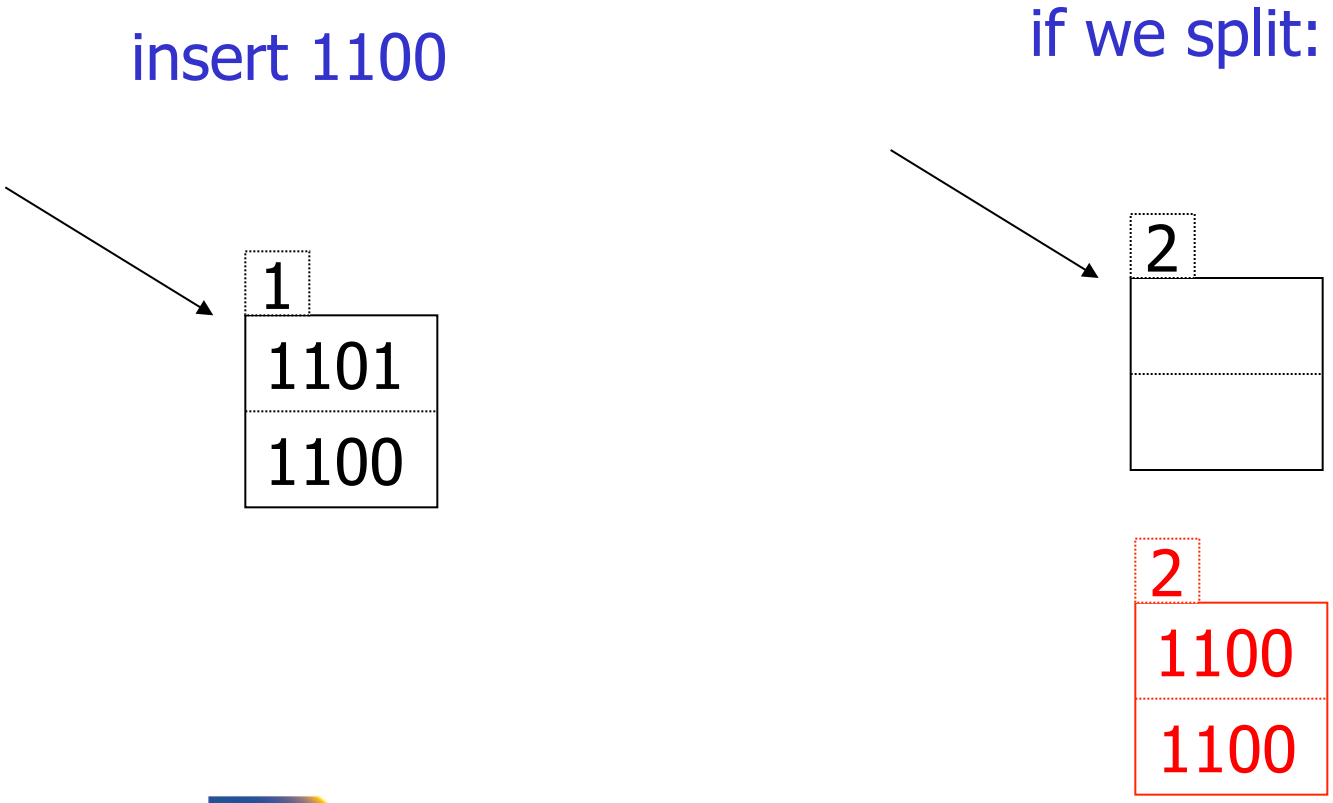
- No merging of blocks
- Merge blocks
  and cut directory if possible
  (Reverse insert procedure)

Deletion example:

- Run thru insert example in reverse!

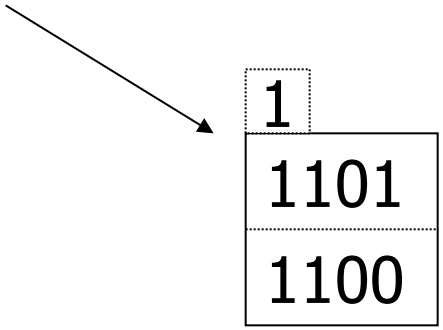
Note: Still need overflow chains

- Example: many records with duplicate keys

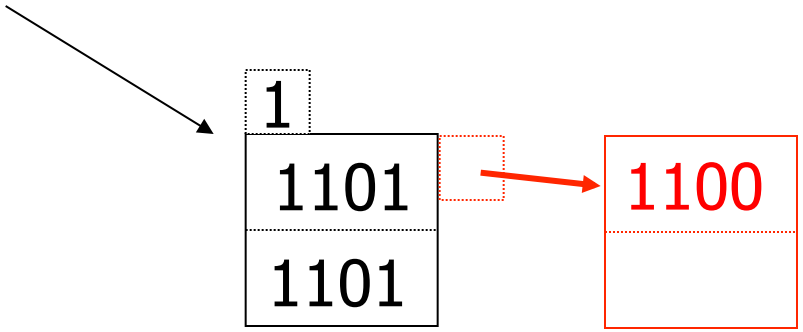


Solution: overflow chains

insert 1100



add overflow block:



Summary Extensible hashing

⊕ Can handle growing files

- with less wasted space
- with no full reorganizations

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

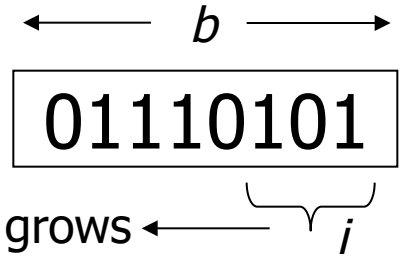
(Now it fits, now it does not)

Linear hashing

- Another dynamic hashing scheme

Two ideas:

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

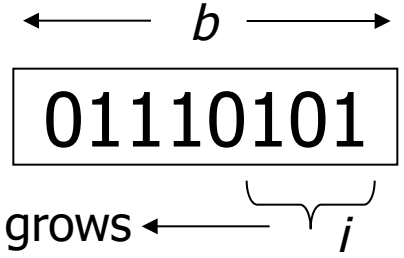


Linear hashing

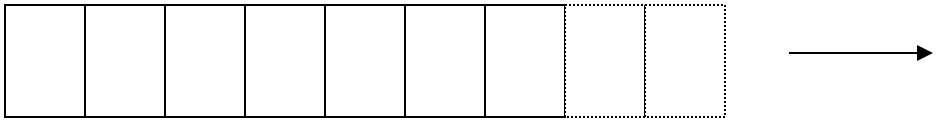
- Another dynamic hashing scheme

Two ideas:

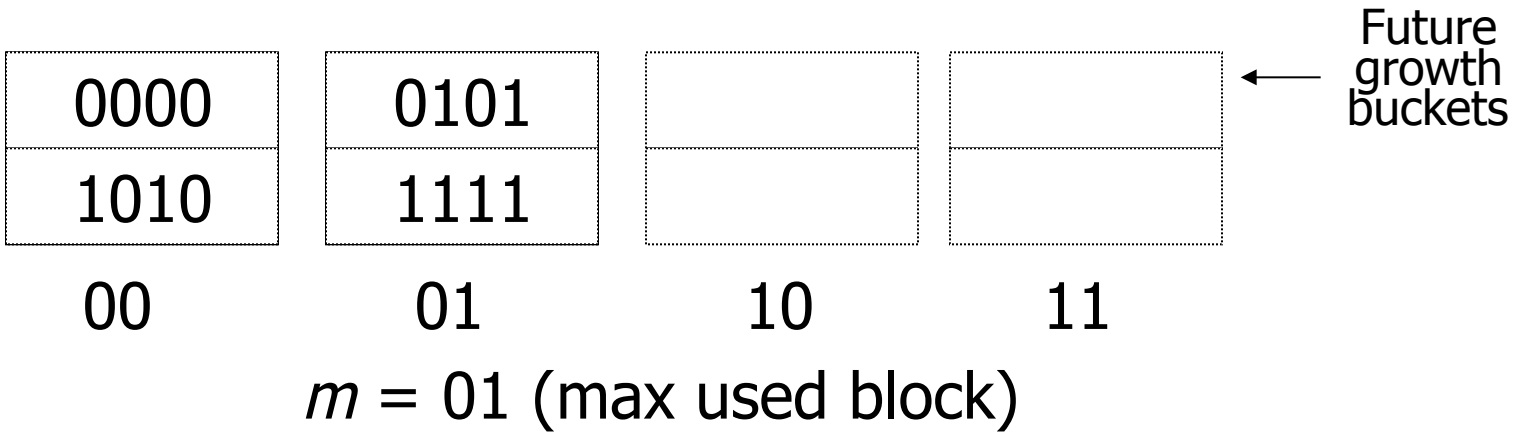
(a) Use *i* low order bits of hash



(b) File grows linearly

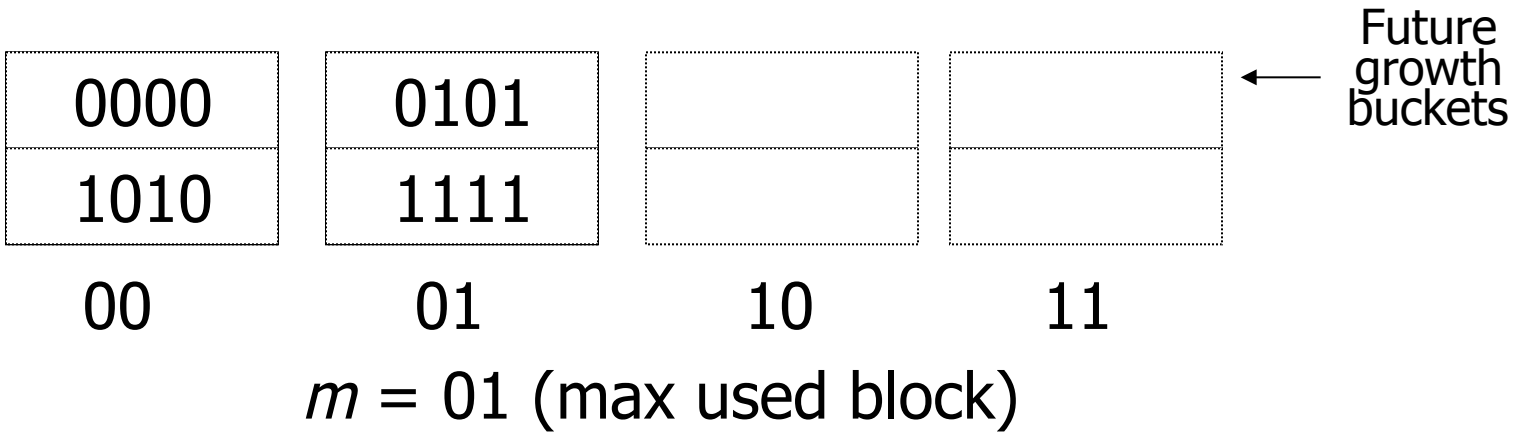


Example b=4 bits, i =2, 2 keys/bucket



m = 01 (max used block)

