CSEN 604: Databases II

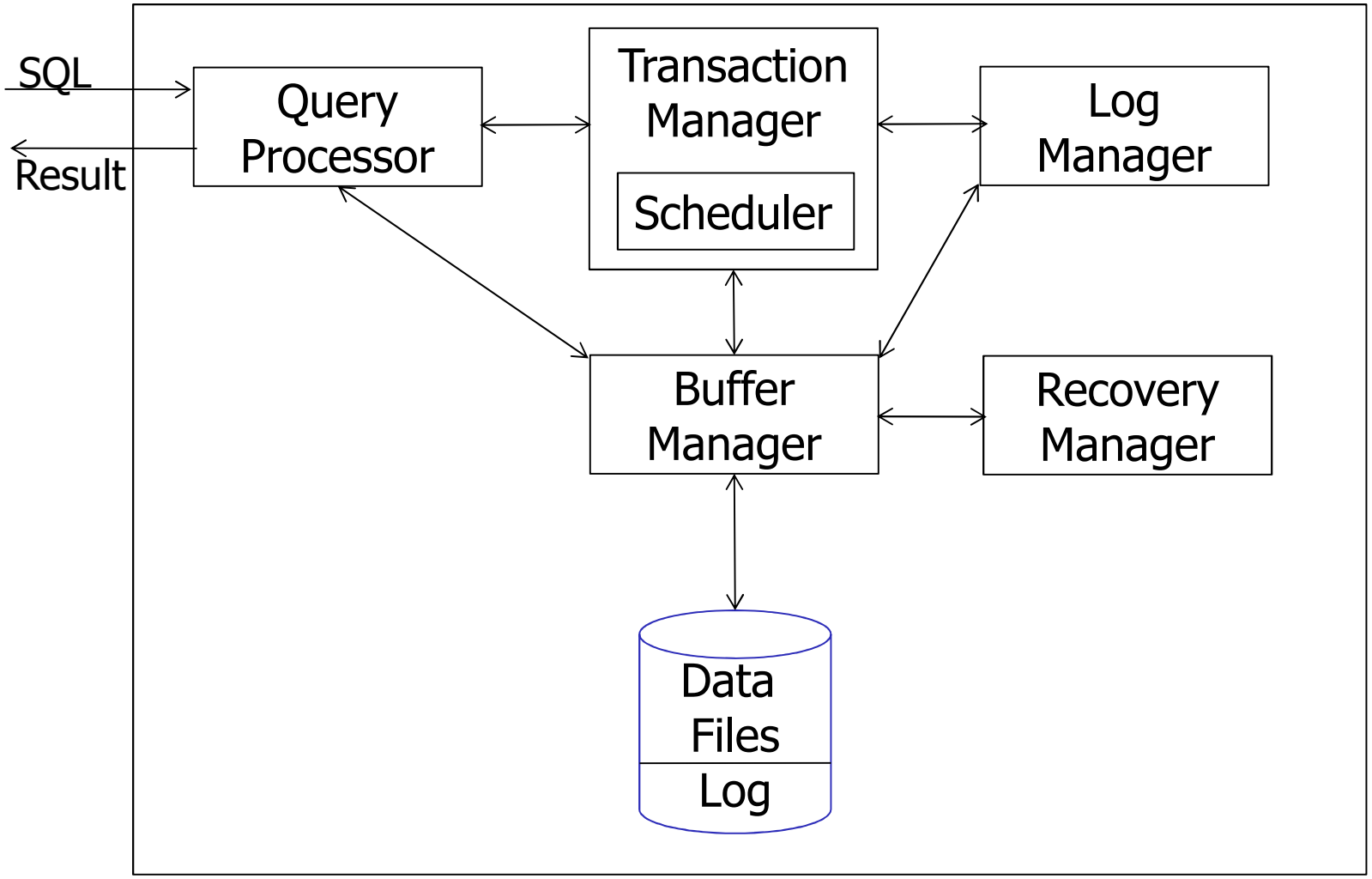
Lecture 3

Dr. Wael Abouelsaadat
wael.abouelsaadat@guc.edu.eg
Office: C7.208

Office Hour is 4th slot Saturday or you can email for appointment

Acknowledgment: these slides are based on Prof. Garcia-Molina & Prof. Ullman slides accompanying the book: *Database Systems; the Complete Book*

DBMS Architecture



Topics

- B+ Tree
- Hashing

- B+ Tree
  - A data structure used to build an index
  - Most nodes are on hard-disk → a storage oriented data structures
  - Just like any tree: basic building blocks;
    - Node (link in CS3)
    - Node consists of data + references to children nodes.
  - Self-balanced data structure → $O(\log n)$

Deletion from B+tree

(a) Simple case - no example

(b) Coalesce with neighbor (sibling)

(c) Re-distribute keys

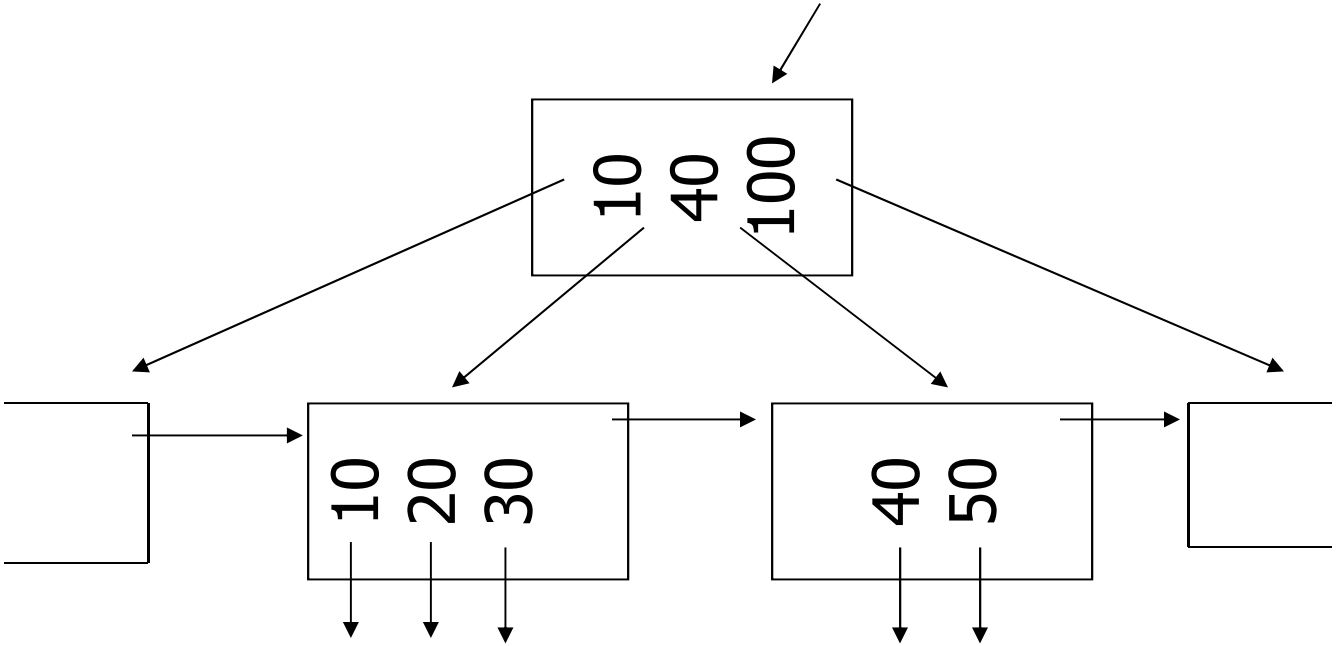
(d) Cases (b) or (c) at non-leaf

(b) Coalesce with sibling

– Delete 50

n=4

if n = 4, then **key** count
Non-leaf: max: 4, min: 2
Leaf: max: 4, min: 2

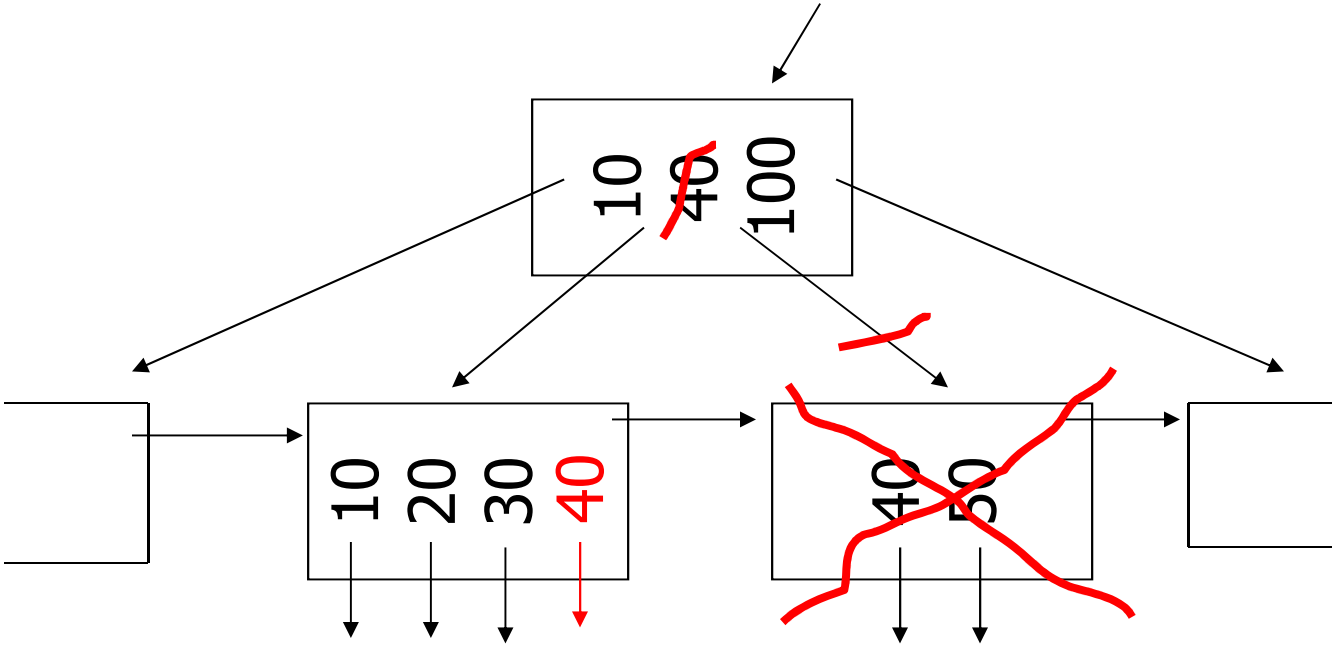


(b) Coalesce with sibling

– Delete 50

n=4

if n = 4, then **key** count
Non-leaf: max: 4, min: 2
Leaf: max: 4, min: 2

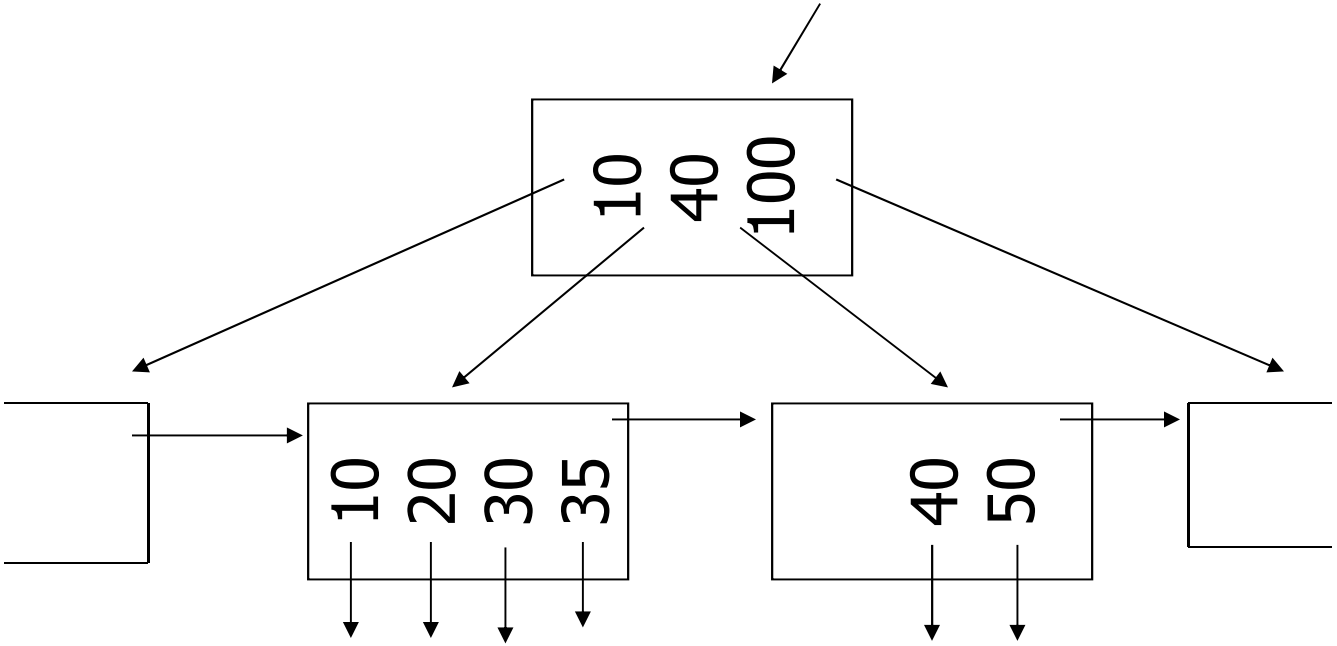


(c) Redistribute keys

– Delete 50

n=4

if n = 4, then **key** count
Non-leaf: max: 4, min: 2
Leaf: max: 4, min: 2

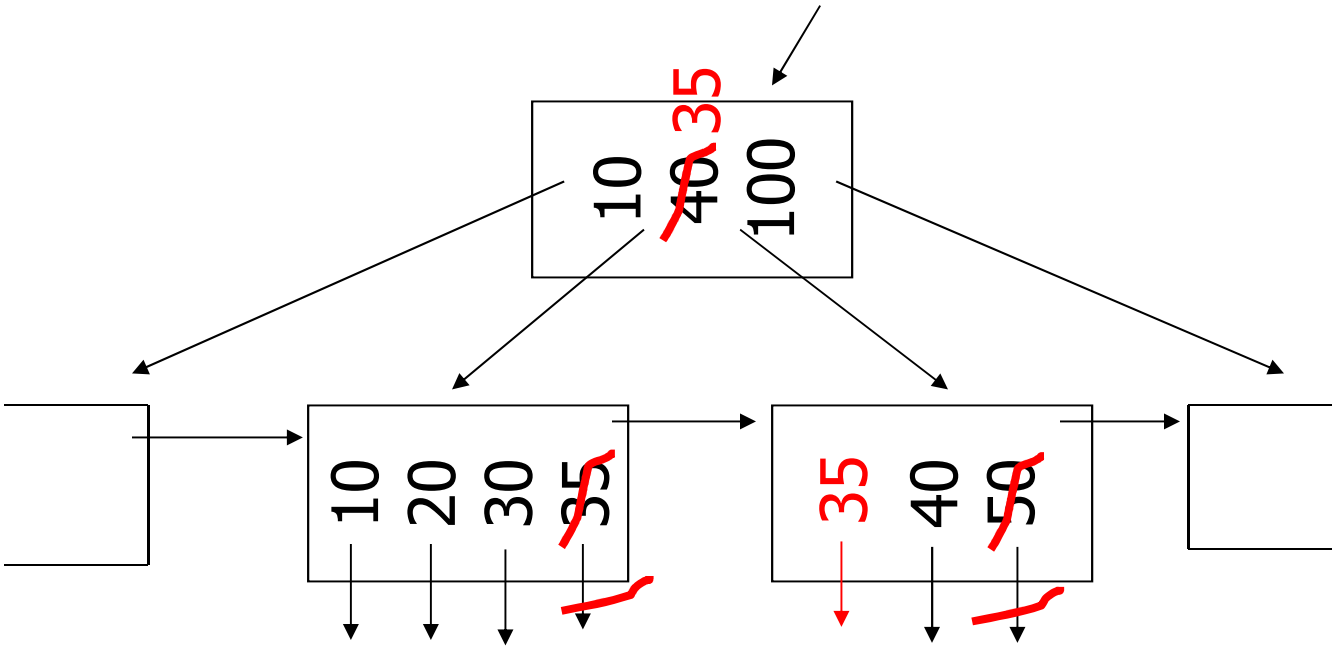


(c) Redistribute keys

– Delete 50

n=4

if n = 4, then **key** count
Non-leaf: max: 4, min: 2
Leaf: max: 4, min: 2

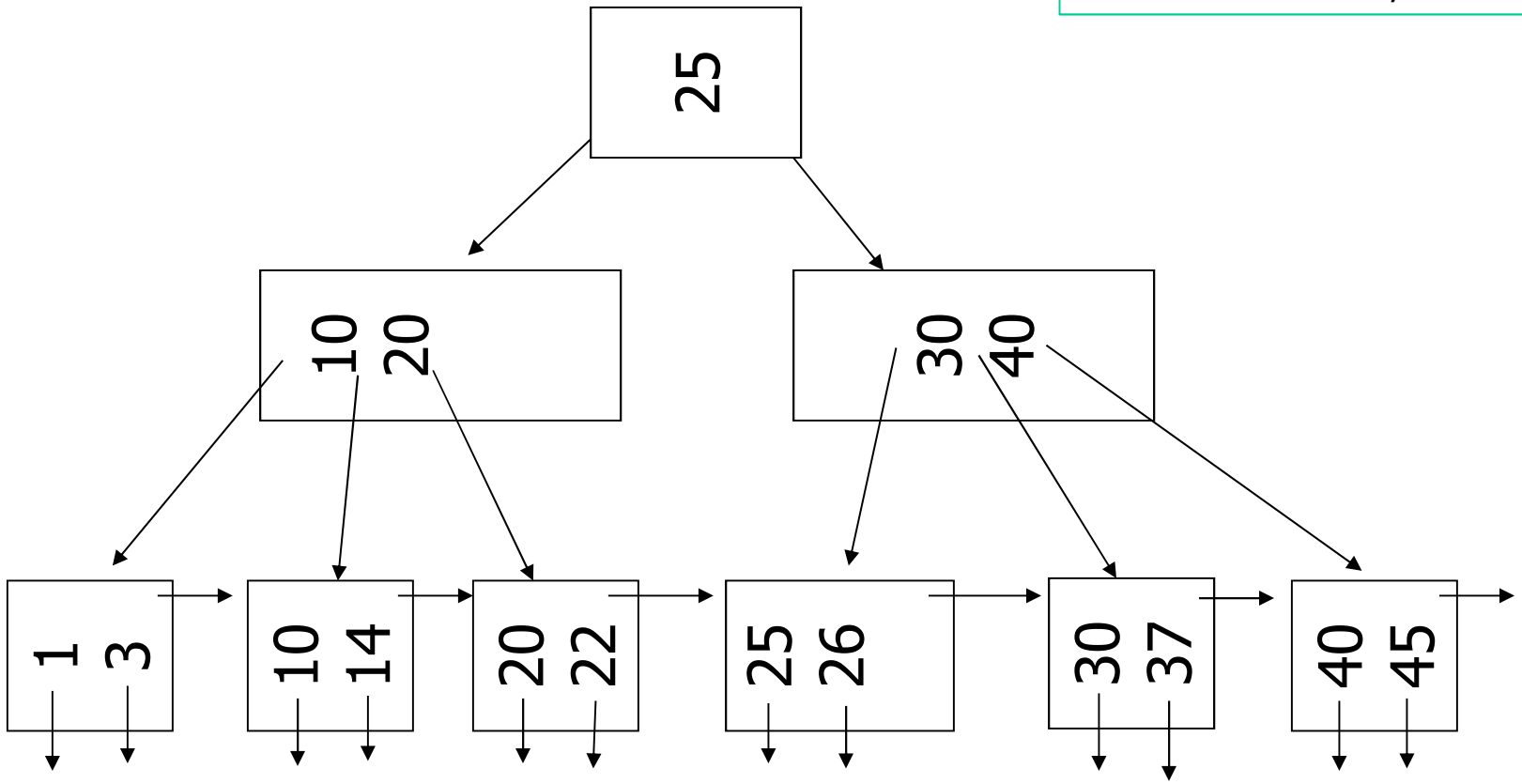


(d) Non-leaf coalese

– Delete 37

n=4

if n = 4, then **key** count
Non-leaf: max: 4, min: 2
Leaf: max: 4, min: 2

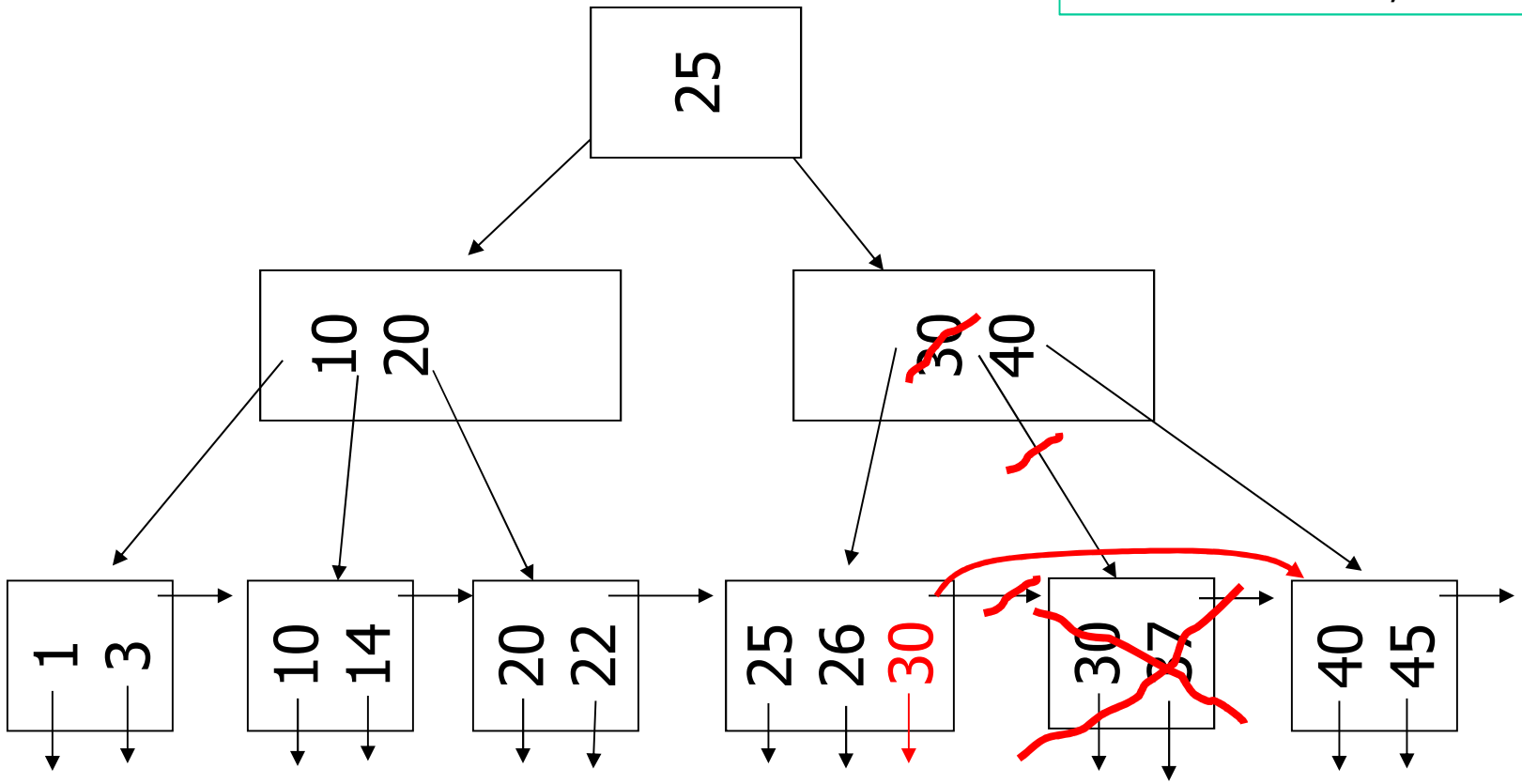


(d) Non-leaf coalese

– Delete 37

n=4

if n = 4, then **key** count
Non-leaf: max: 4, min: 2
Leaf: max: 4, min: 2

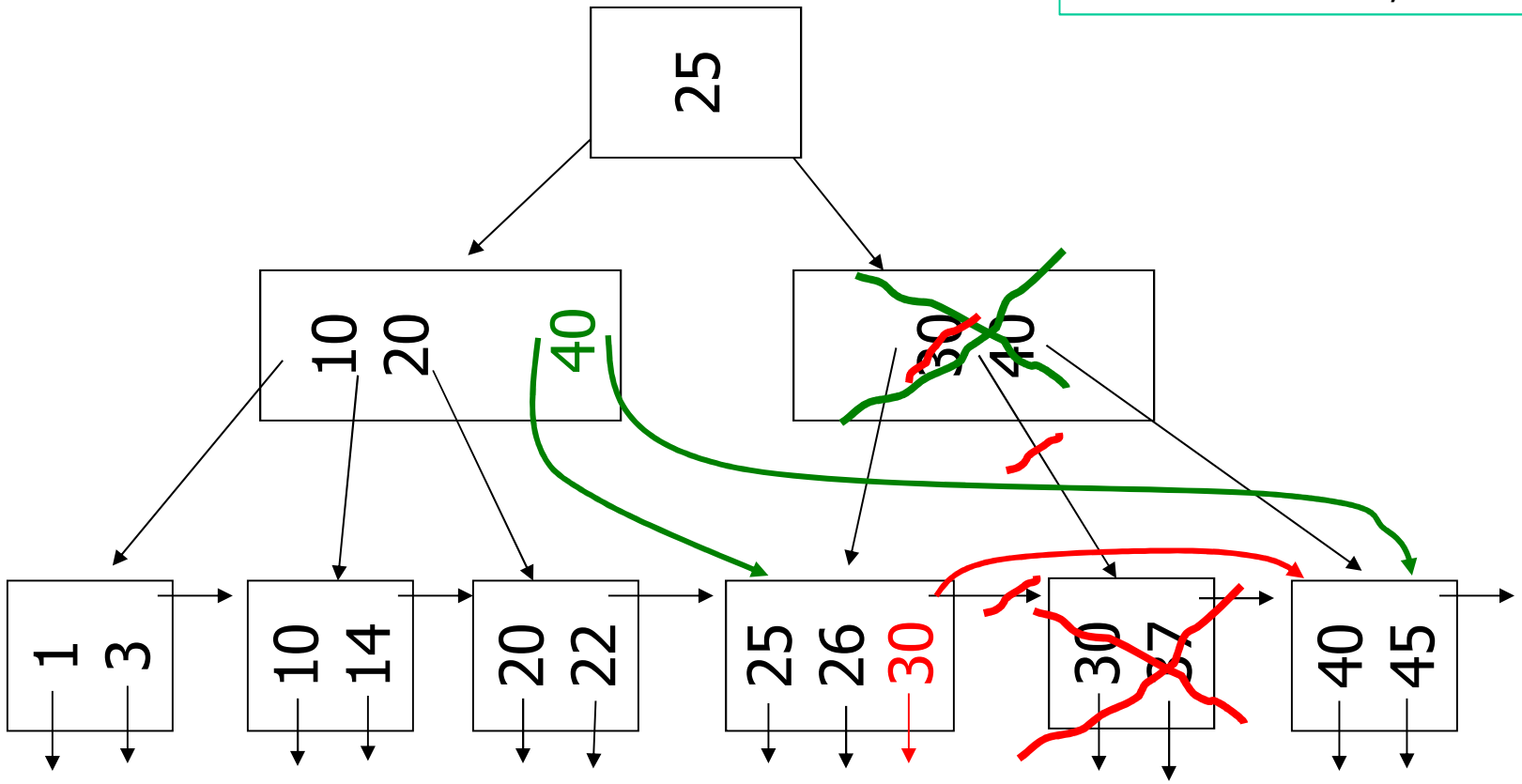


(d) Non-leaf coalese

– Delete 37

n=4

if n = 4, then **key** count
Non-leaf: max: 4, min: 2
Leaf: max: 4, min: 2

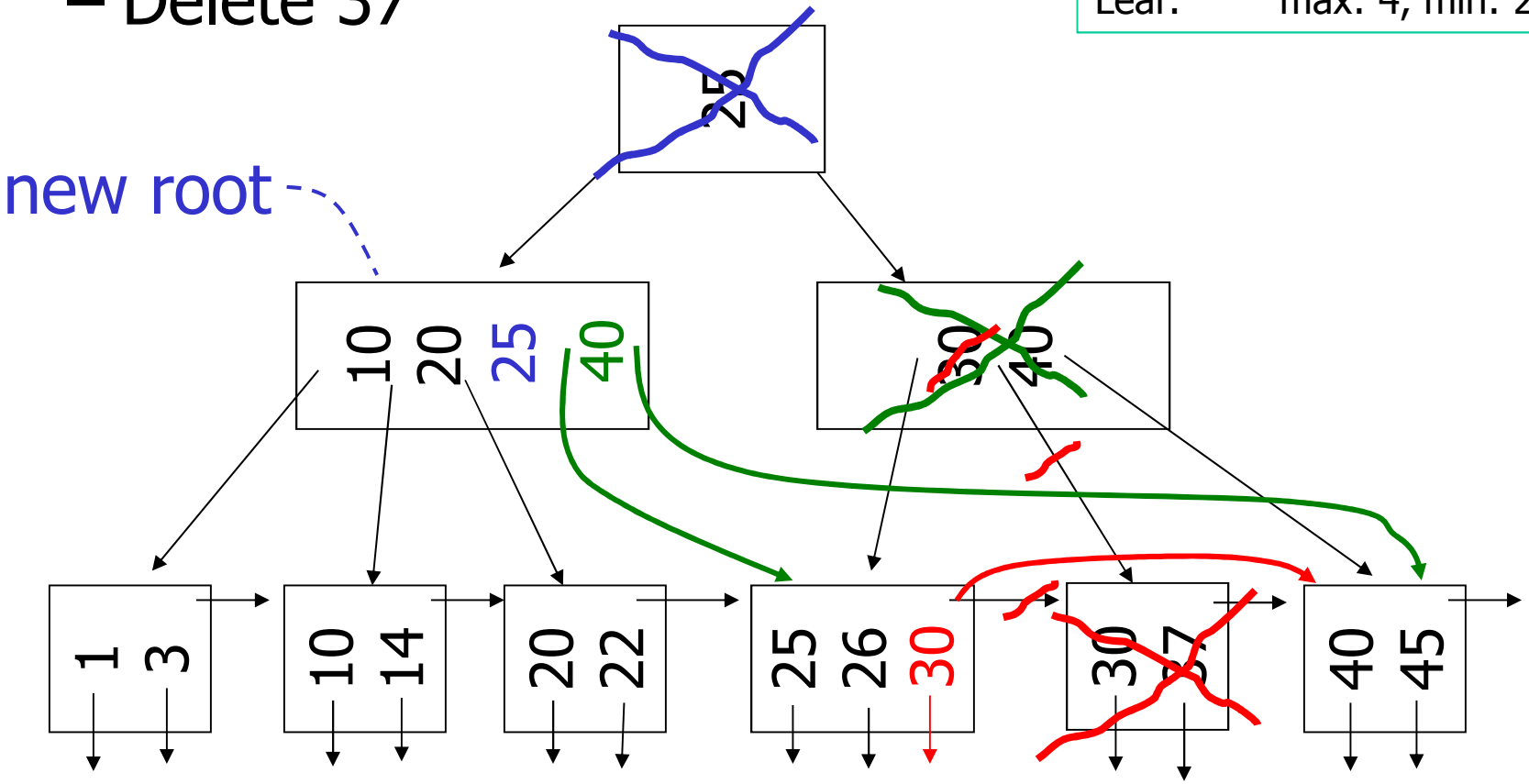


(d) Non-leaf coalese

– Delete 37

n=4

if n = 4, then **key** count
Non-leaf: max: 4, min: 2
Leaf: max: 4, min: 2



When to create a B+ tree index ?