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



m = 01 (max used block)

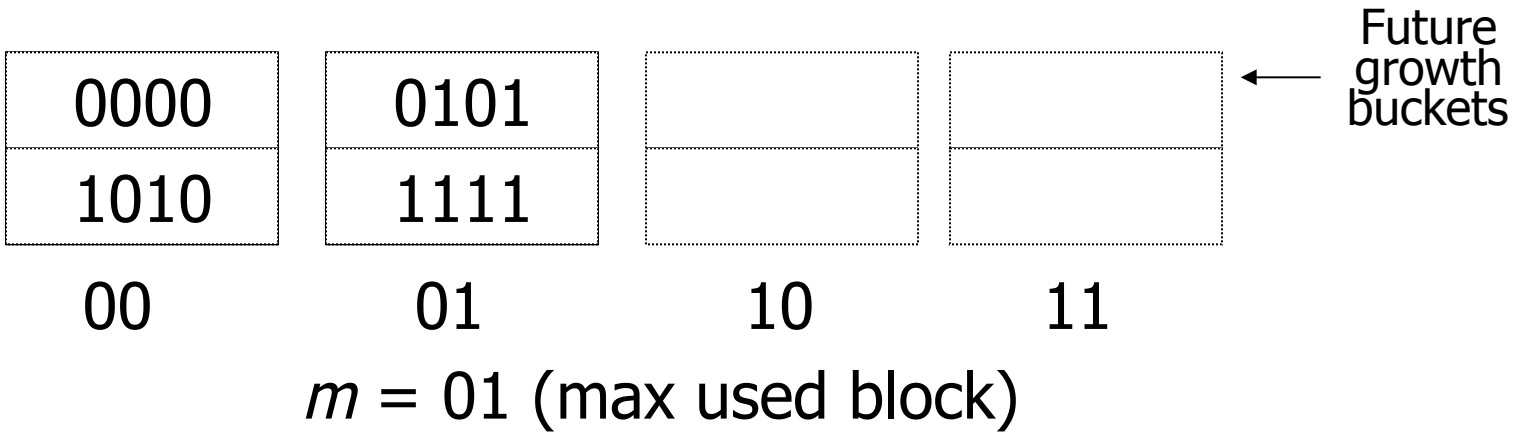
Rule If h(k)[i] ≤ m, then

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

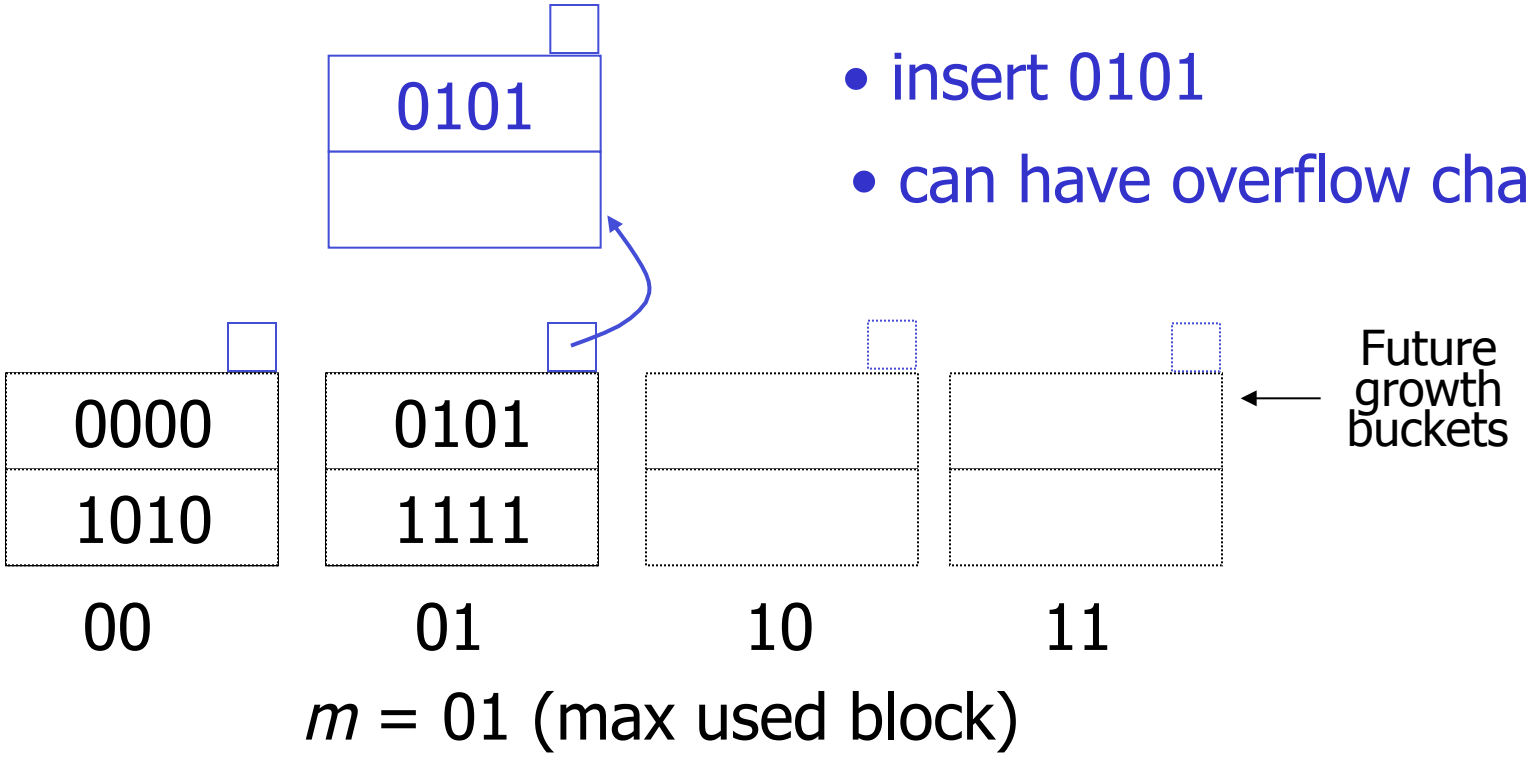


Rule If h(k)[i] ≤ m, then

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)

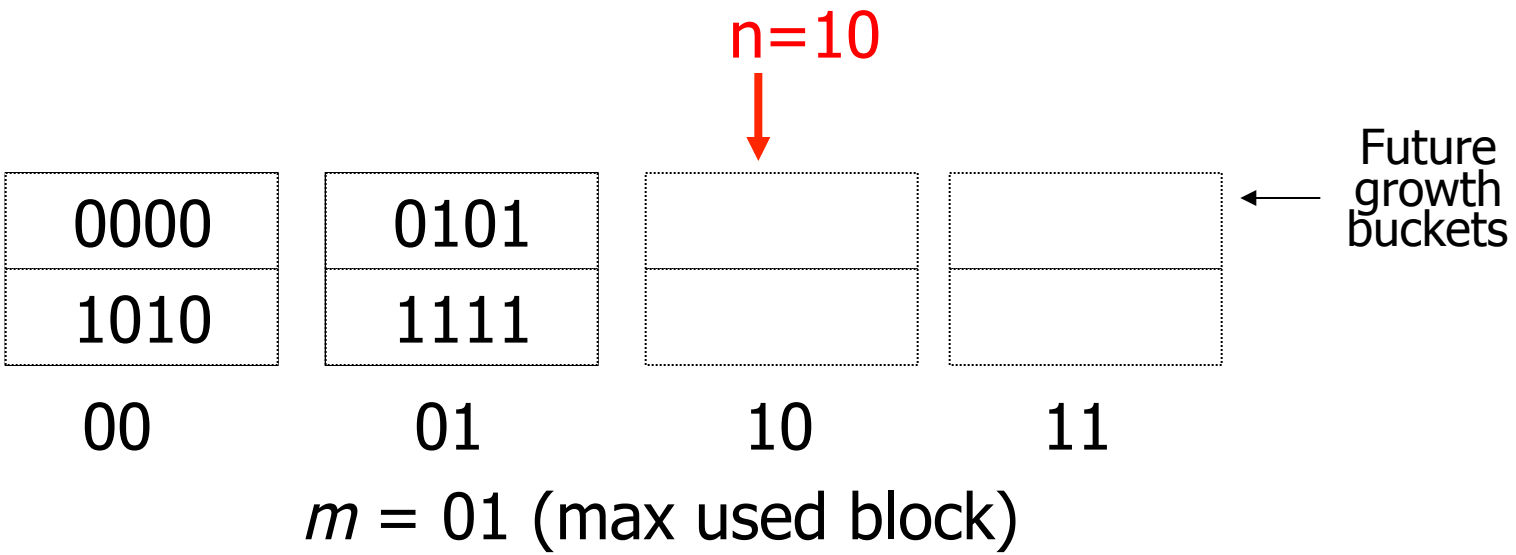
Rule If h(k)[i] ≤ m, then

look at bucket h(k)[i]