- You want to support efficient search on a single column

+

- A majority of received queries involve a range search (min, max, between, …)

Creating an Index using B+ Tree

- PostgreSQL
  - SQL: create index index_name on table-name(column-name) using B_tree;
  - http://www.postgresql.org/docs/8.2/static/sql-createindex.html

Index vs. key vs. Clustering

- Key: logical concept
- Index: a secondary storage data structure
- Clustering: disk/page organizational issue
  - According to which column the data is stored in pages

Clustered Index vs. Non-clustered Index

- Clustered Index
  - Clustering defines the page content interms of order tuples. Index and relation match in ordering
  - Only one clustered index can be created on a given database table.
  - Clustered indices can greatly increase overall speed of retrieval, but usually only where the data is accessed sequentially in the same or reverse order of the clustered index, or when a range of items is selected.

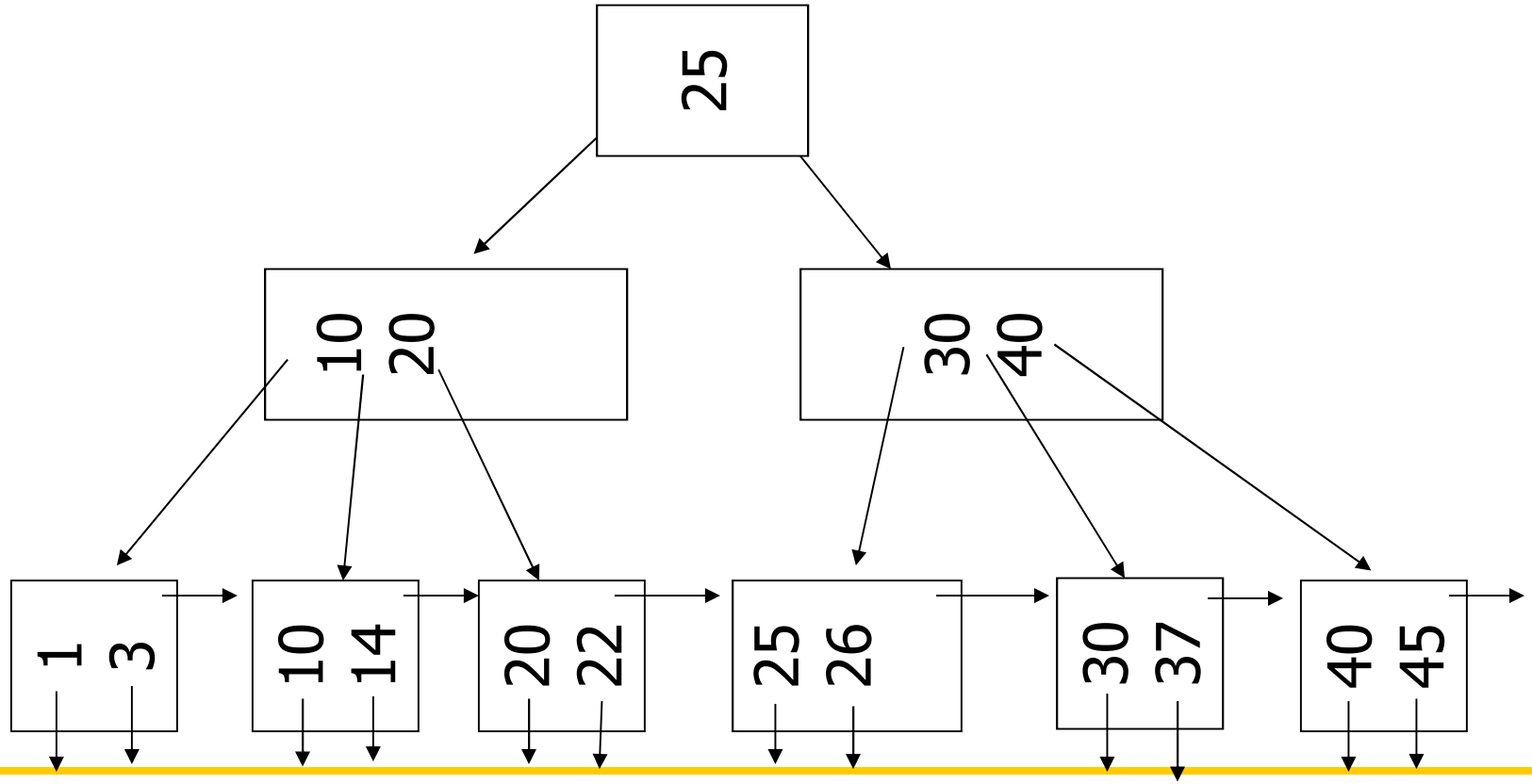
Clustered Index vs. Non-clustered Index

- Non-clustered Index
  - logical ordering is specified by the index only.
  - The physical order of the rows is not the same as the index order.
  - The indexed columns are typically non-primary key columns (used in JOIN, WHERE, and ORDER BY clauses)
  - There can be more than one non-clustered index on a database table.

Clustered Index vs. Non-clustered Index

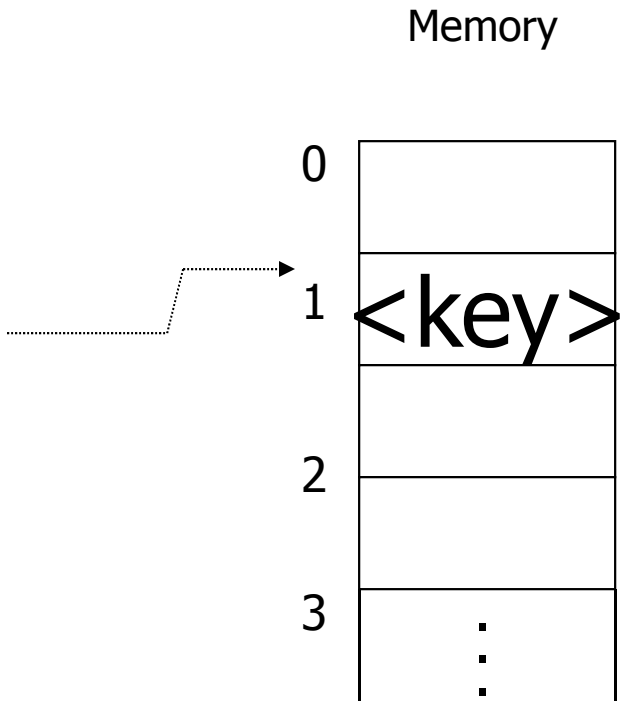
– E.g. SQL Server

```
create table DemonstrationTable (
    TableIdColumn int not null primary key nonclustered );
```

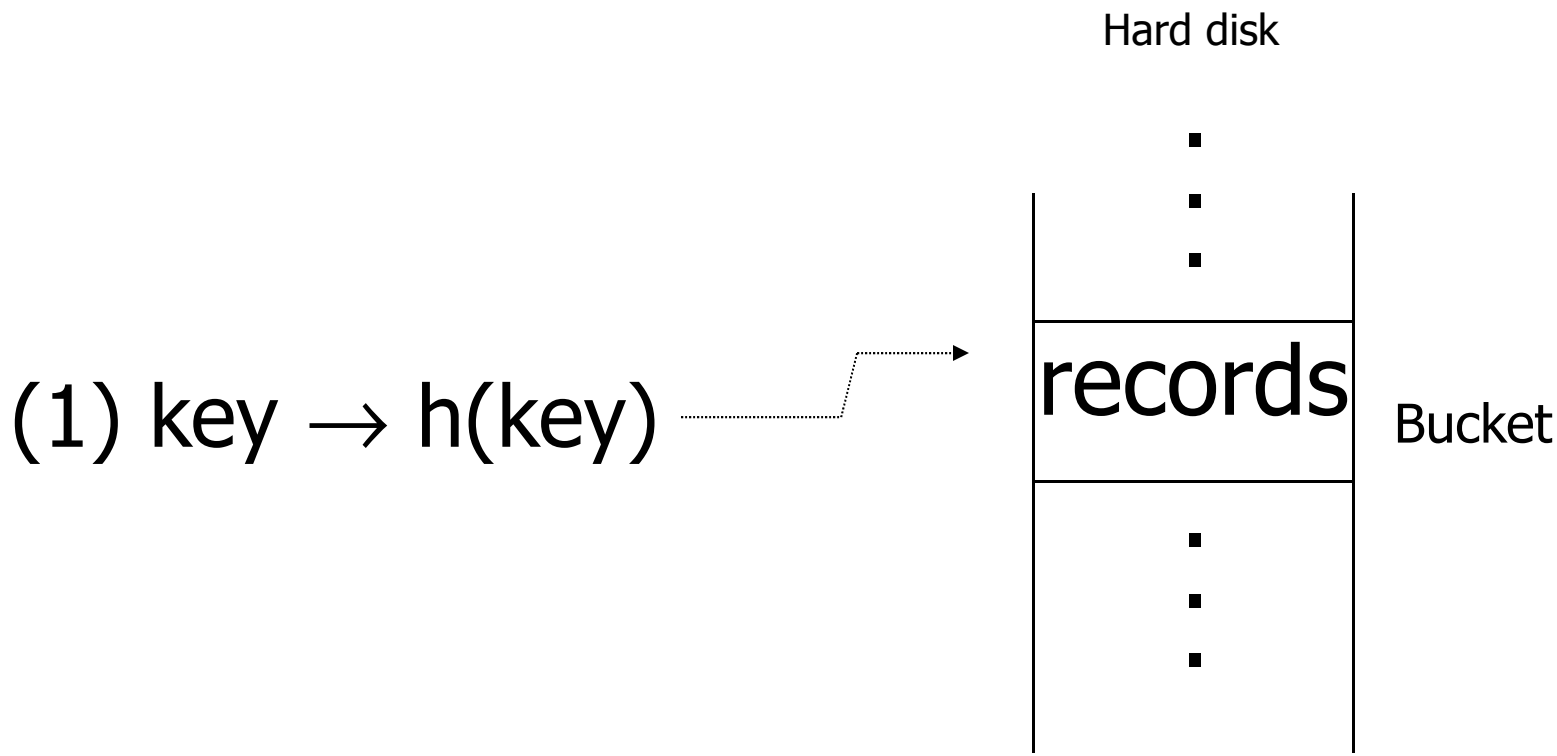


Hashing

key $\rightarrow$ h(key)



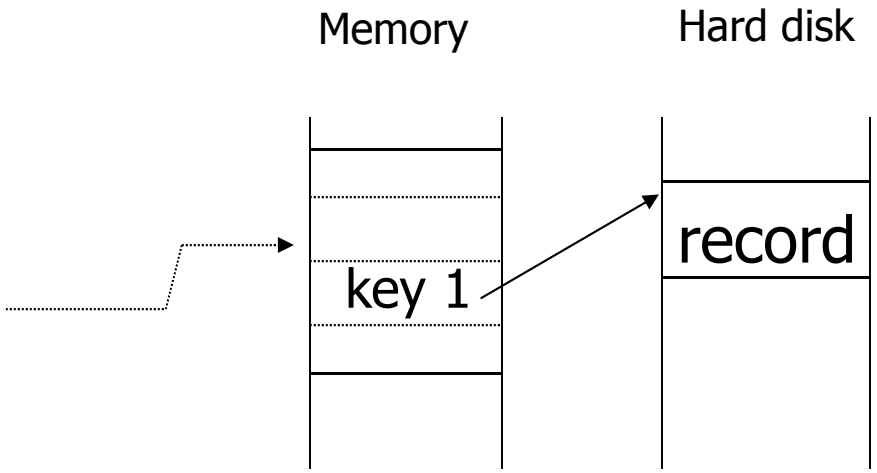
Two alternatives



- Alt (1) for "primary" search key

Two alternatives

(2) key → h(key)



Index

- 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:

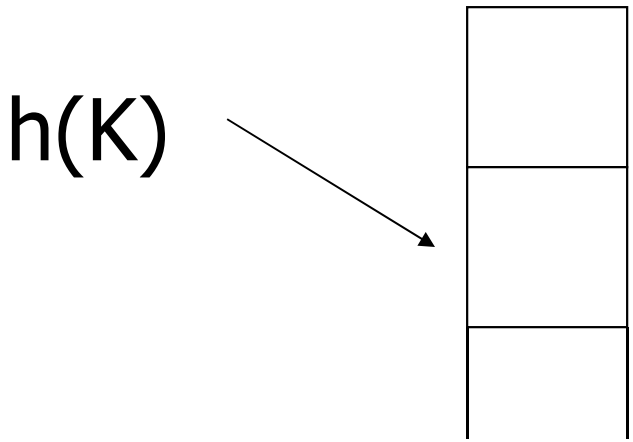
☞ Expected number of hash-value/bucket is the same for all buckets

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

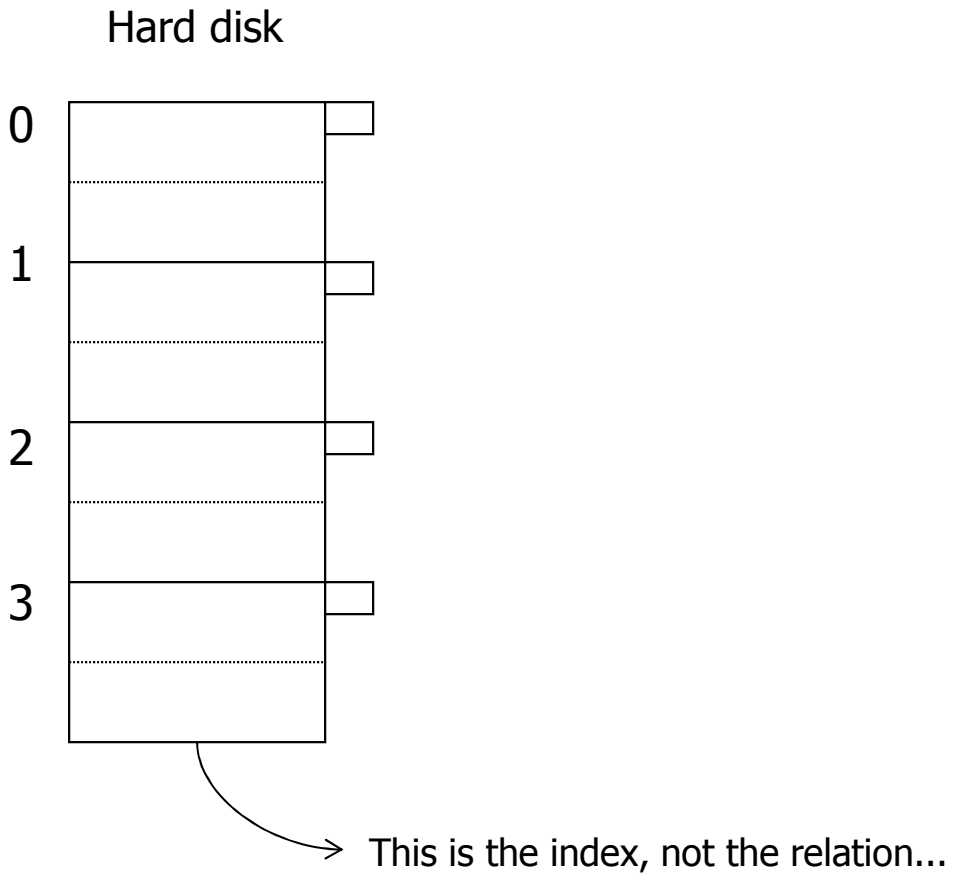
INSERT:

h(a) = 1

h(b) = 2

h(c) = 1

h(d) = 0



EXAMPLE 2 records/bucket

INSERT:

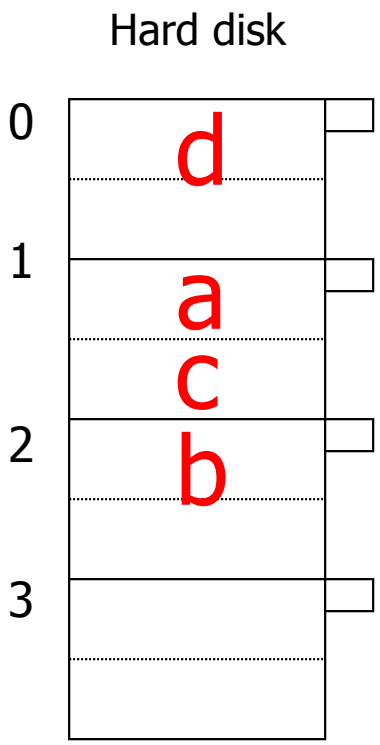
h(a) = 1

h(b) = 2

h(c) = 1

h(d) = 0

h(e) = 1



EXAMPLE 2 records/bucket

INSERT:

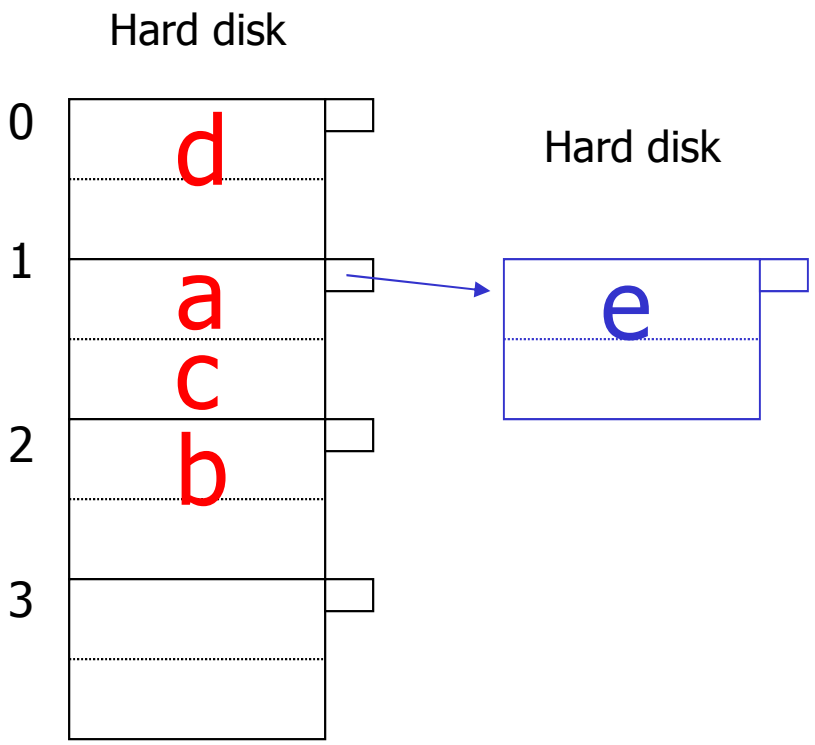
h(a) = 1

h(b) = 2

h(c) = 1

h(d) = 0

h(e) = 1

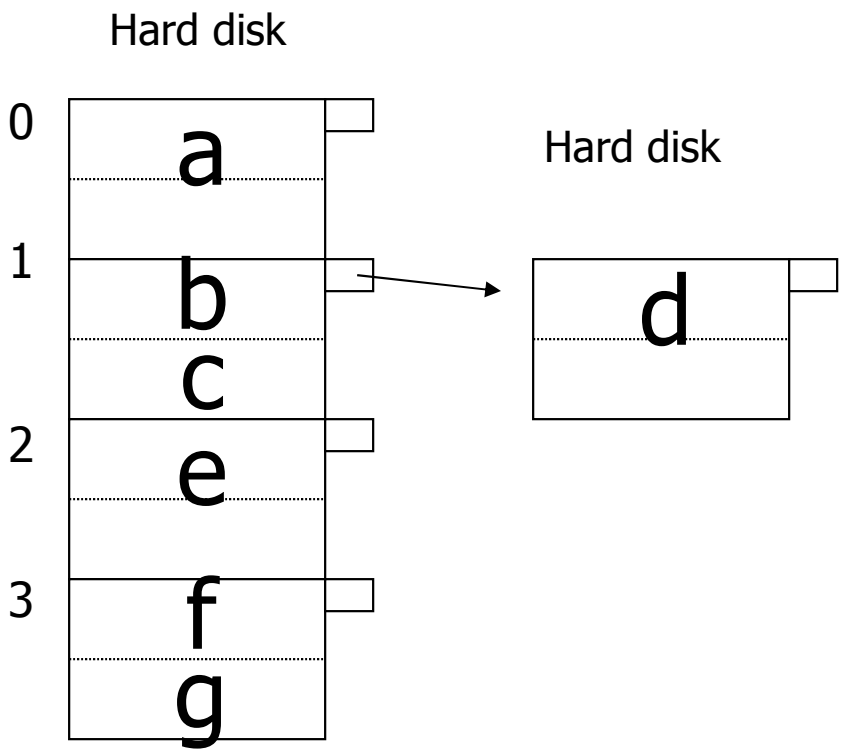


EXAMPLE: deletion

Delete:

e

f



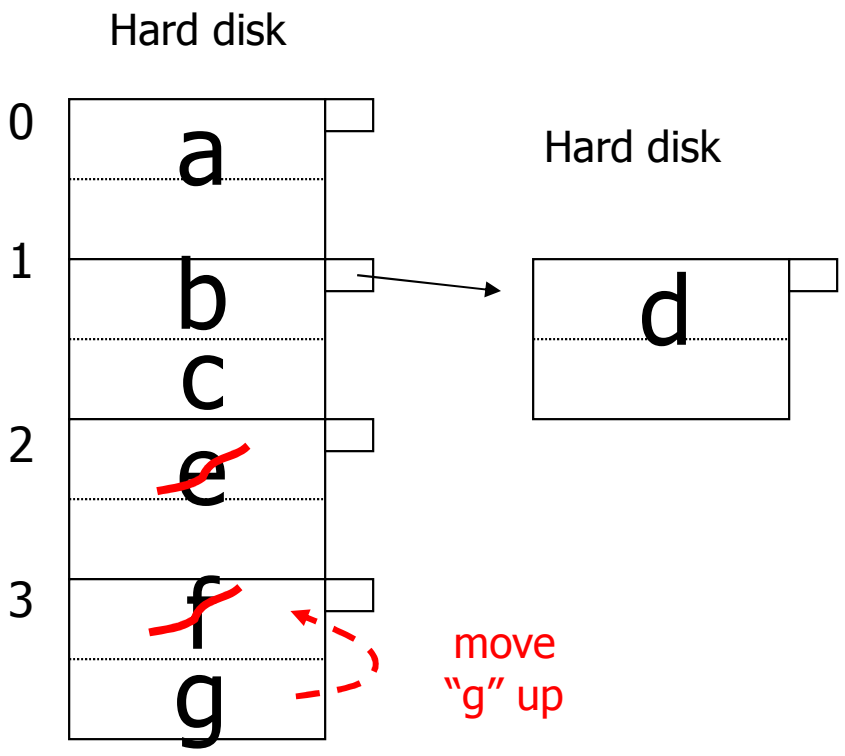
EXAMPLE: deletion

Delete:

e

f

c



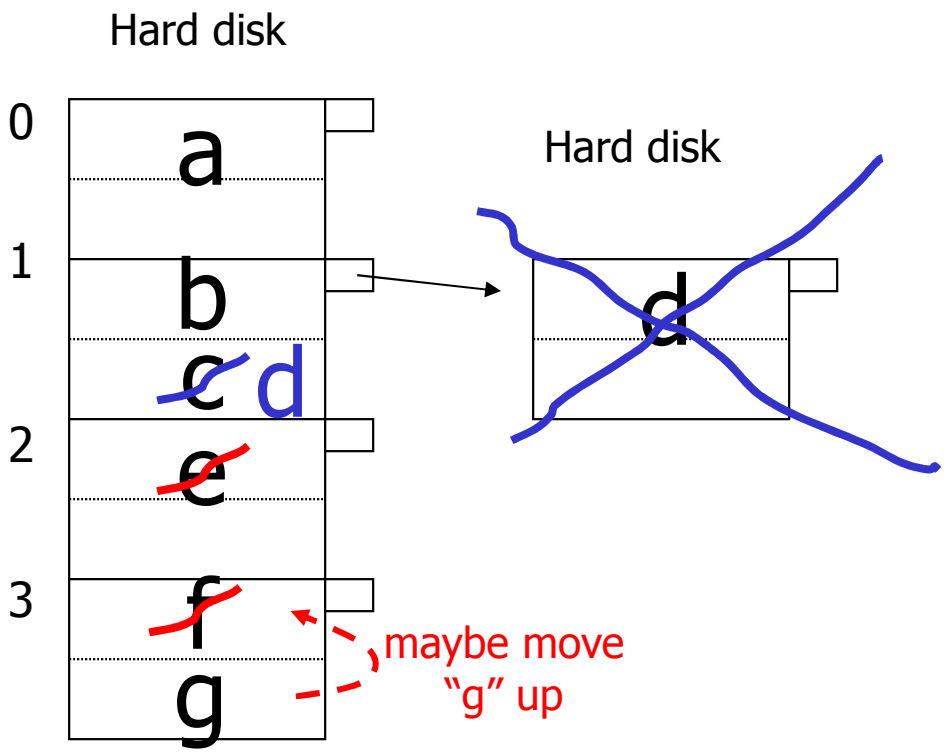
EXAMPLE: deletion

Delete:

e

f

c



Rule of thumb:

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

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

Rule of thumb:

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

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

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

How do we cope with growth?

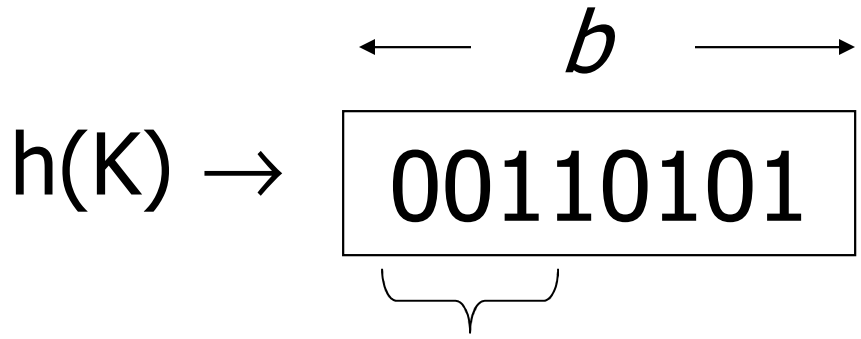
- Reorganizations
  - Very expensive → read all pages and rehash

How do we cope with growth?

- Reorganizations
  - Very expensive → read all pages and rehash
- Dynamic hashing
  - Extensible hashing
  - Linear hashing

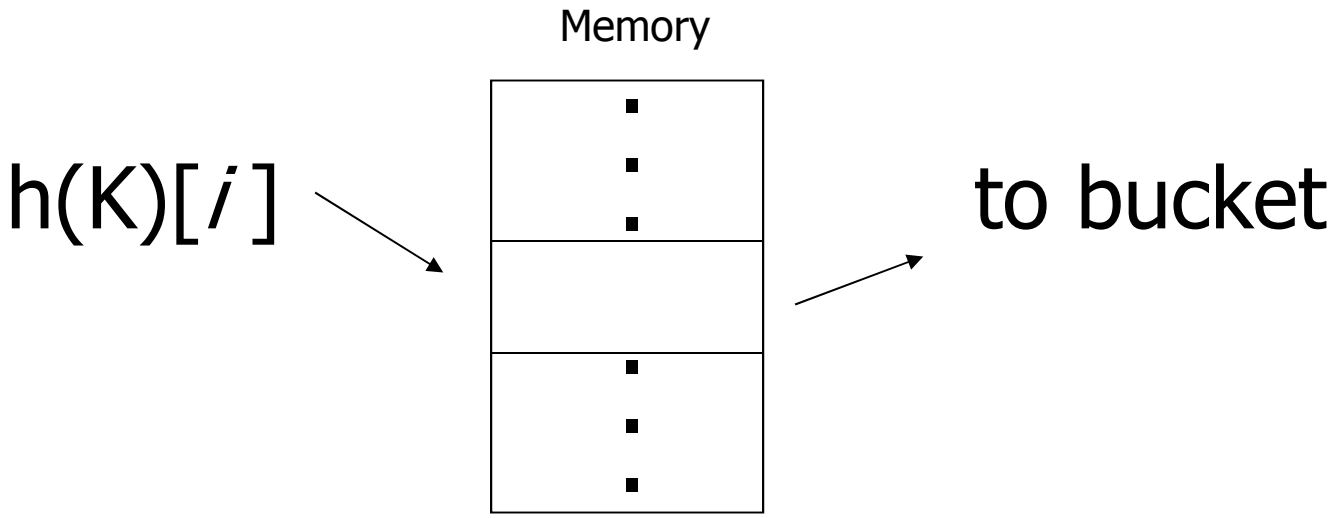
Extensible hashing: two ideas

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

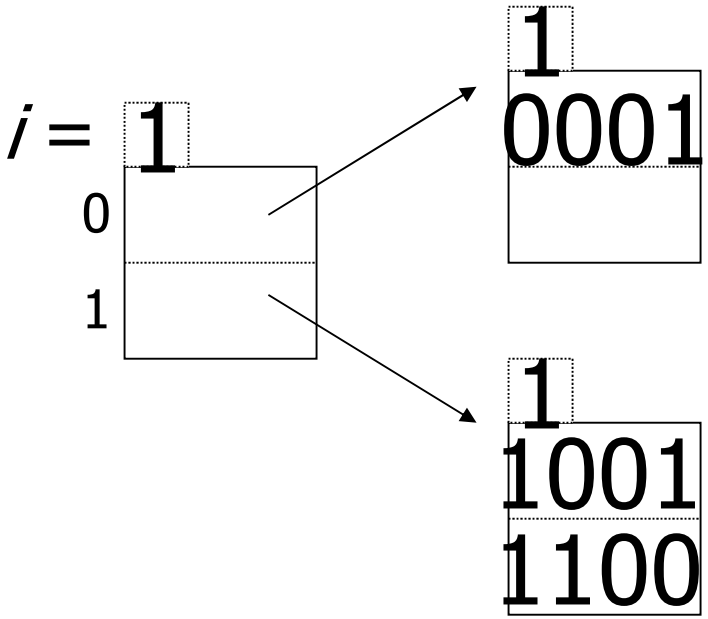


use *i* → grows over time….