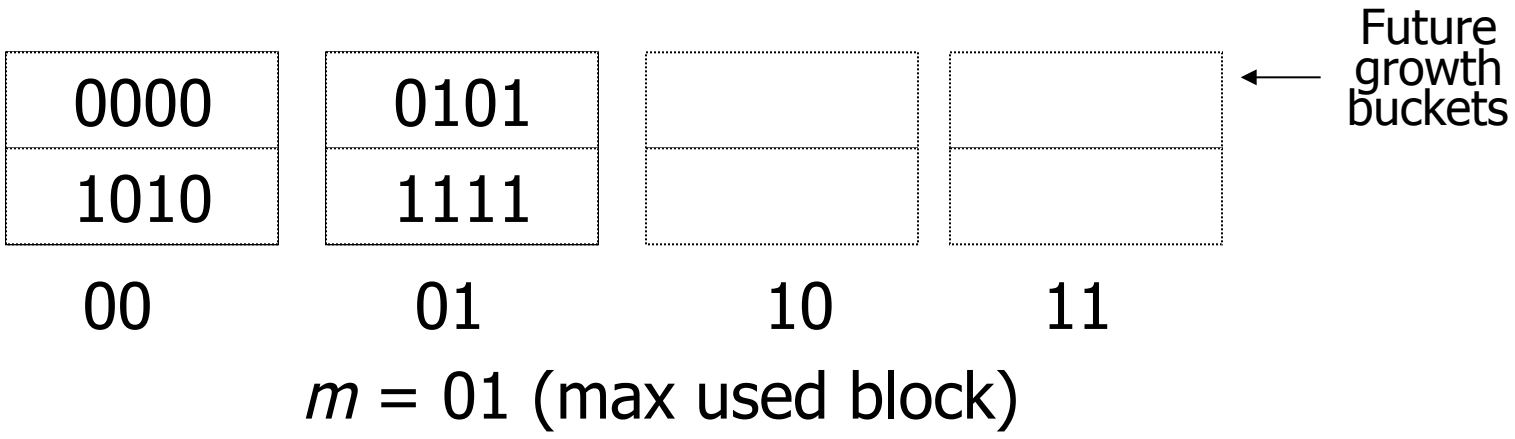
else, look at bucket h(k)[i] - 2^{i-1}

Note

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

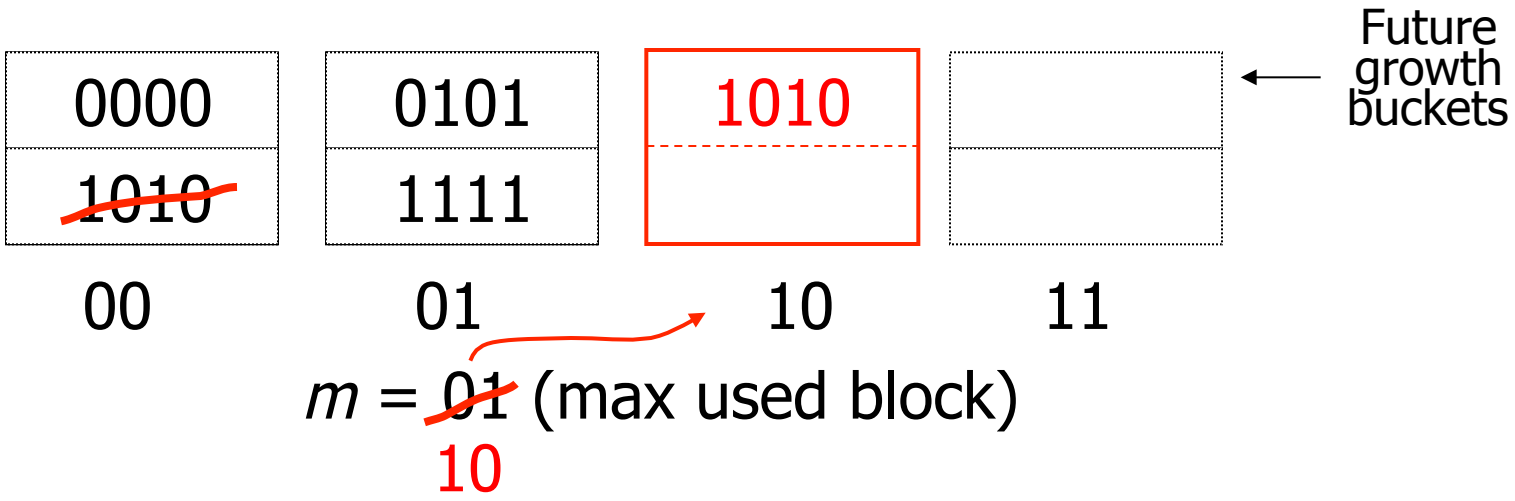


Example b=4 bits, i =2, 2 keys/bucket

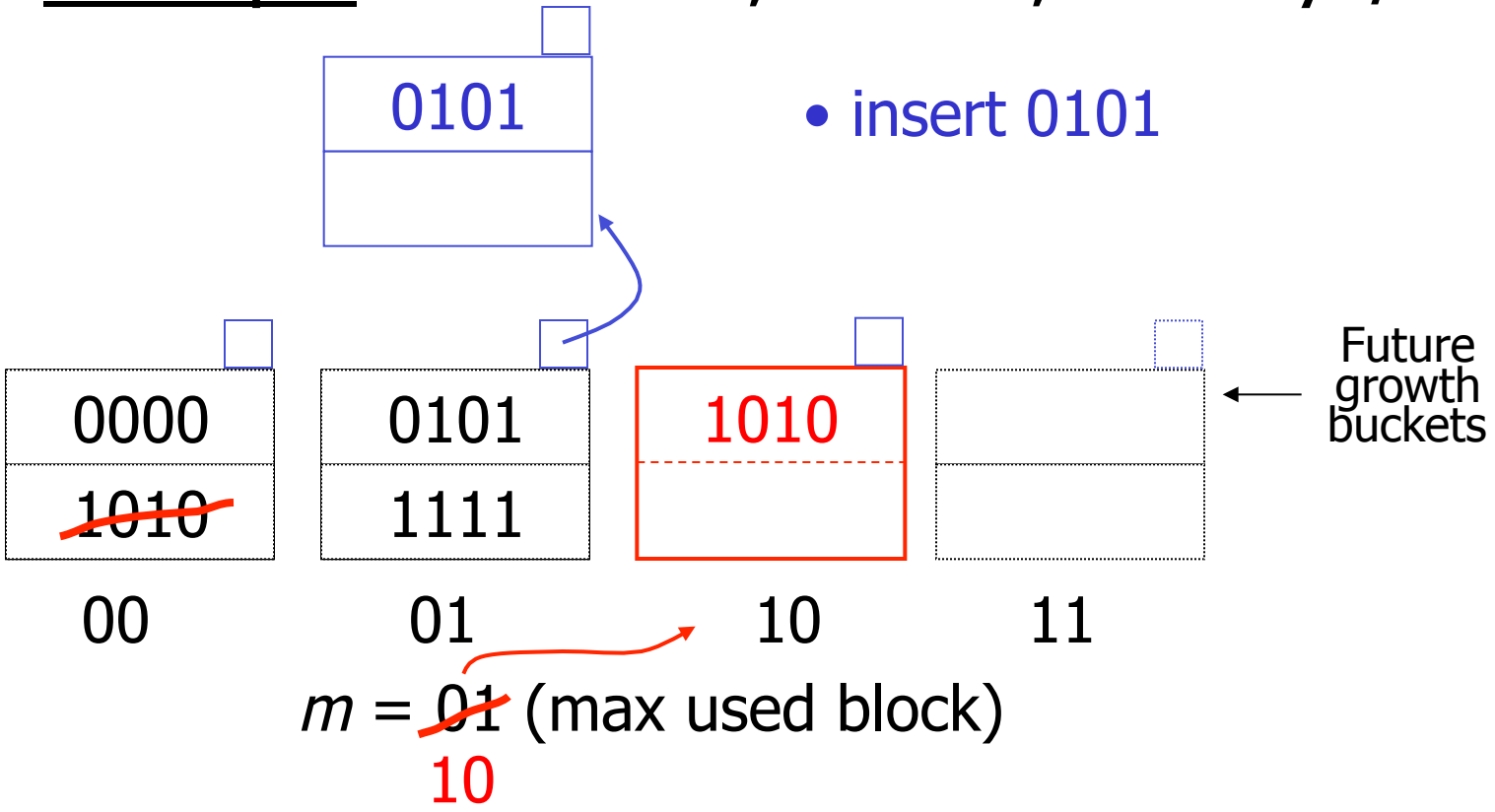


m = 01 (max used block)