(b) Use directory

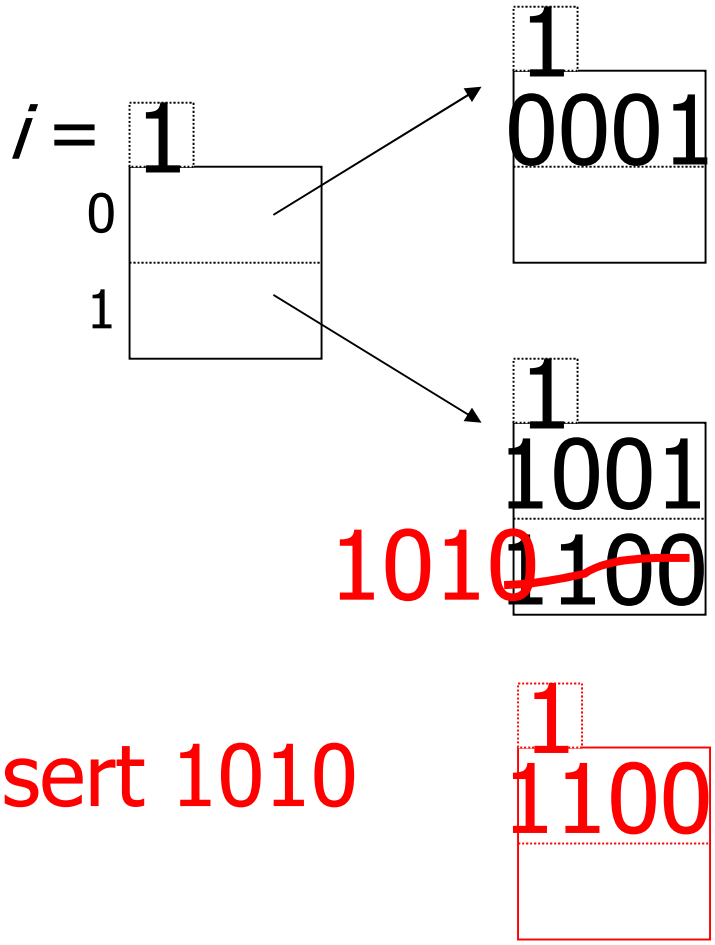


Example: h(k) is 4 bits; 2 keys/bucket

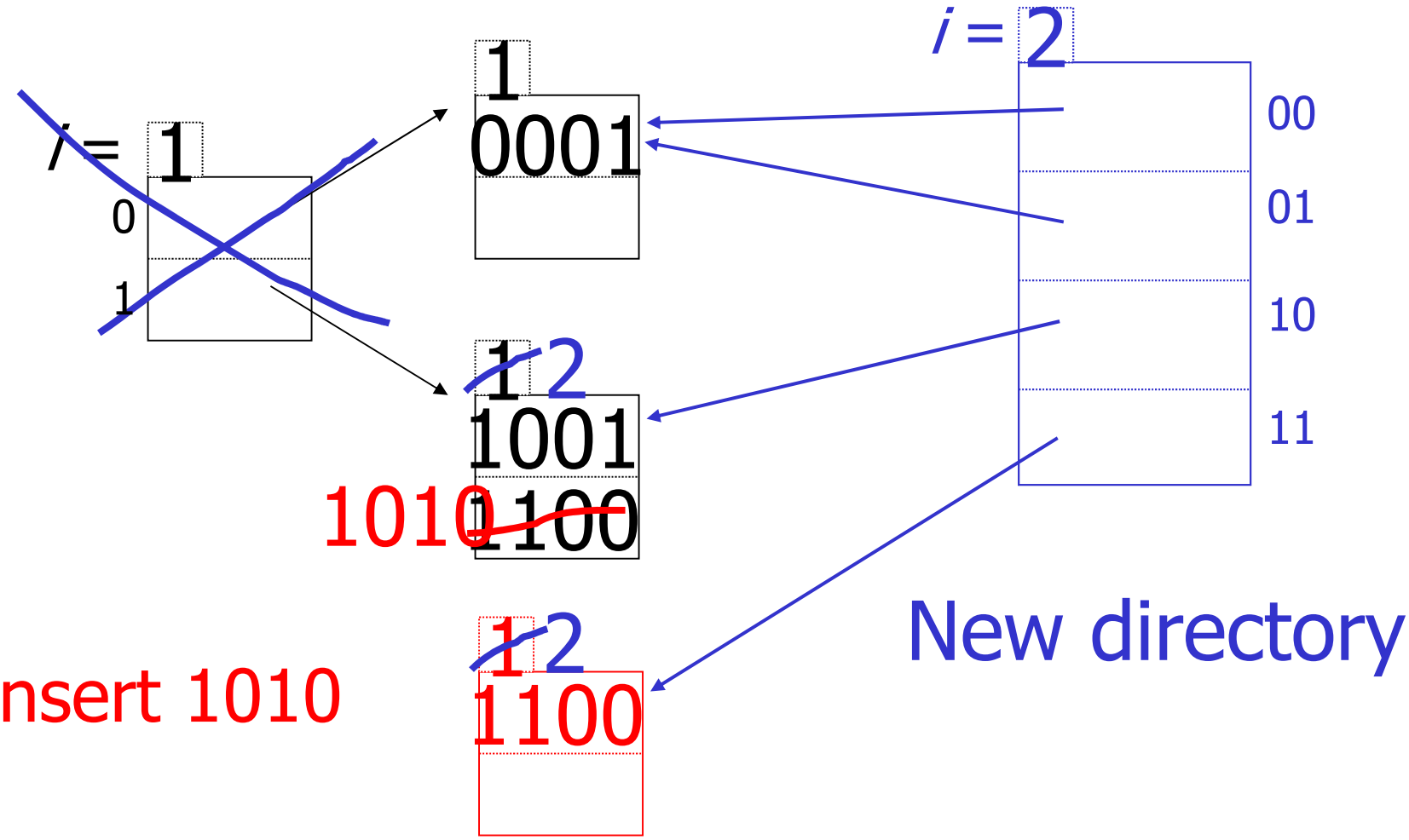


Insert 1010

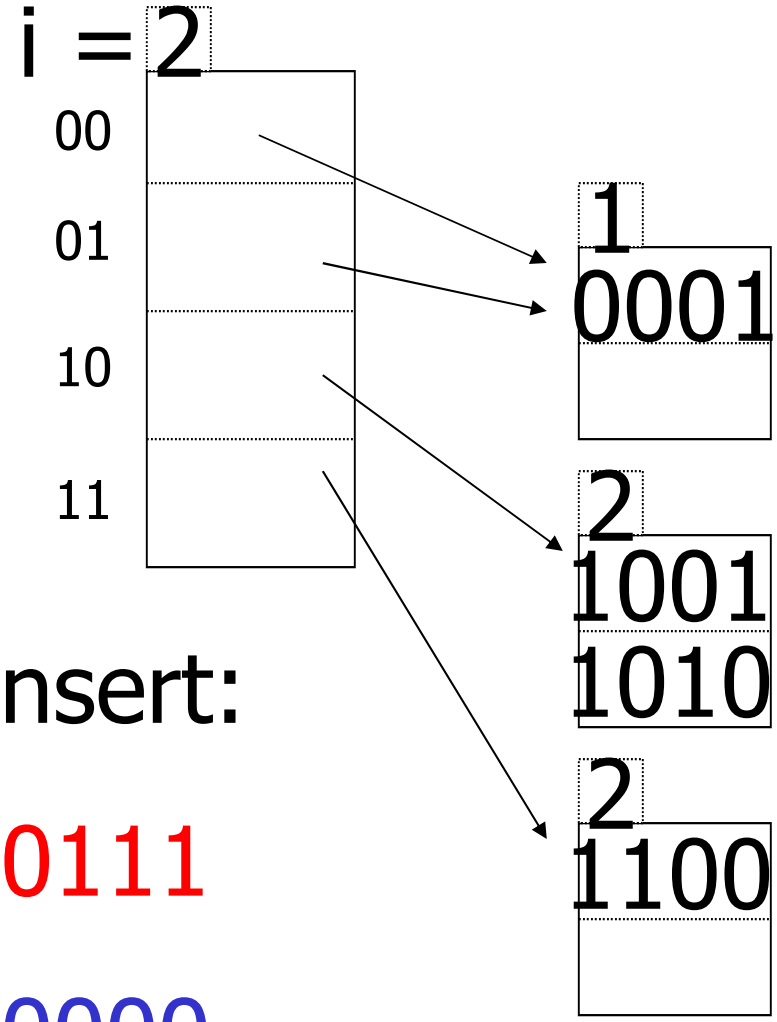
Example: h(k) is 4 bits; 2 keys/bucket



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



Example continued

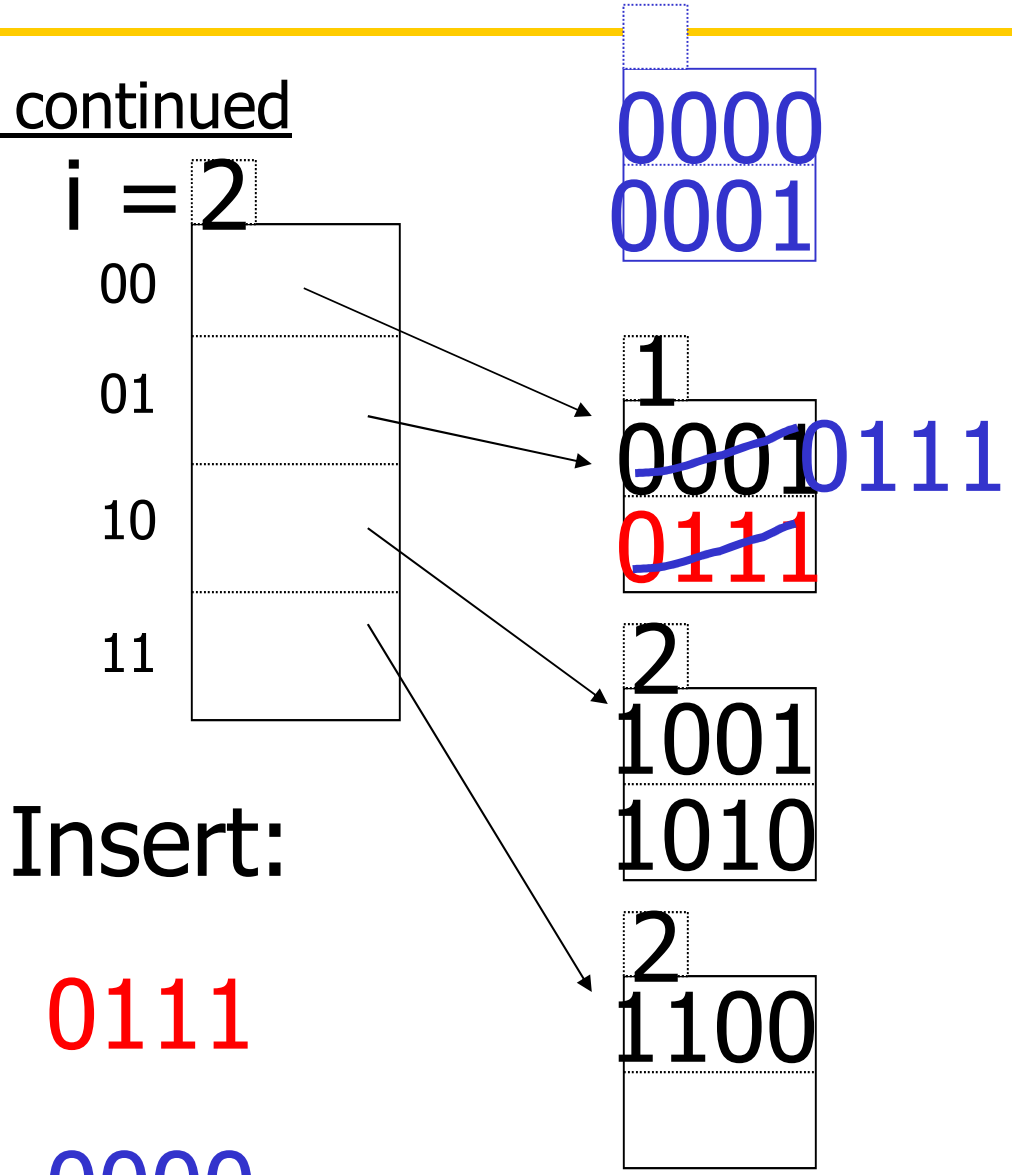


Insert:

0111

0000

Example continued

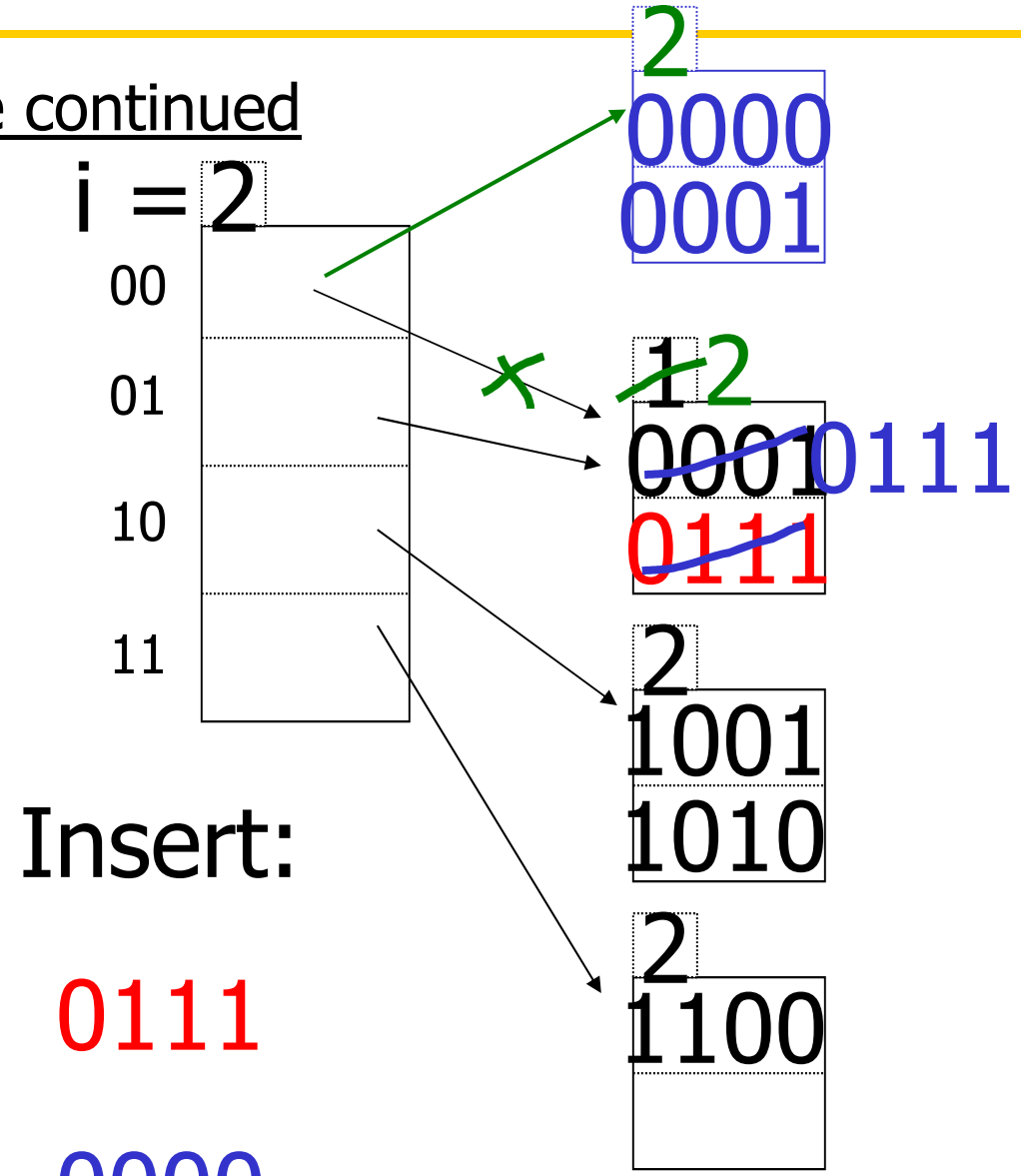


Insert:

0111

0000

Example continued

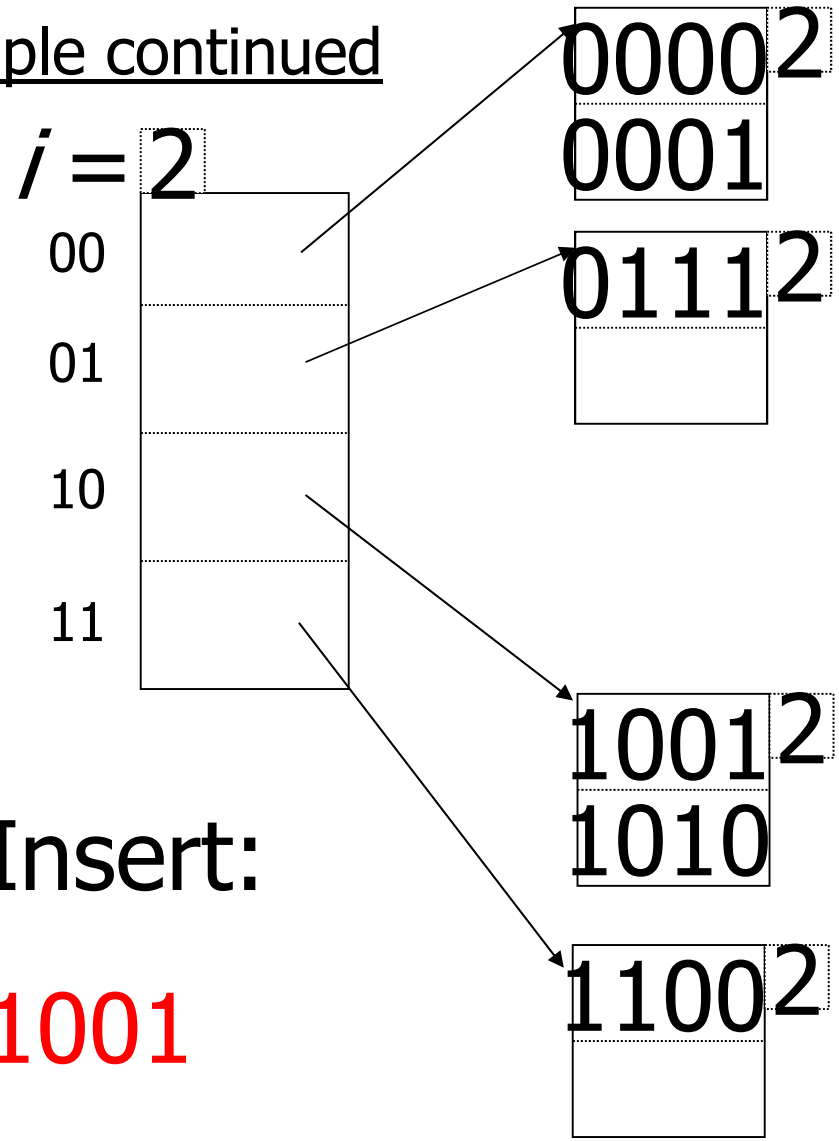


Insert:

0111

0000

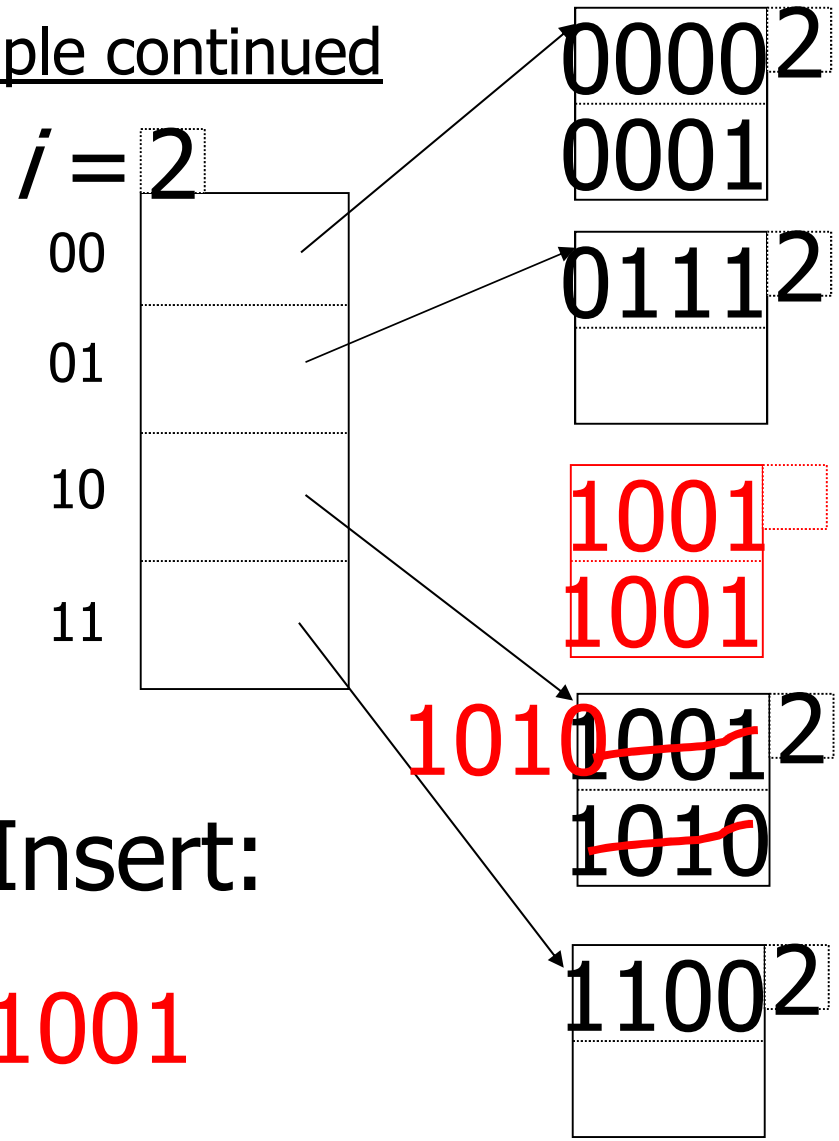
Example continued



Insert:

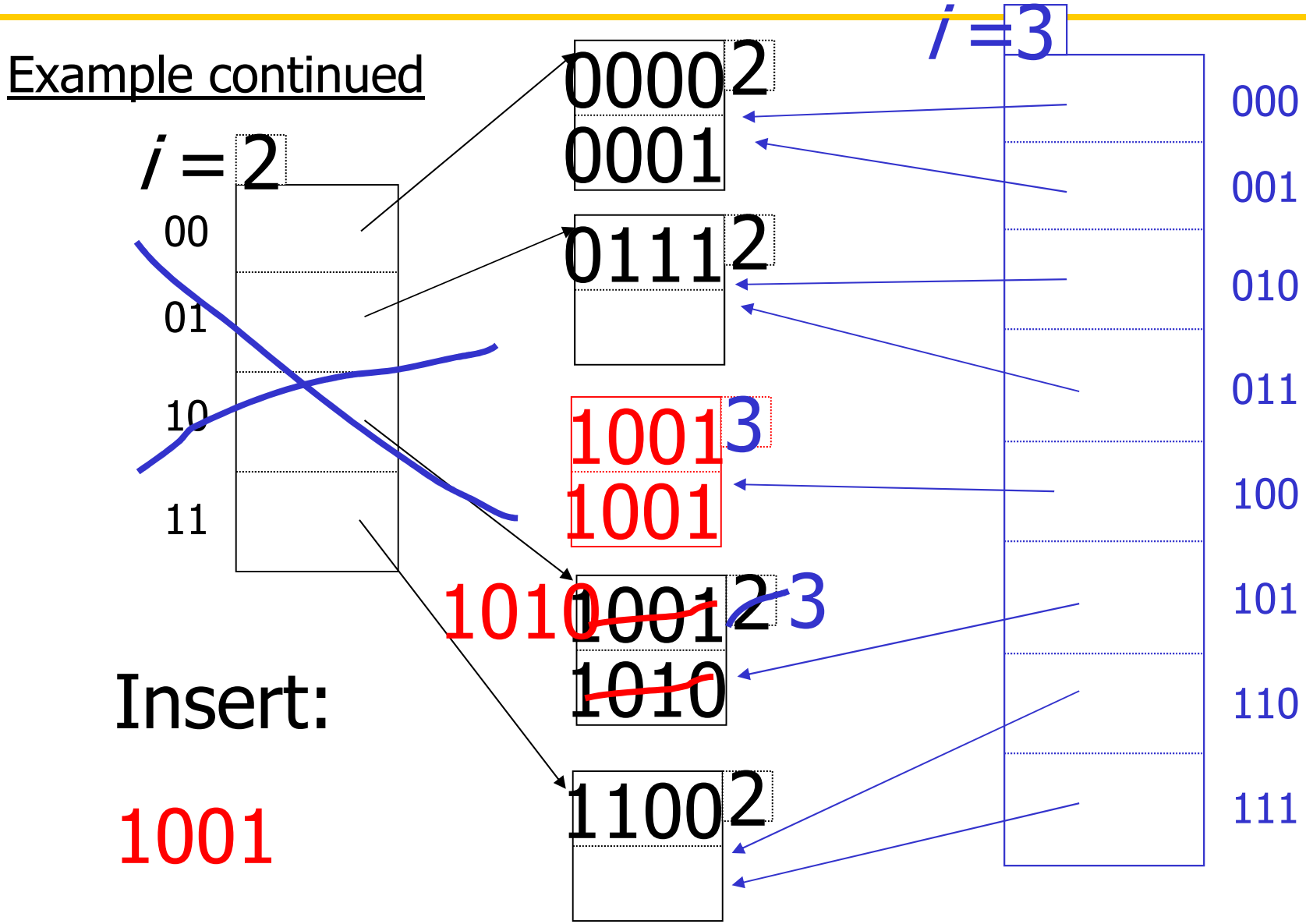
1001

Example continued



Insert:

1001


Example continued
i = 2
00
01
10
11
0000 2
0001
0111 2
1001 3
1001
1010 1001 2 3
1010
1100 2
i = 3
000
001
010
011
100
101
110
111
Insert:
1001

Extensible hashing: deletion

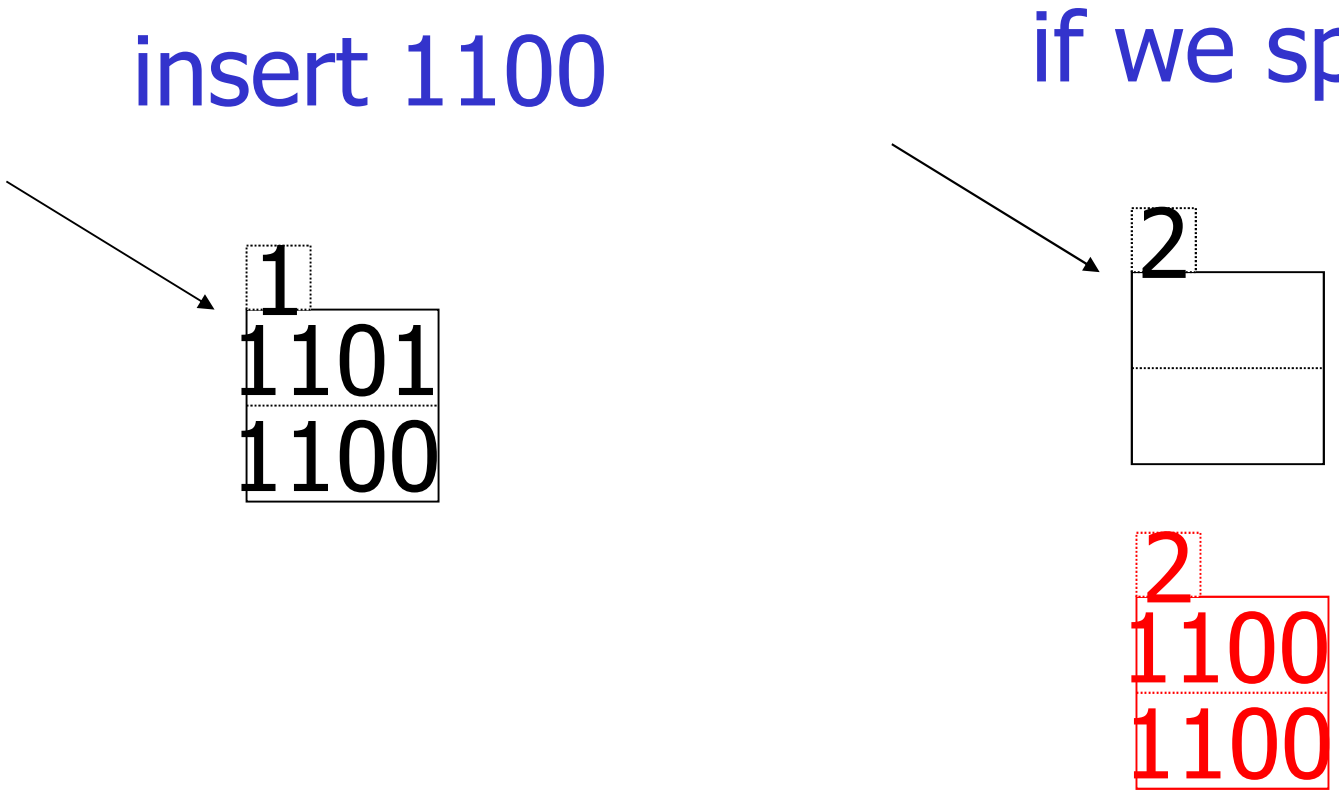
- No merging of blocks
- Merge blocks
  and cut directory if possible
  (Reverse insert procedure)

Deletion example:

- Run thru insert example in reverse!

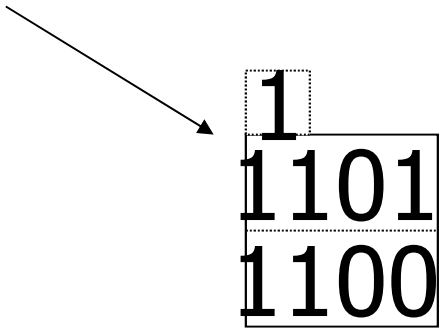
Note: Still need overflow chains

- Example: many records with duplicate keys

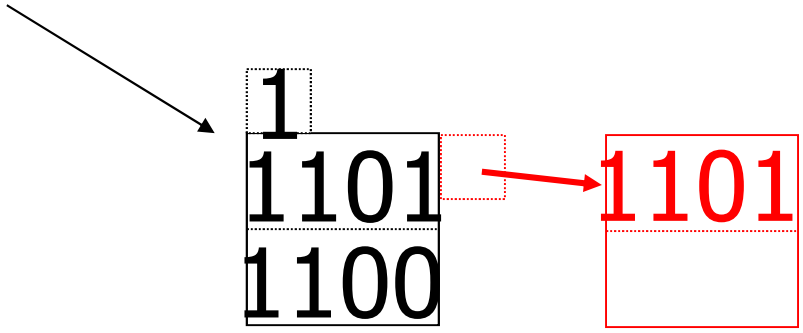


Solution: overflow chains

insert 1101



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

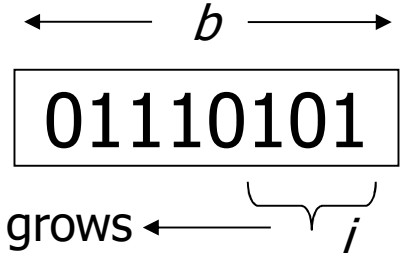
(Now it fits, now it does not)

Linear hashing

- Another dynamic hashing scheme

Two ideas:

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

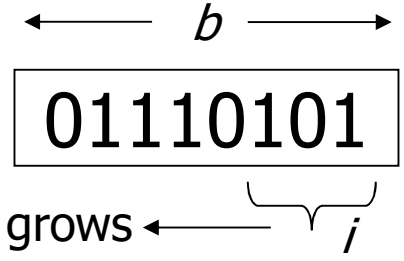


Linear hashing

- Another dynamic hashing scheme

Two ideas:

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



(b) File grows linearly

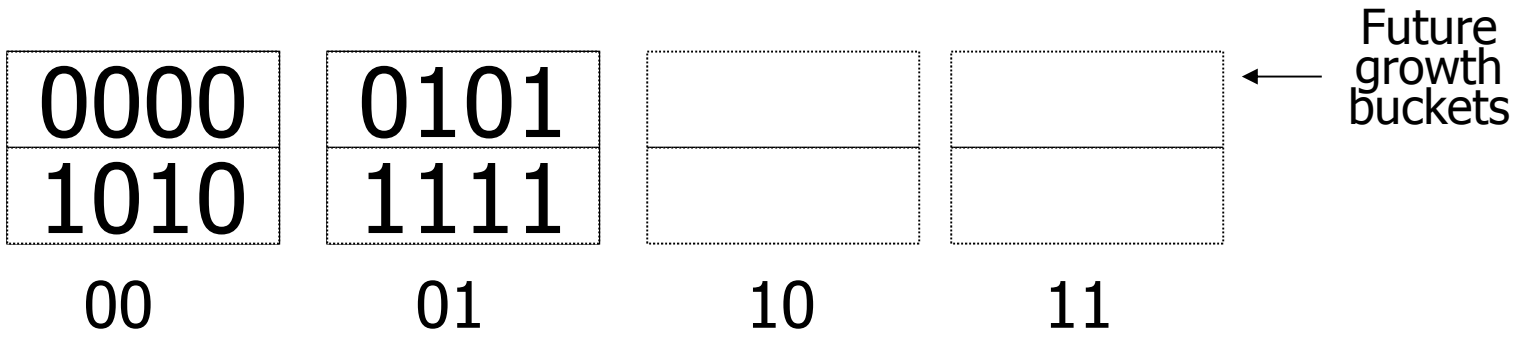
☛ When do we expand file?