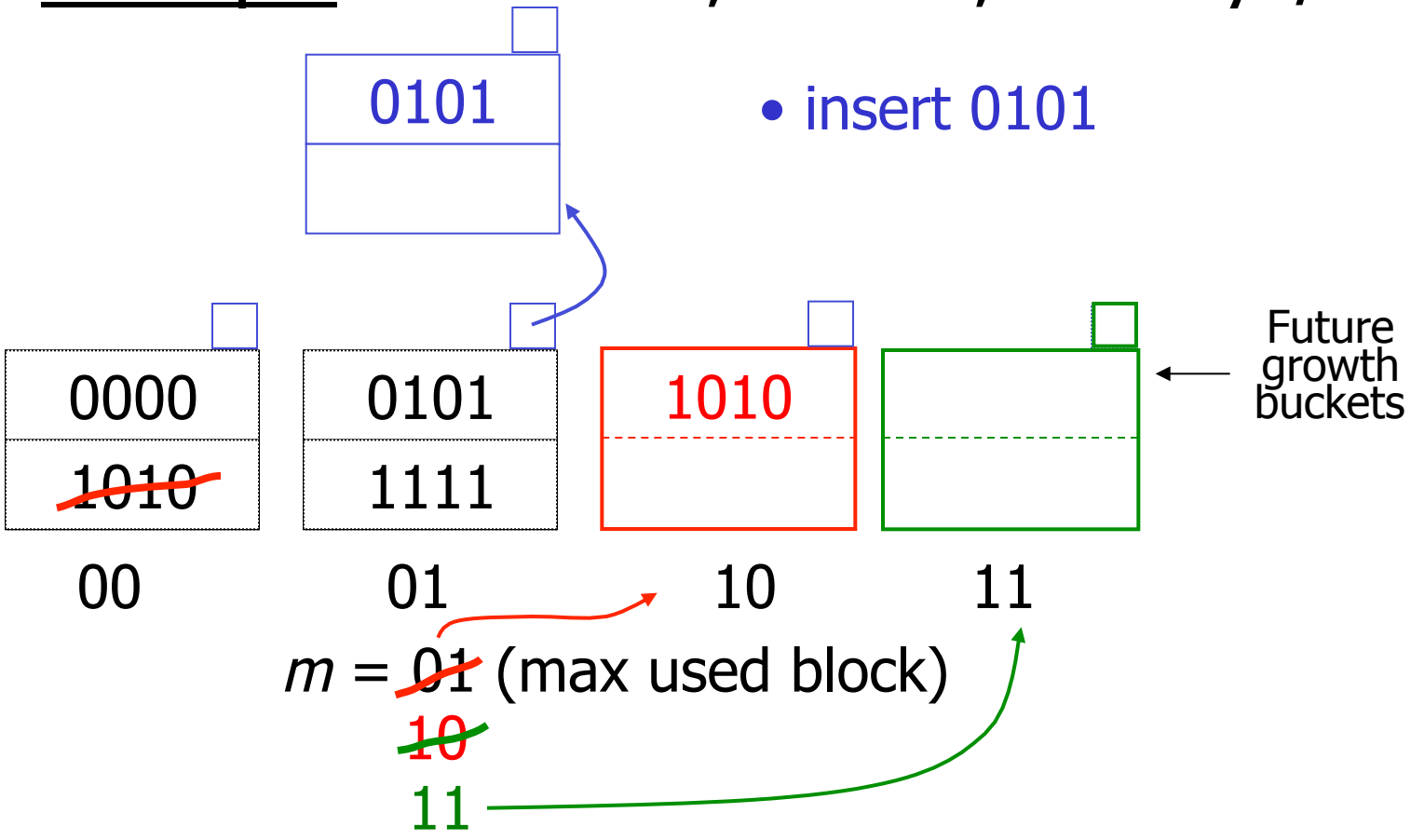
Example b=4 bits, i =2, 2 keys/bucket



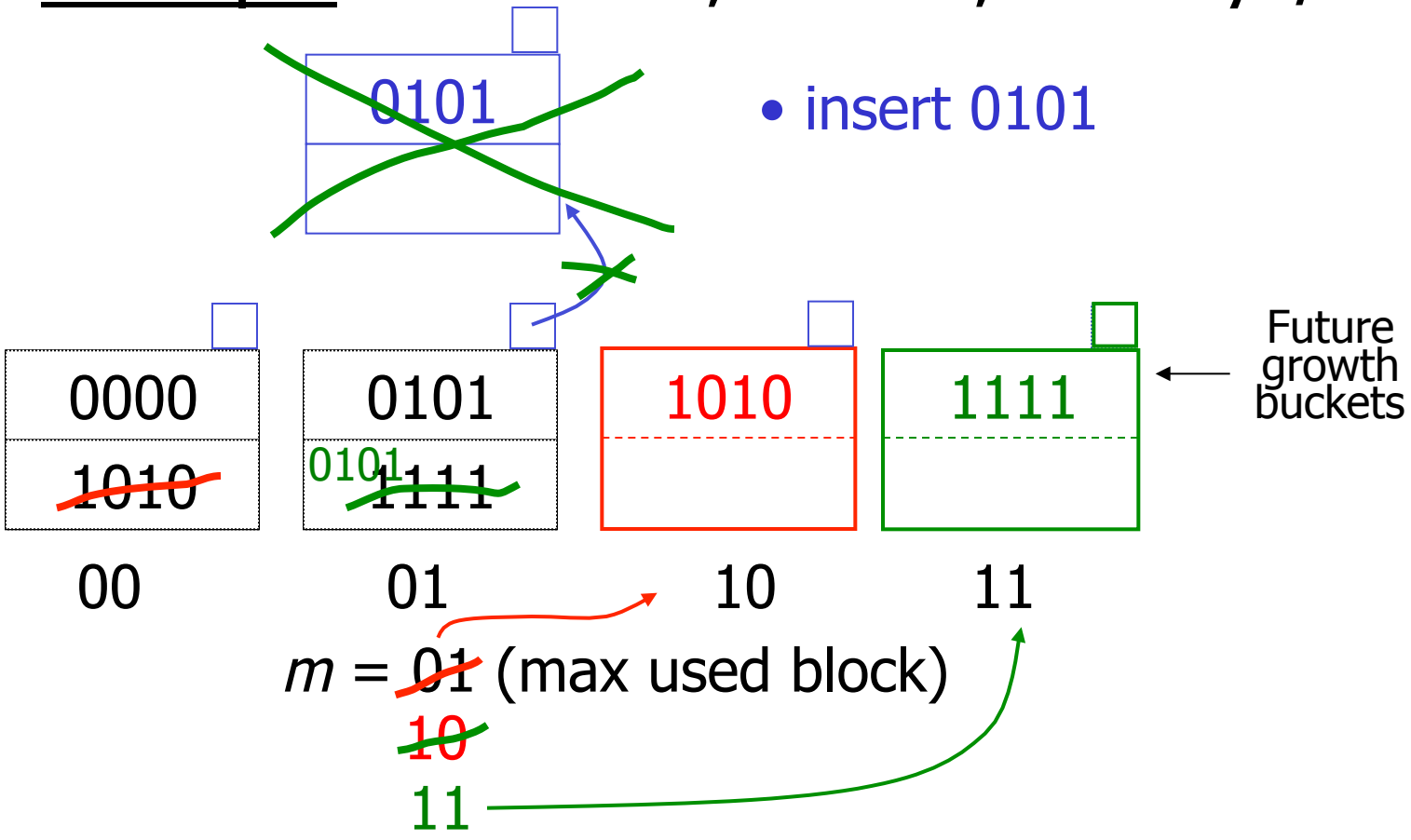
Example b=4 bits, i =2, 2 keys/bucket



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

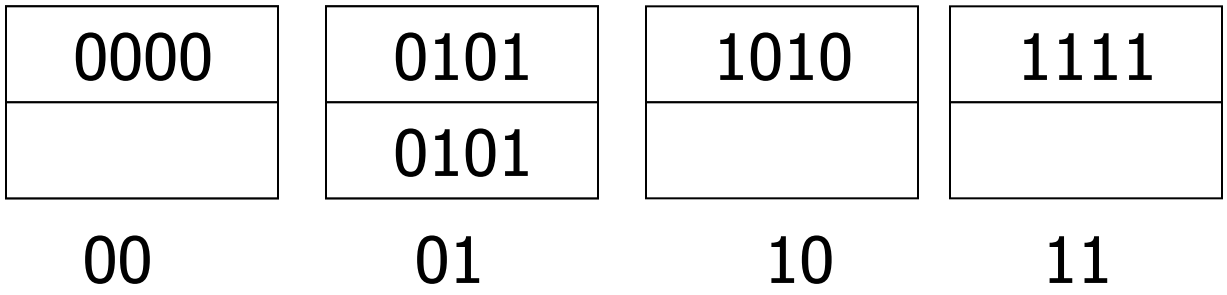


Example b=4 bits, i =2, 2 keys/bucket



Example Continued: How to grow beyond this?

$i = 2$

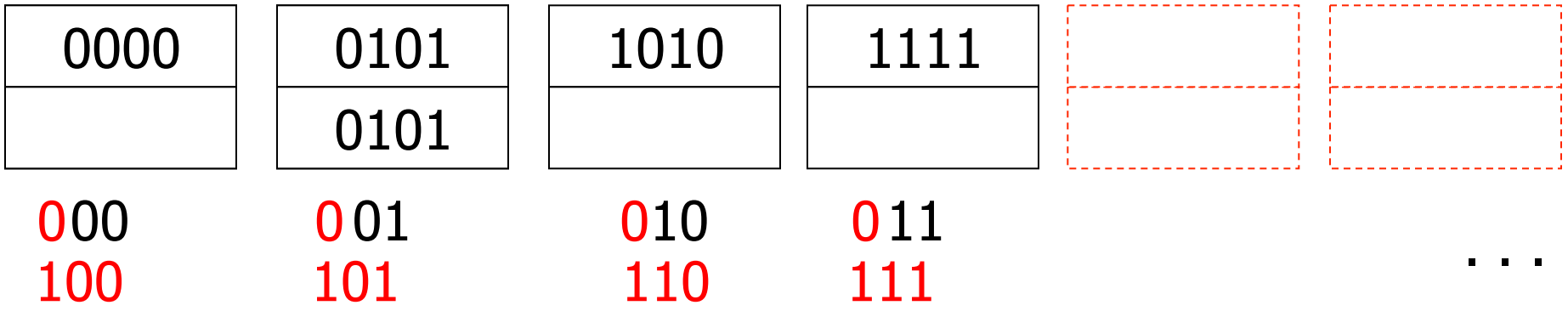


. . .

$m = 11$ (max used block)

Example Continued: How to grow beyond this?