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

- After every insertion, check if U > threshold then increase buckets by 1

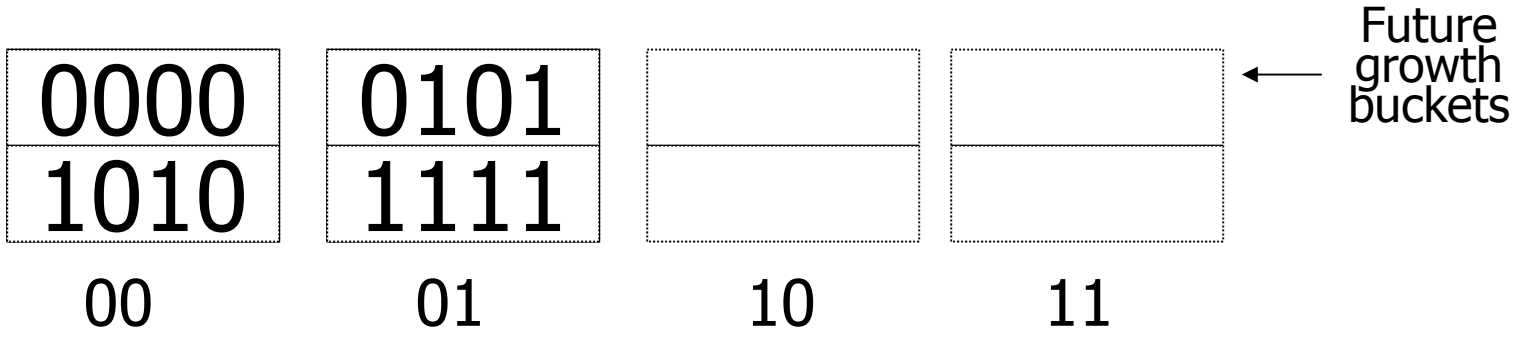
- If you run out of bits, add 1 more
  - 00 becomes 000

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)

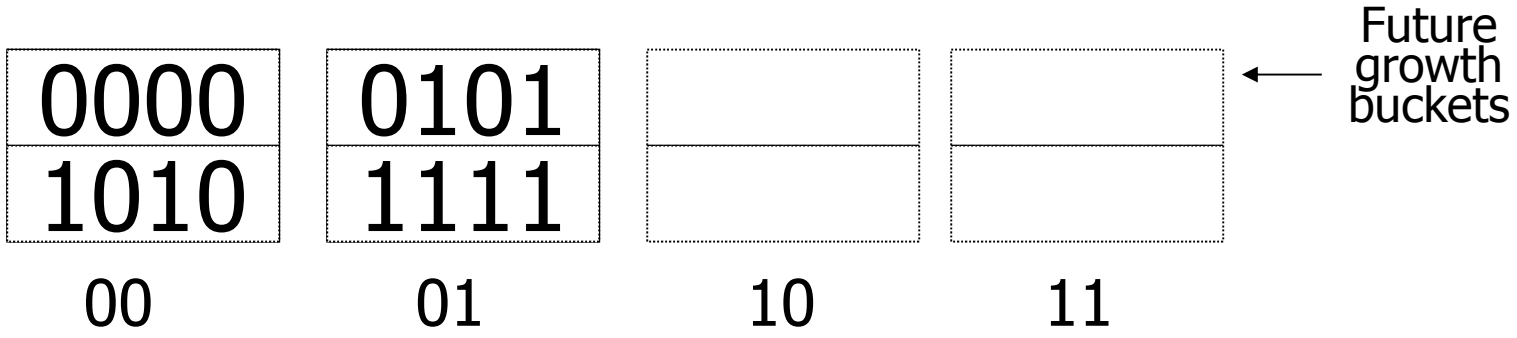
Rule If h(k)[i] ≤ m, then

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

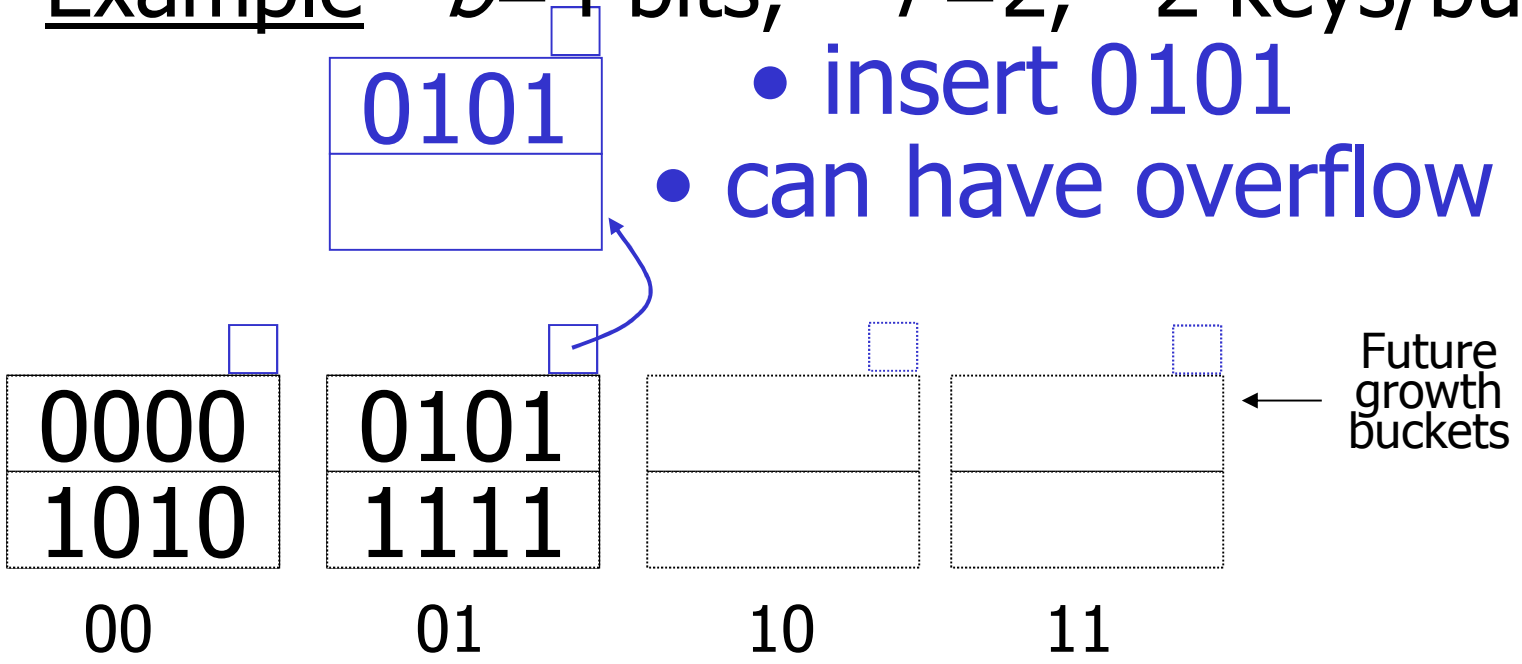


m = 01 (max used block)

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



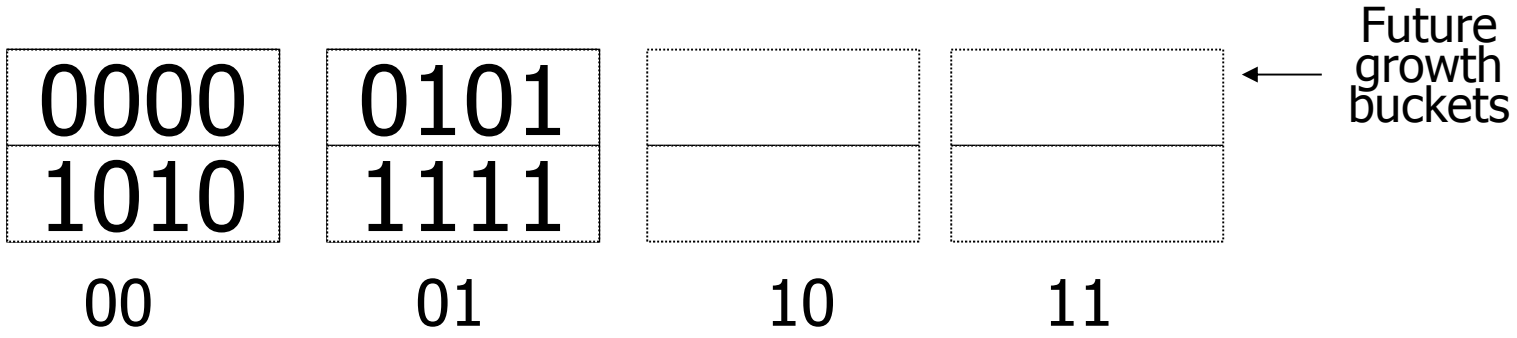
m = 01 (max used block)

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

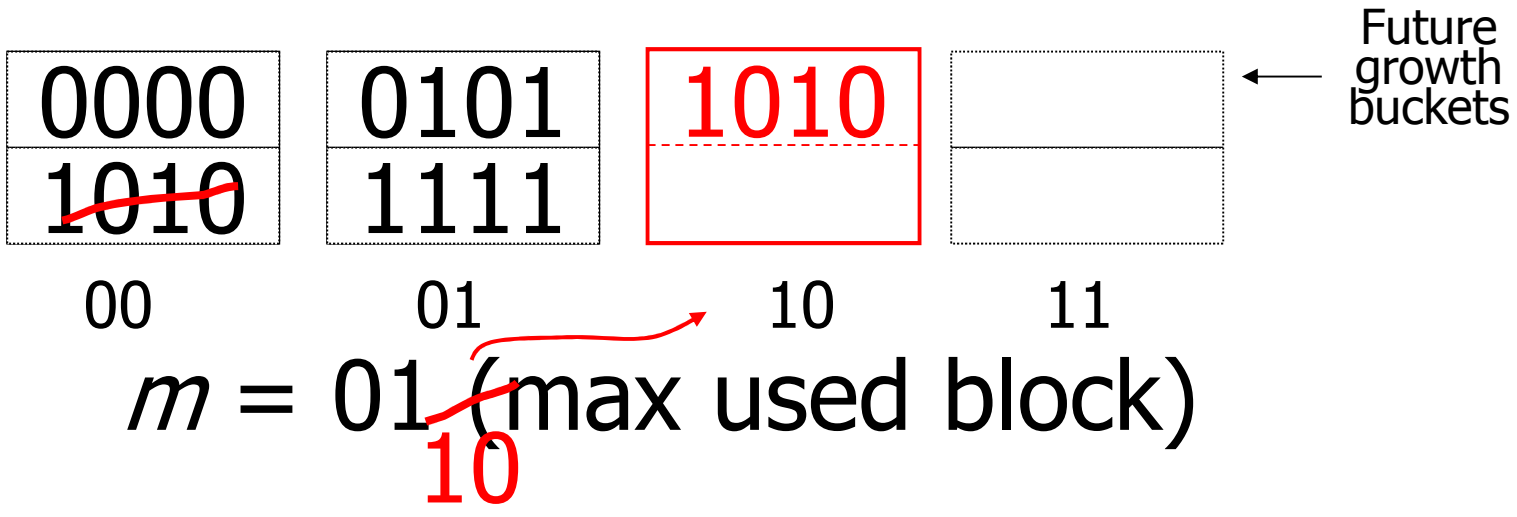


m = 01 (max used block)

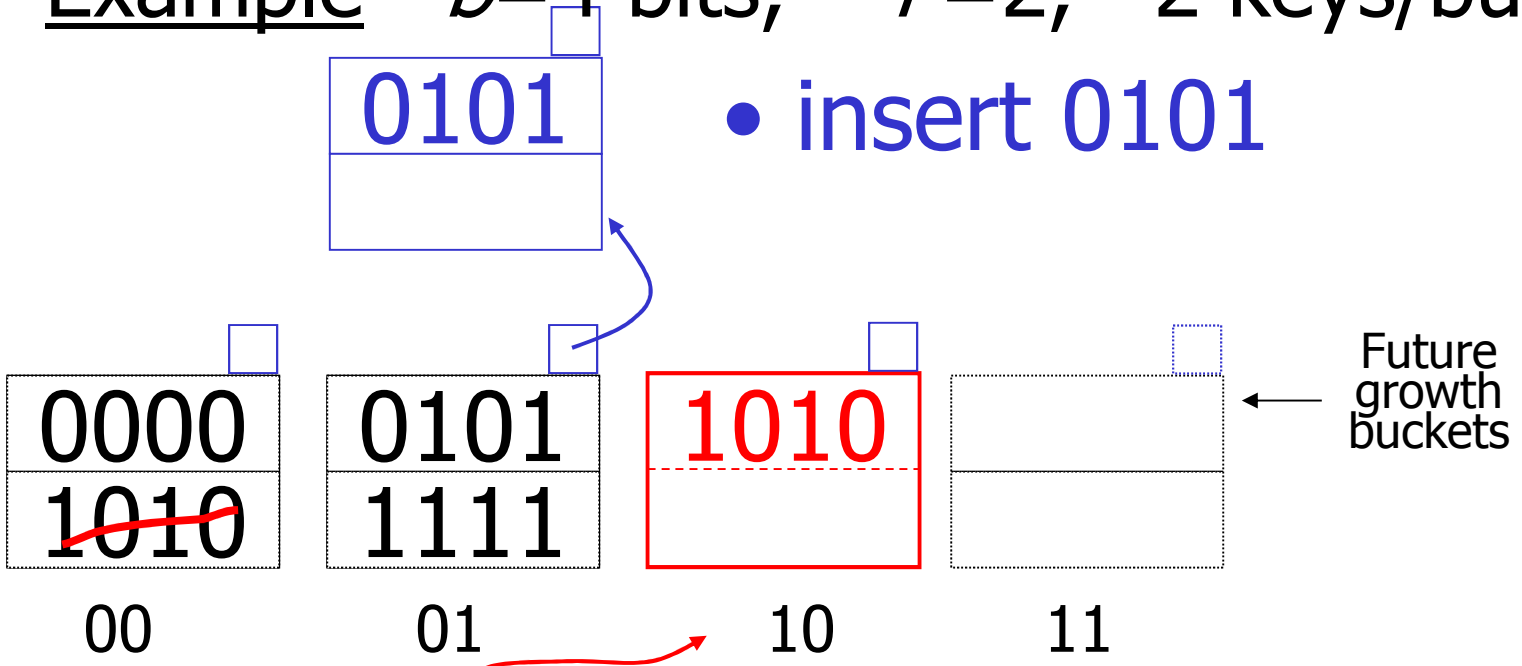
- After every insertion, check if U > threshold (e.g. 0.8) then increase buckets by 1

U= # used slots / total # of slots

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



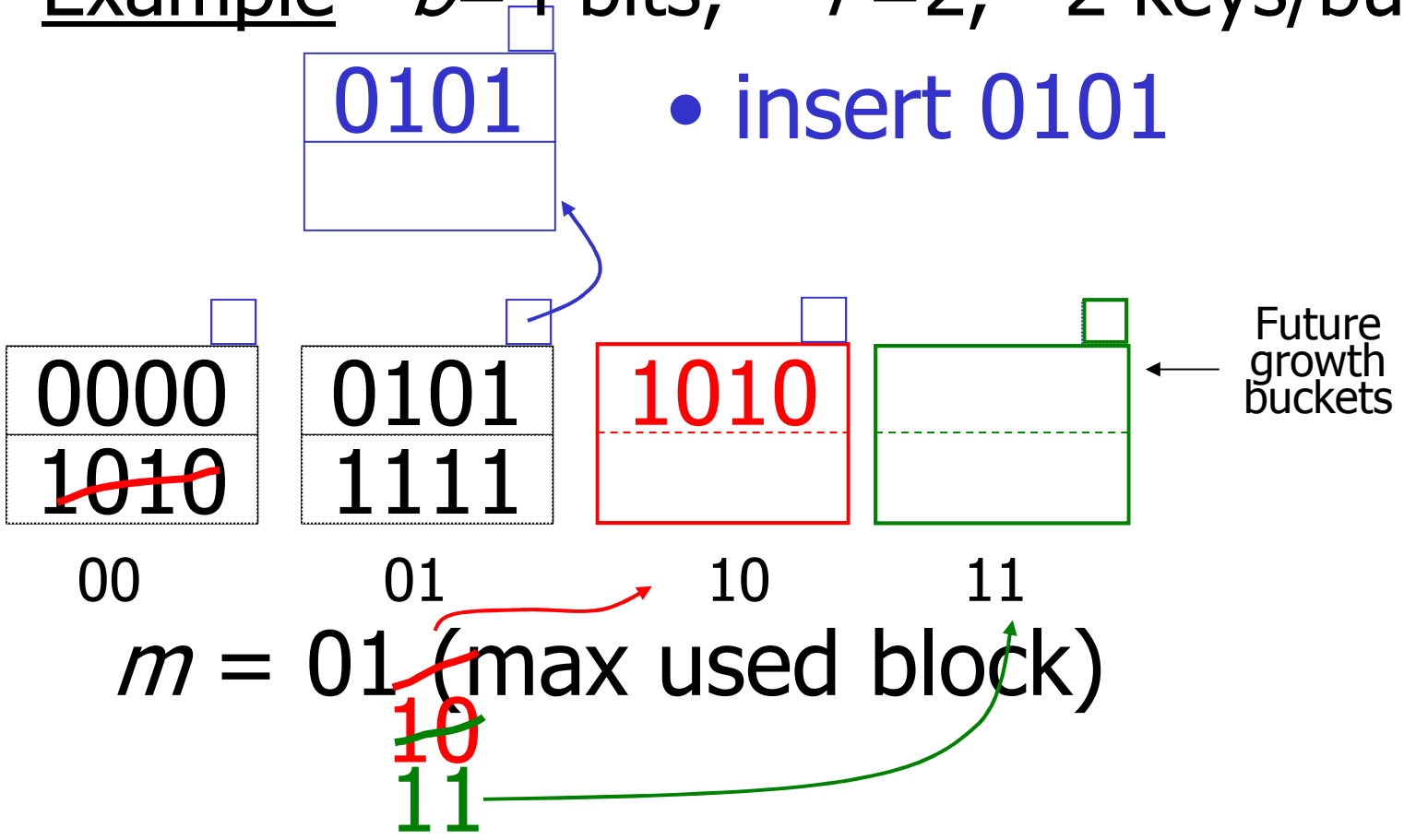
Example b=4 bits, i=2, 2 keys/bucket



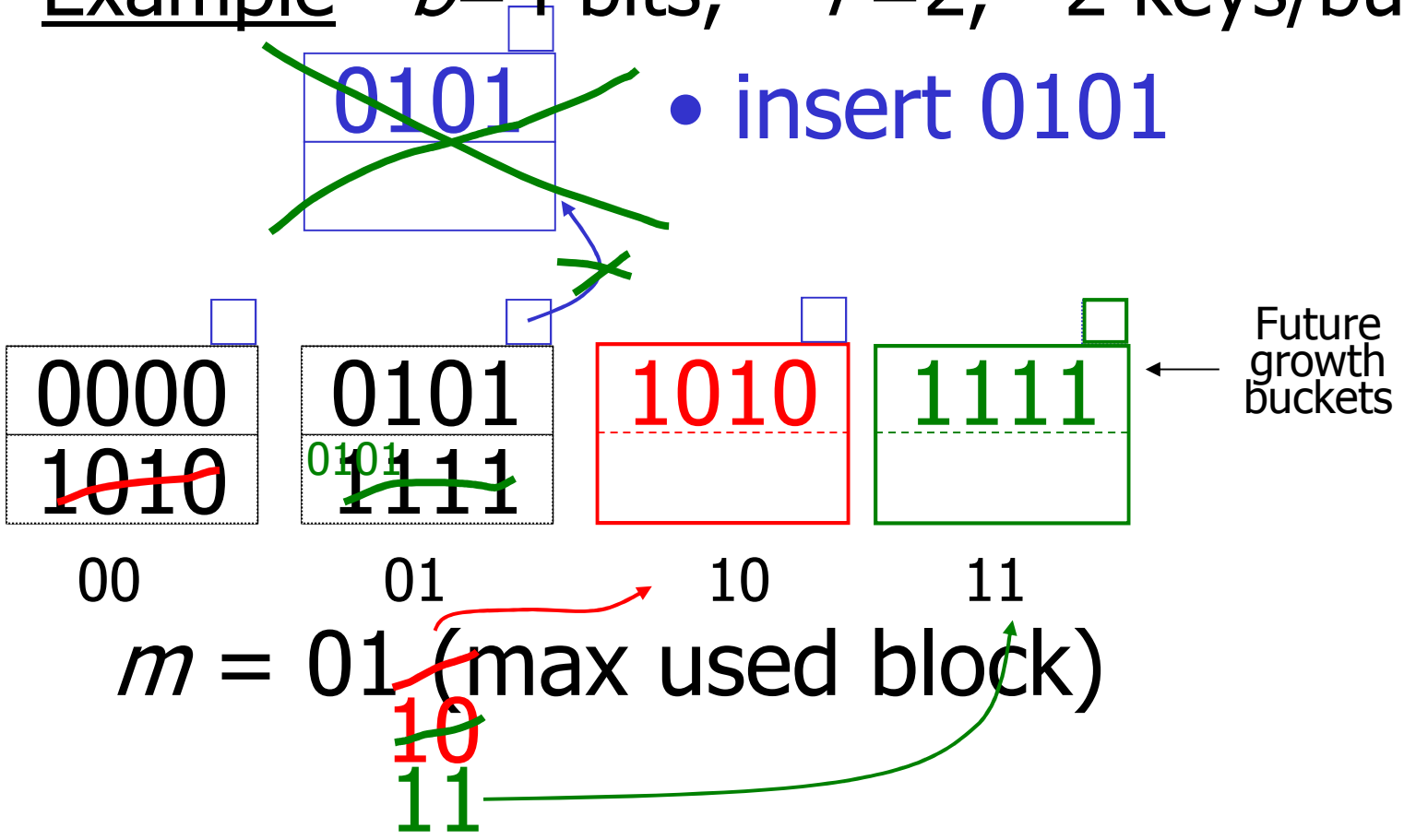
m = 01 (max used block)

10

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

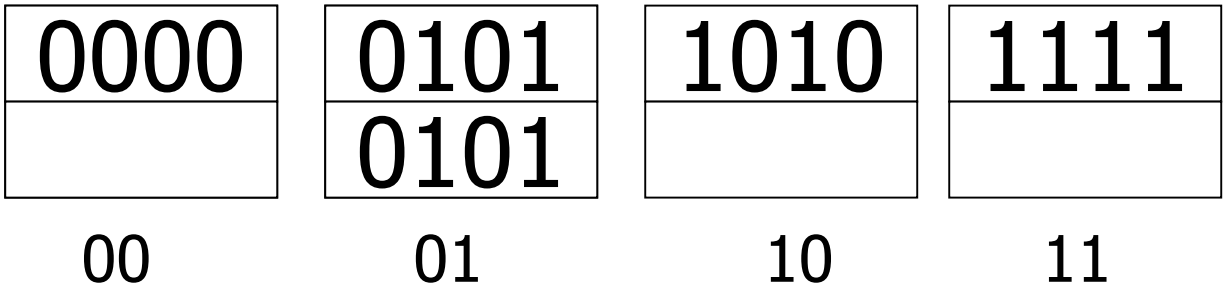


Example b=4 bits, i=2, 2 keys/bucket



Example Continued: How to grow beyond this?

$i = 2$

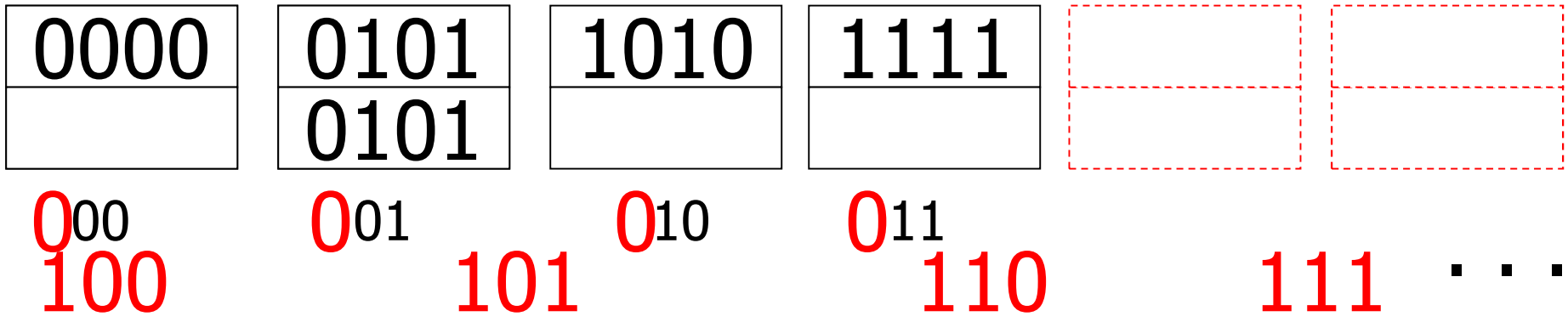


. . .

$m = 11$ (max used block)

Example Continued: How to grow beyond this?

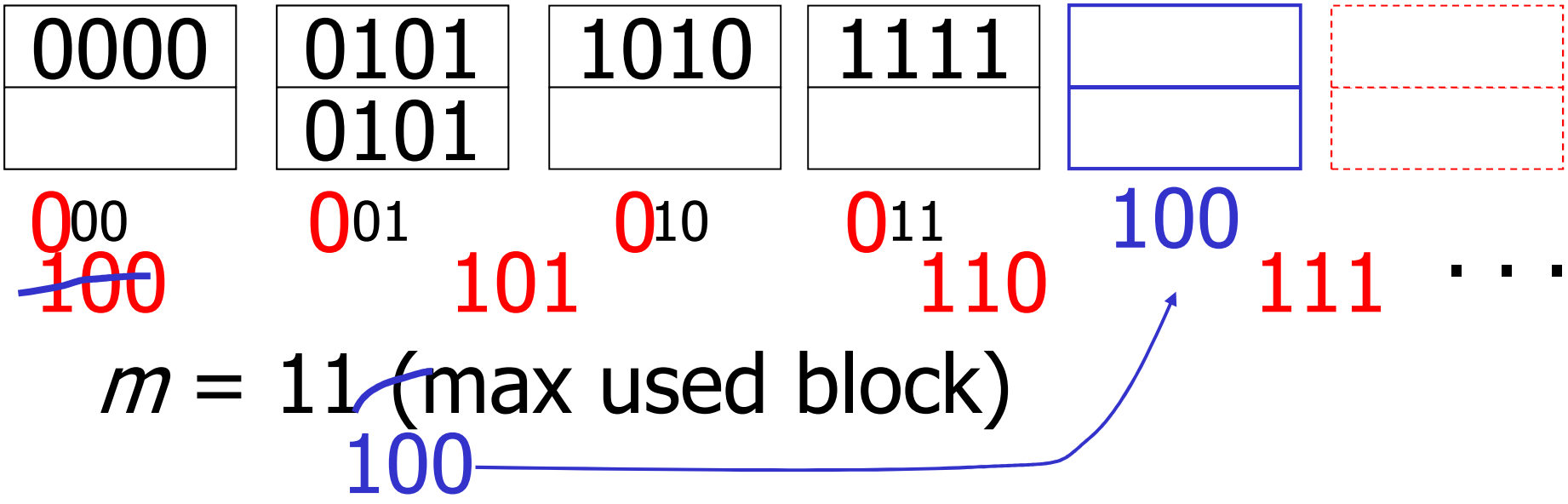
$i = 2$ 3



m = 11 (max used block)

Example Continued: How to grow beyond this?

$i = 2$ 3

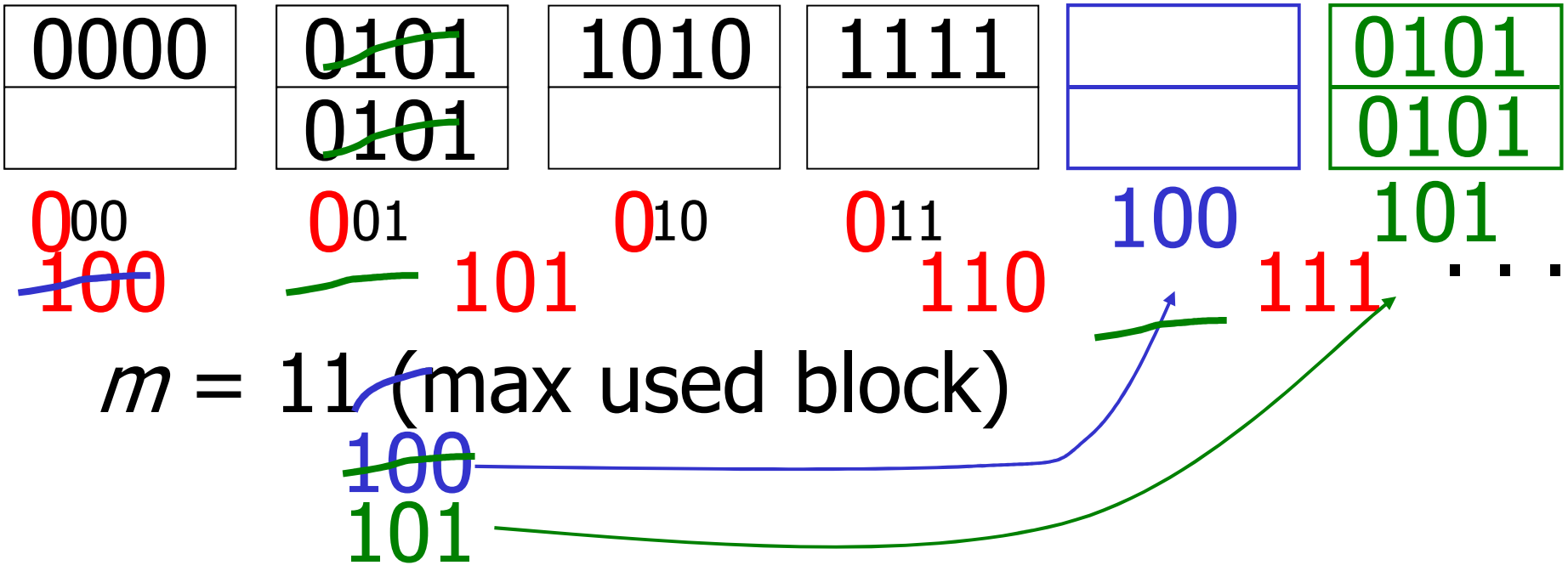


$m = 11$ (max used block)

100

Example Continued: How to grow beyond this?

$i = 2$ 3



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

Hash-based Indices in RDBMS:

- PostgreSQL:

  CREATE INDEX name ON table USING hash (column);

  http://www.postgresql.org/docs/9.1/static/indexes-types.html

Read

- Reference:
  - dropbox:

https://www.dropbox.com/s/fqv14g1zqhhl6k5/Chaps14-19.pdf

  - gdrive:

https://drive.google.com/file/d/0B03SaNyIsL_2U2pmMjVrMDl4MWs/view?usp=sharing

  - Read: pg 648-661