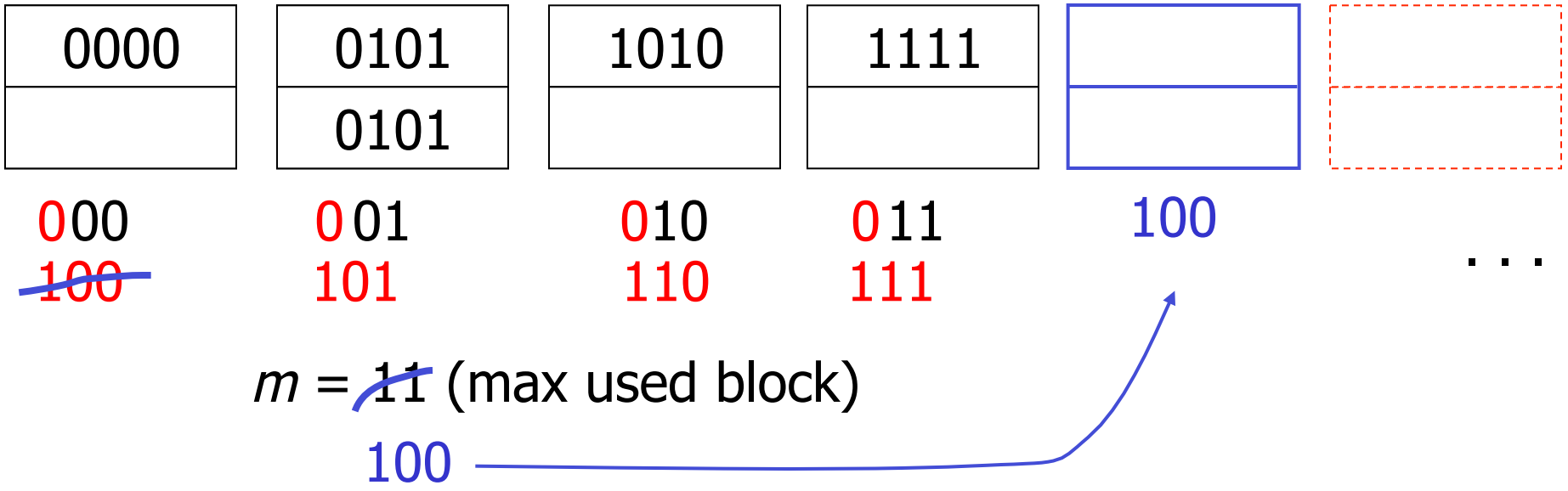
$i = 2$ ~~2~~ 3



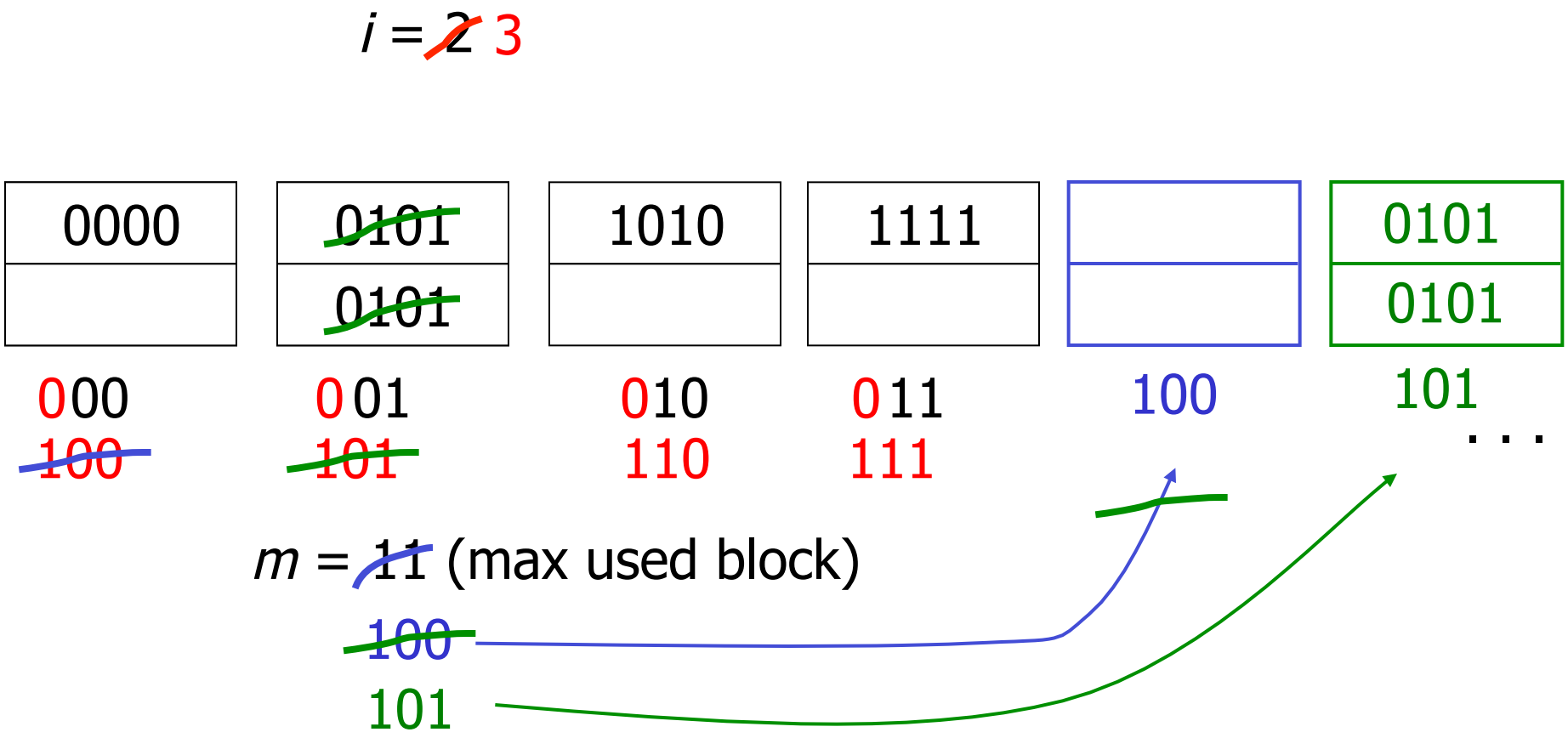
$m = 11$ (max used block)

Example Continued: How to grow beyond this?

$i = \cancel{2}\ 3$



Example Continued: How to grow beyond this?



☛ When do we expand file?

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

☛ When do we expand file?

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

- If U > threshold then increase *m*
  (and maybe *i*)

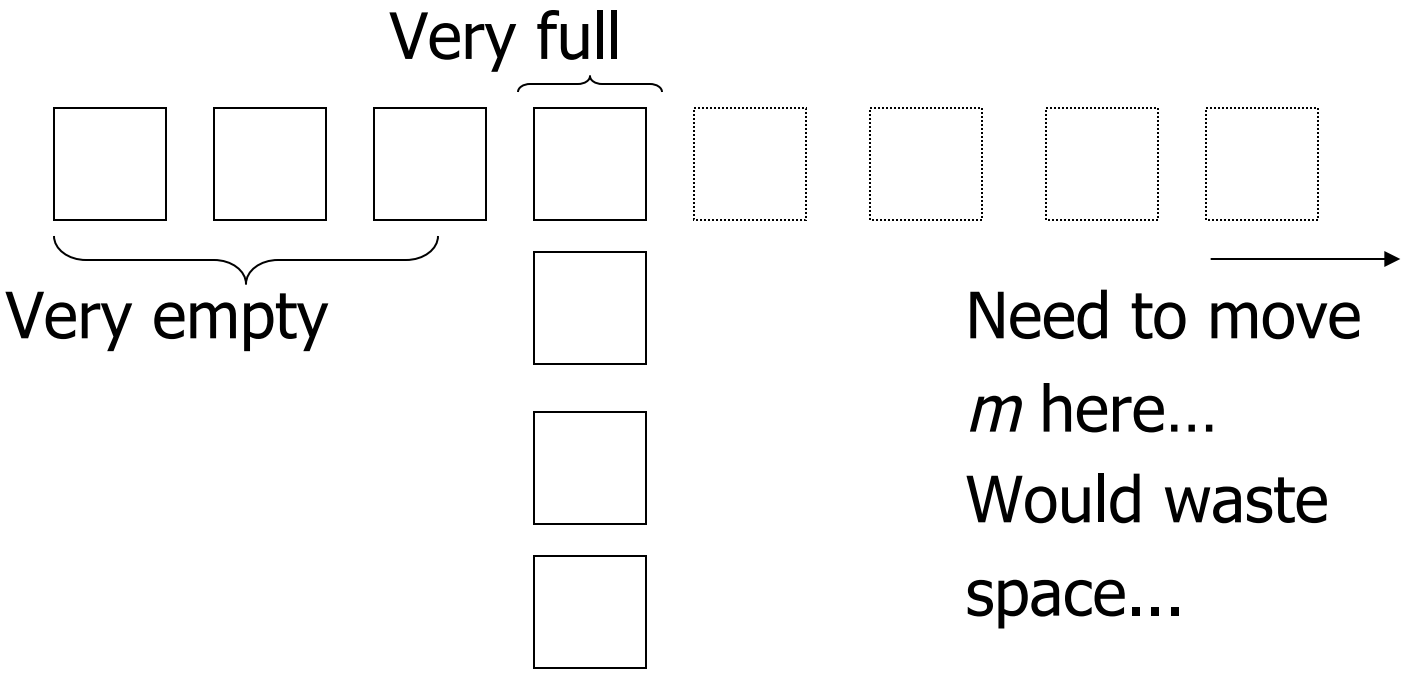
Summary Linear Hashing

⊕ 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



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:

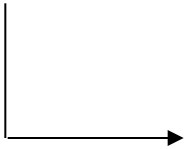
  e.g., SELECT

  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...

Note ATTRIBUTE LIST ⇒ MULTIKEY INDEX

(next)

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

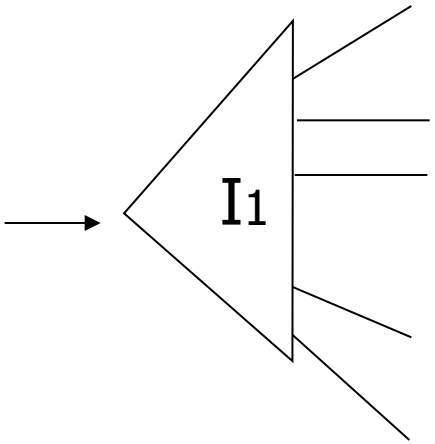
Multi-key Index

Motivation: Find records where

DEPT = “Toy” AND SAL > 50k

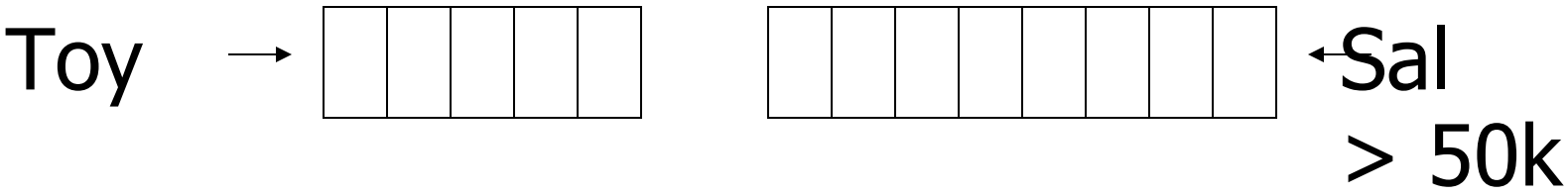
Strategy I:

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



Strategy II:

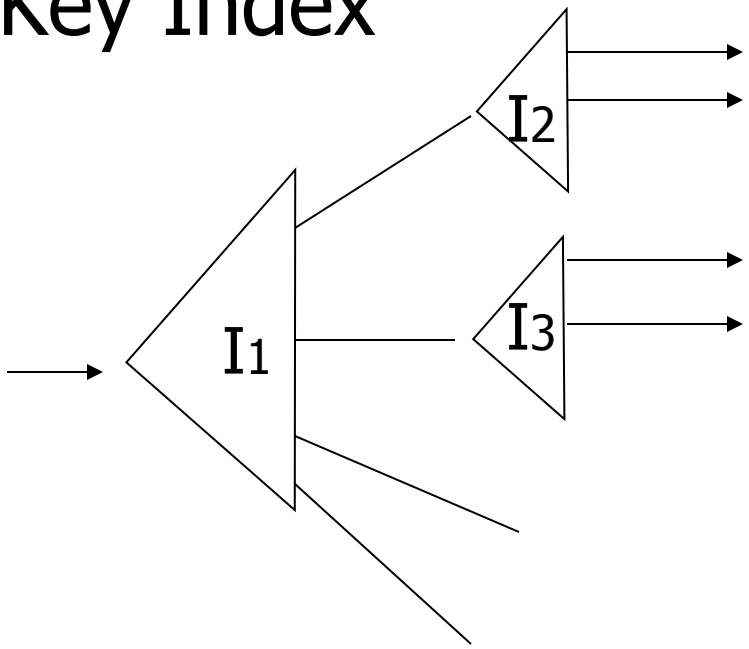
- Use 2 Indexes; Manipulate Pointers



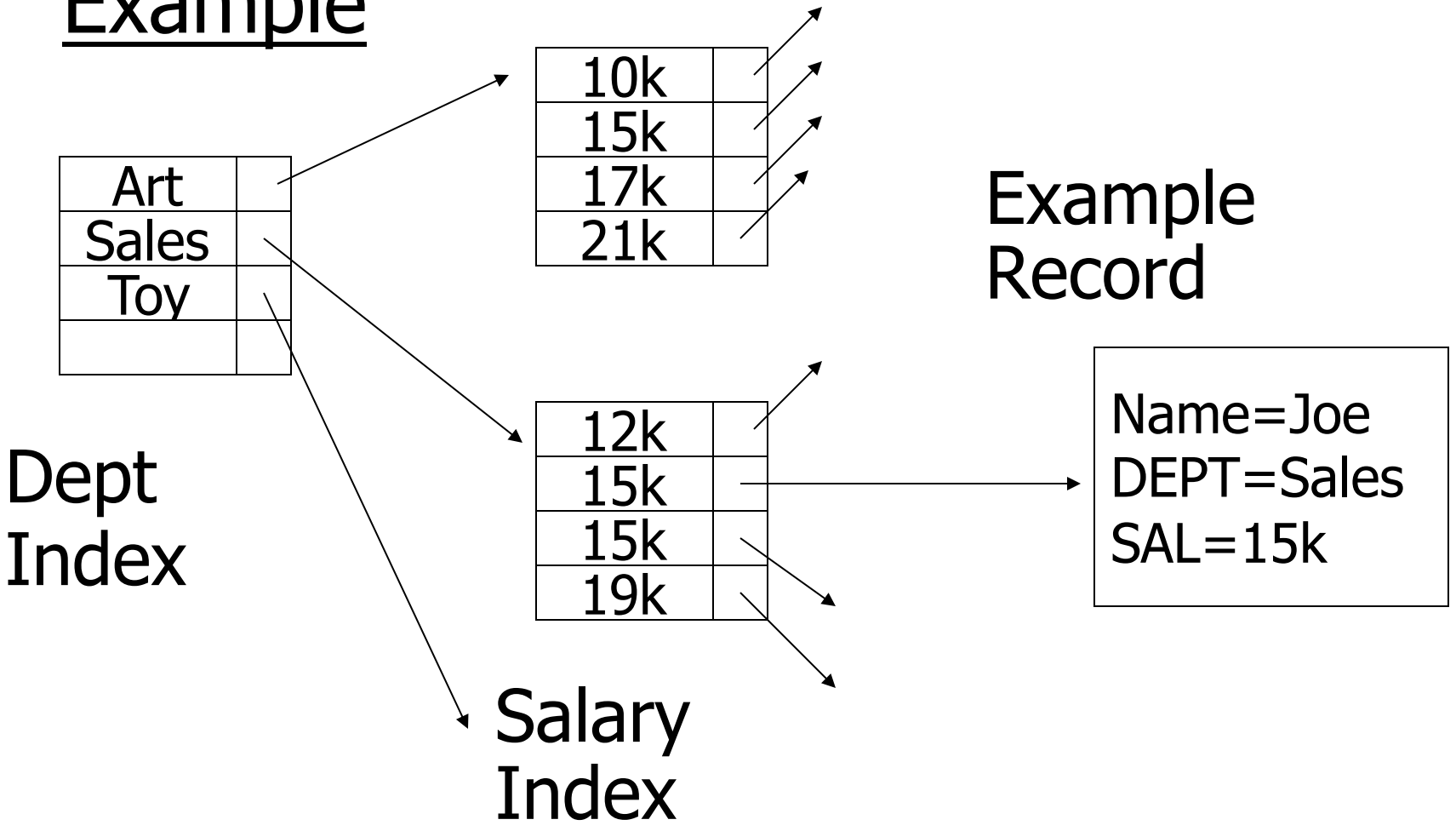
Strategy III:

- Multiple Key Index

One idea:



Example

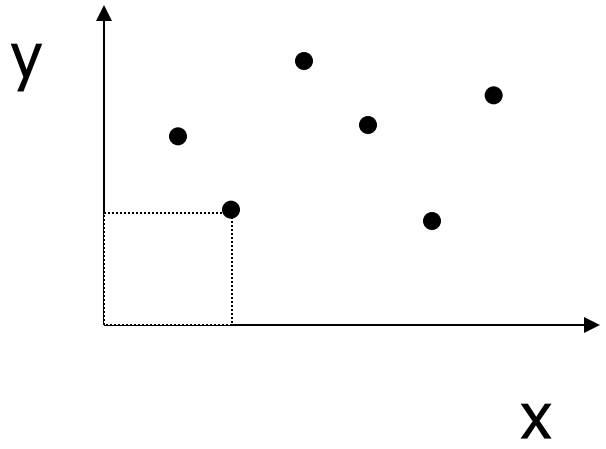

Art
Sales
Toy
10k
15k
17k
21k
12k
15k
15k
19k
Name=Joe
DEPT=Sales
SAL=15k
Example Record
Dept Index
Salary Index

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

Interesting application:

- Geographic Data



DATA:

<X_1,Y_1, Attributes>

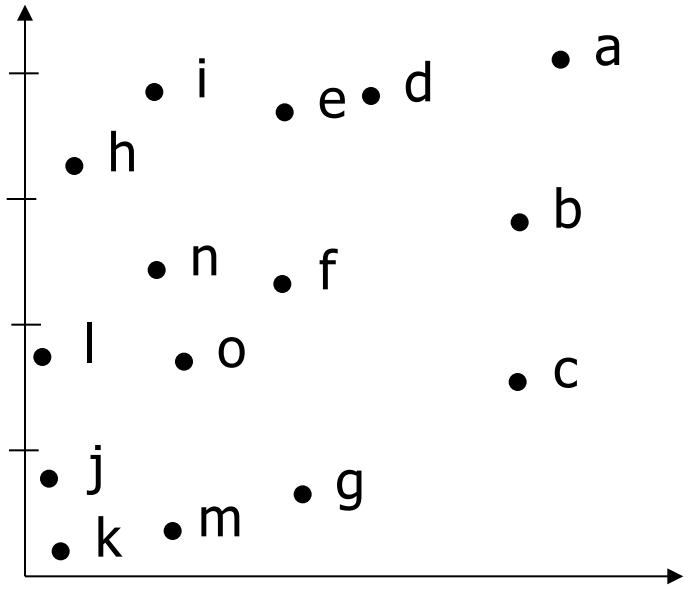
<X_2,Y_2, Attributes>

⋮

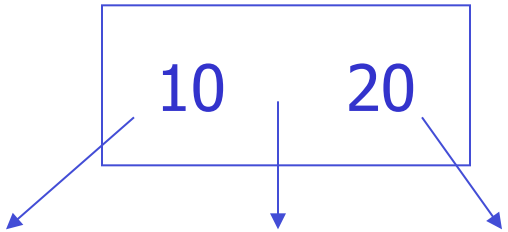
Queries:

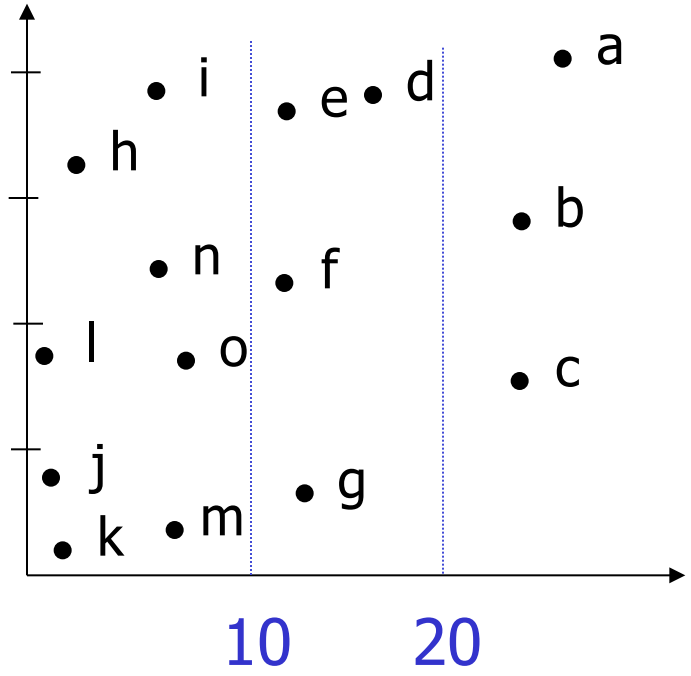
- What city is at <Xi,Yi>?
- What is within 5 miles from <Xi,Yi>?
- Which is closest point to <Xi,Yi>?

Example

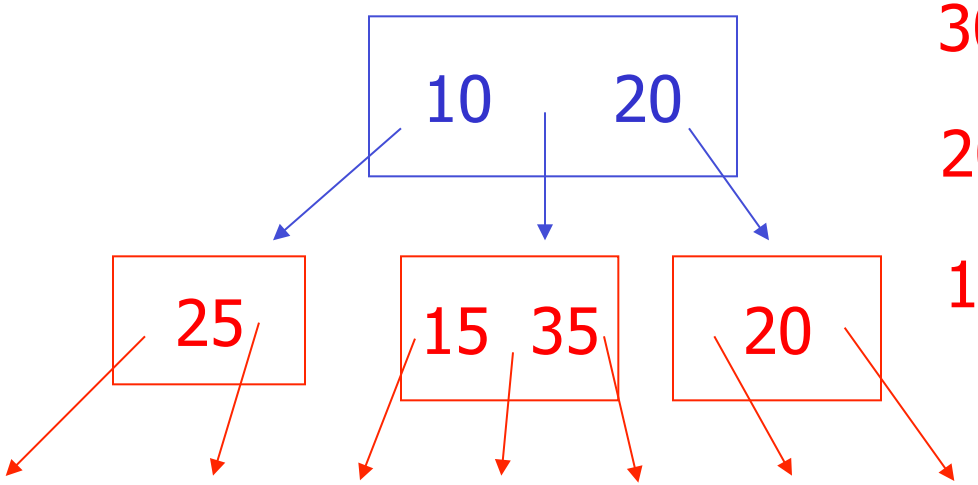


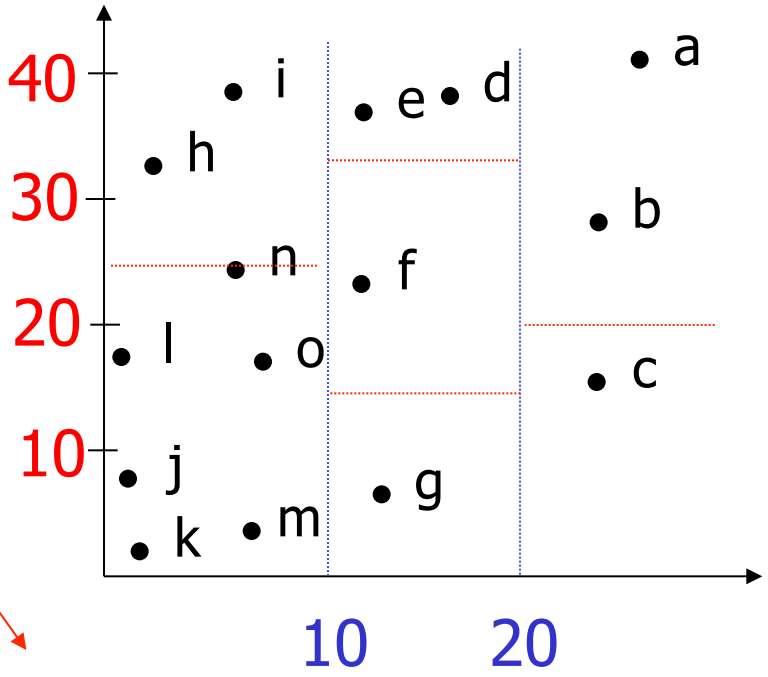
Example



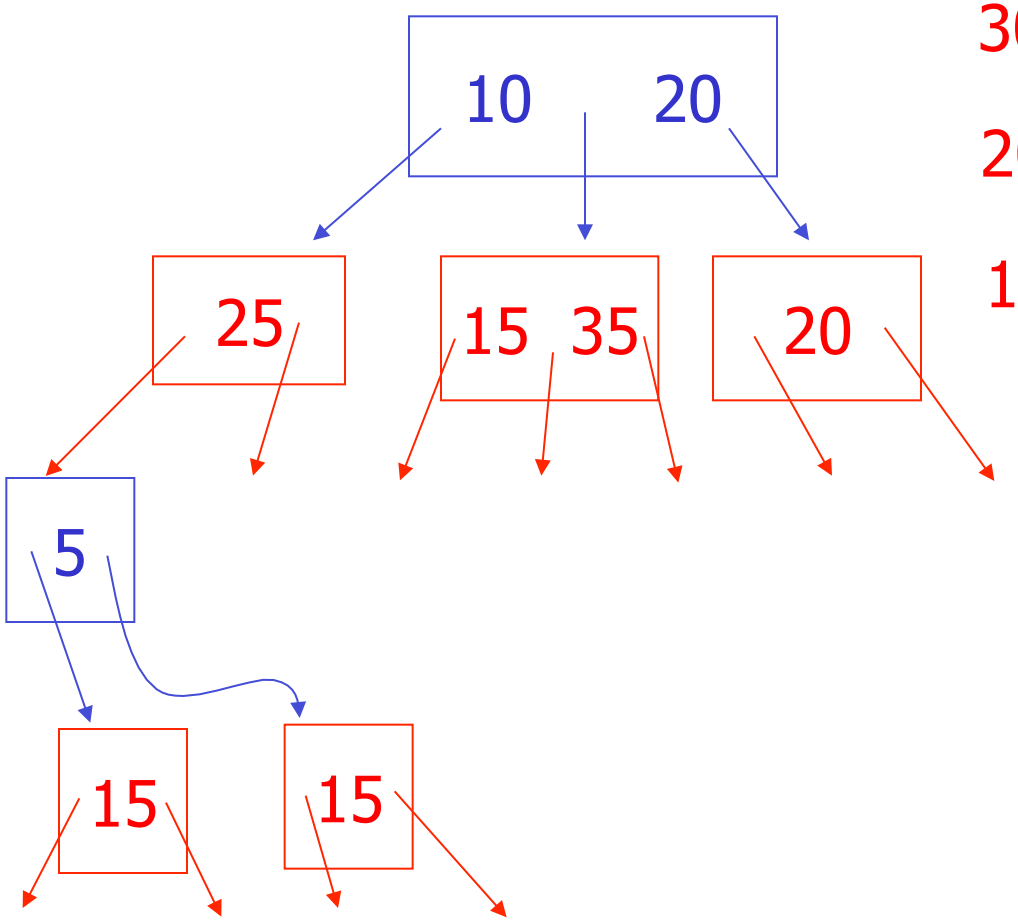


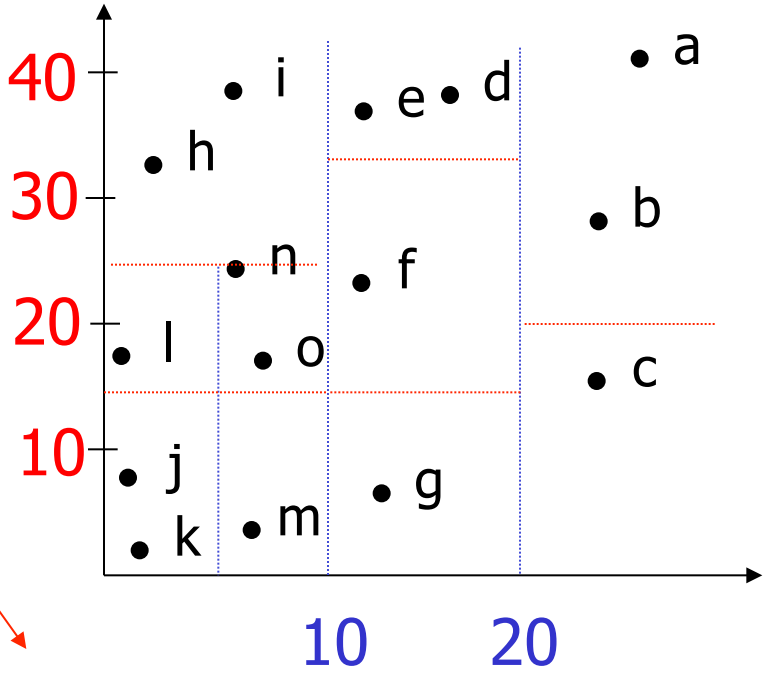
Example



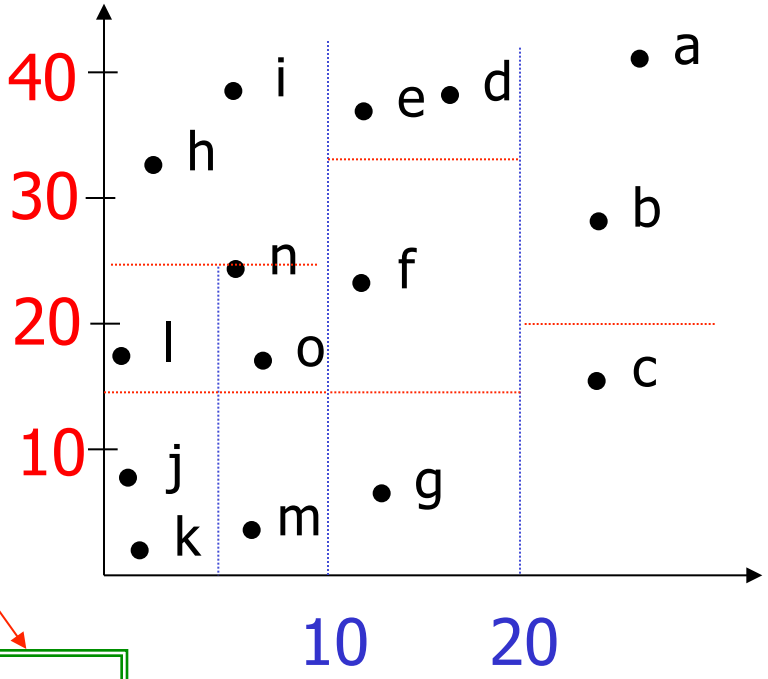


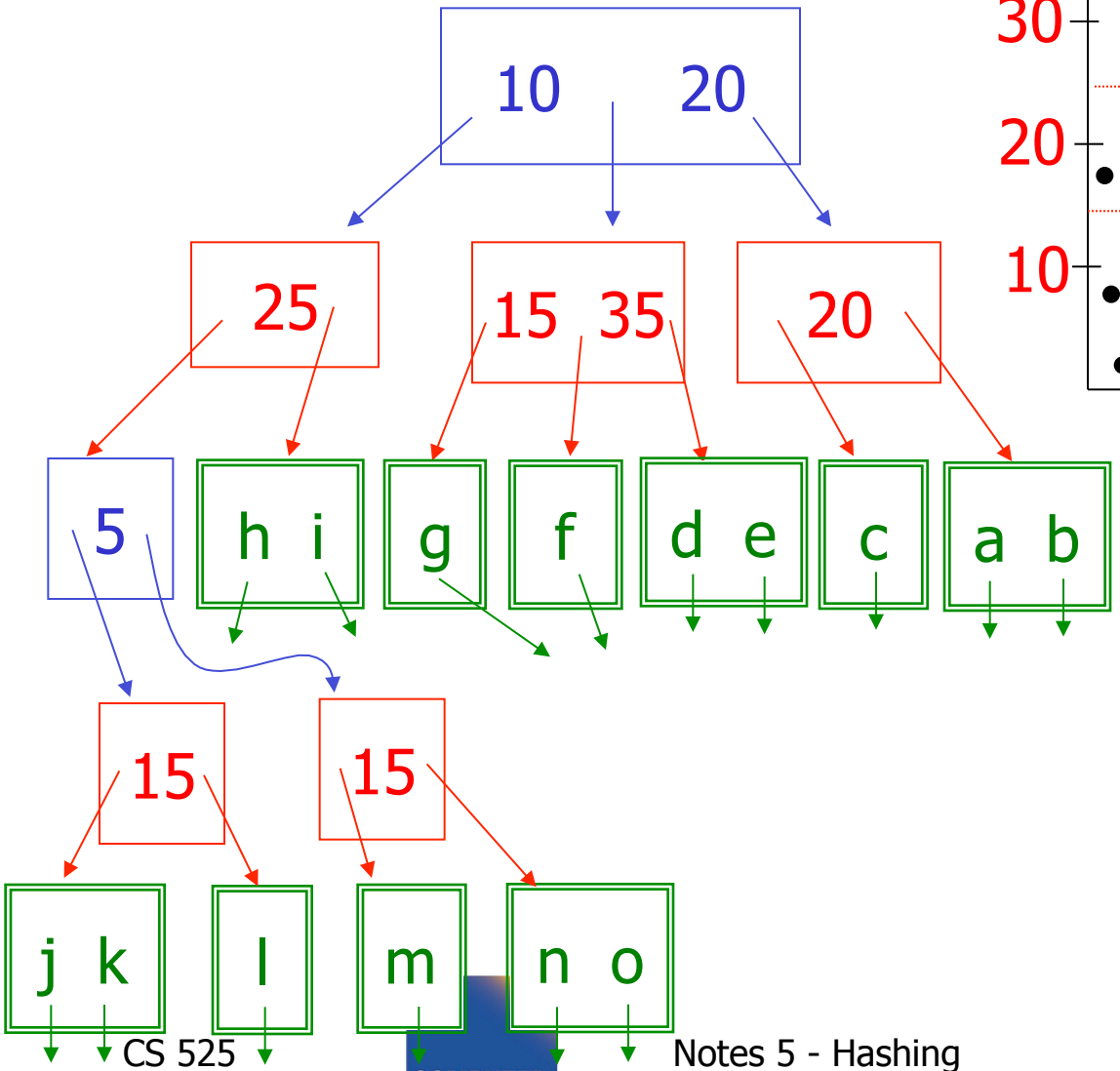
Example



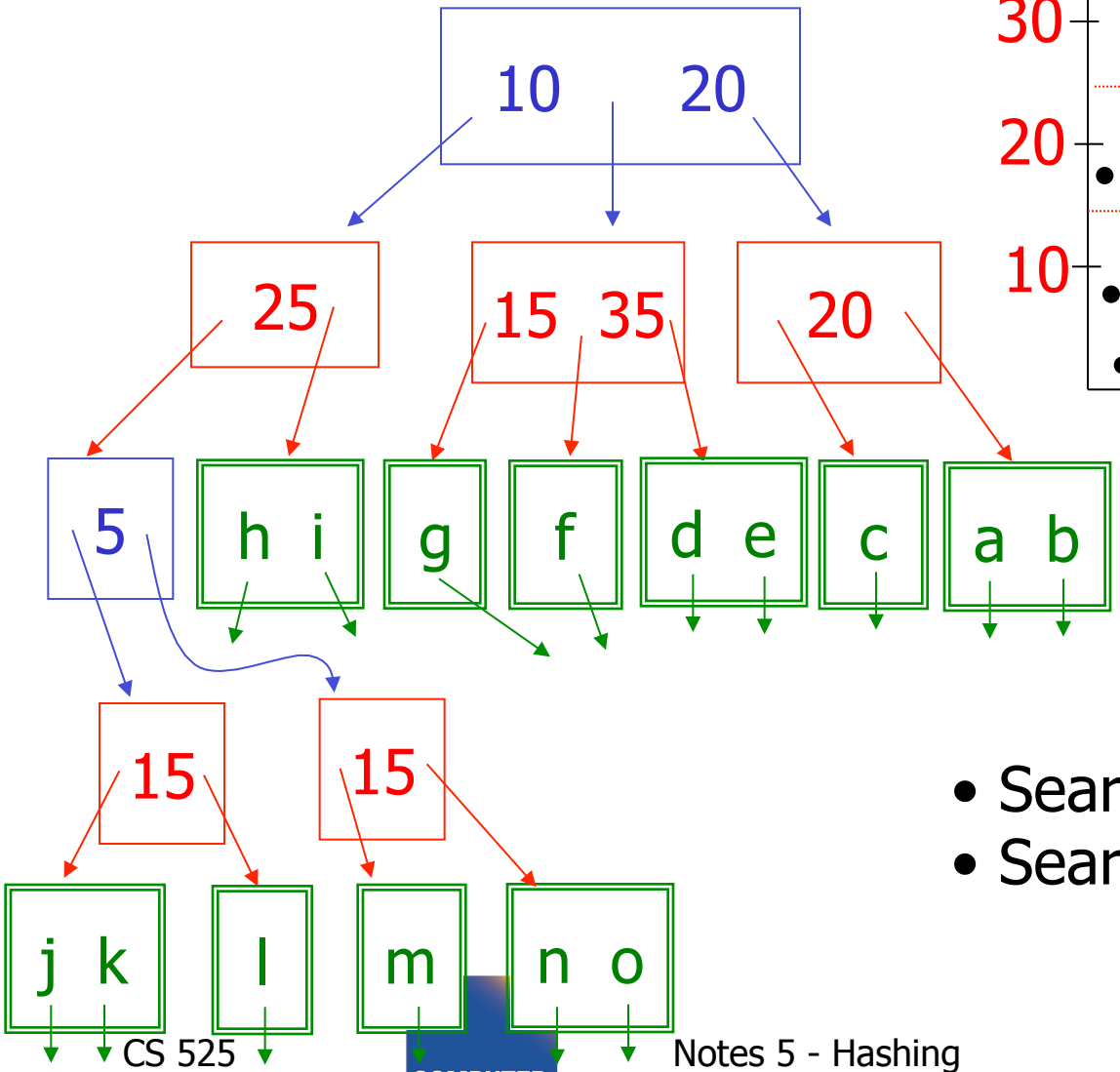


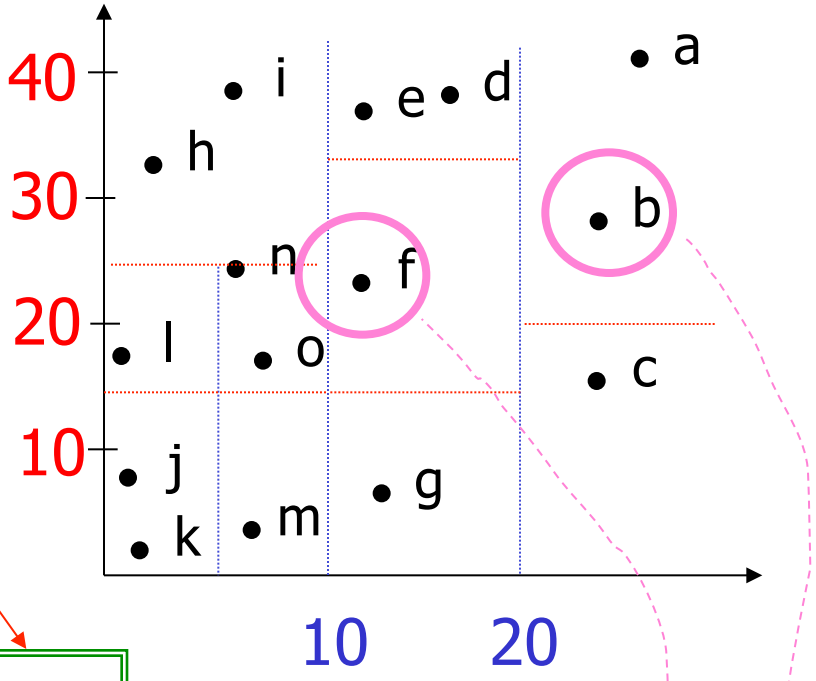
Example


40
30
20
10
i
e
d
a
h
b
n
f
l
o
c
j
g
k
m
10
20


10 20
25
15 35
20
5
h i
g
f
d e
c
a b
15
15
j k
l
m
n o

Example





- Search points near f
- Search points near b

COMPUTER SCIENCE

IIT College of Science and Letters
ILLINOIS INSTITUTE OF TECHNOLOGY

Queries

- Find points with Yi > 20
- Find points with Xi < 5
- Find points “close” to i = <12,38>
- Find points “close” to b = <7,24>

Next

- Even more index structures ☺