

Advanced Database Technology

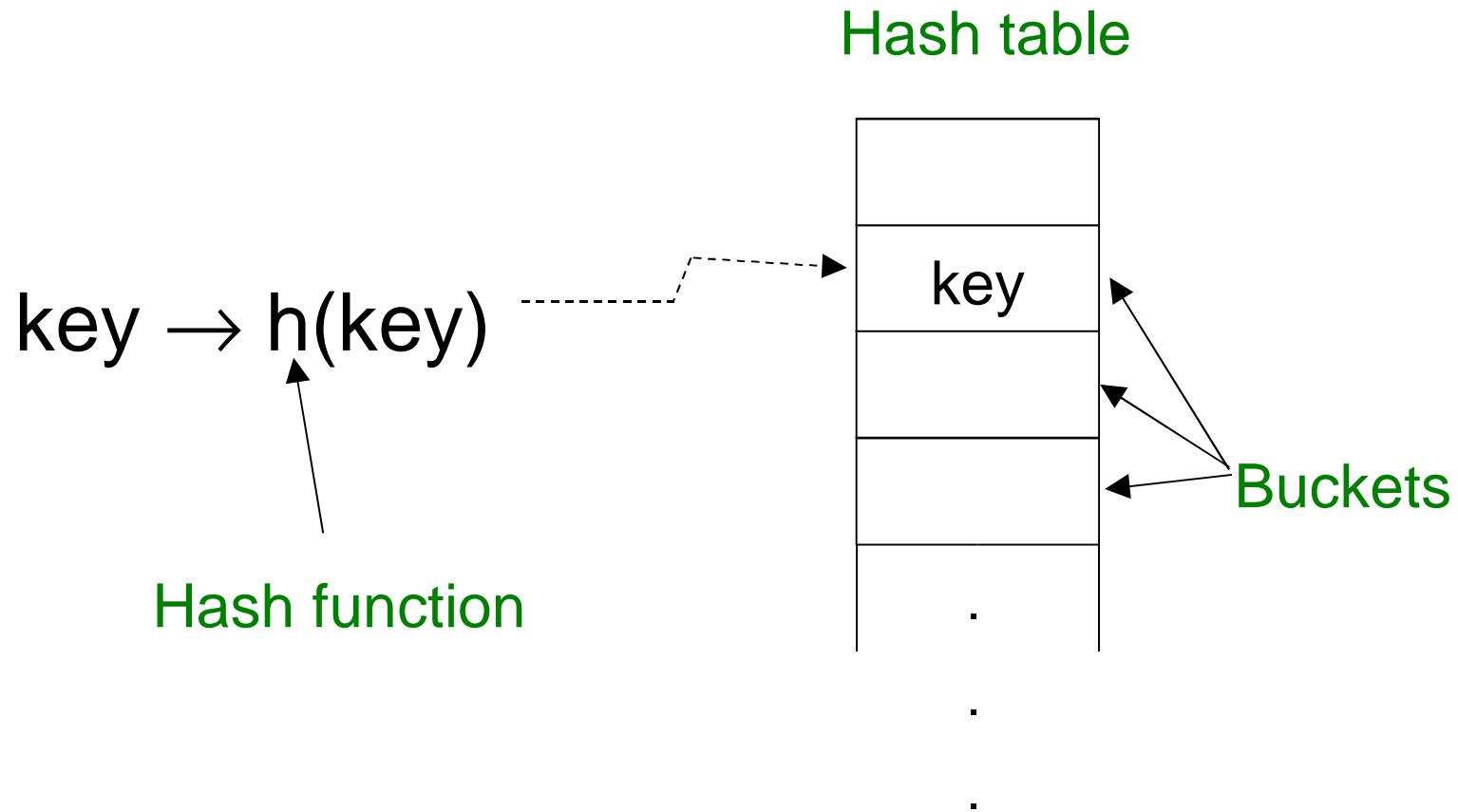
March 6, 2003

INDEXING II

Lecture based on [GUW, 13.4] and [Pagh03, sec. 4]

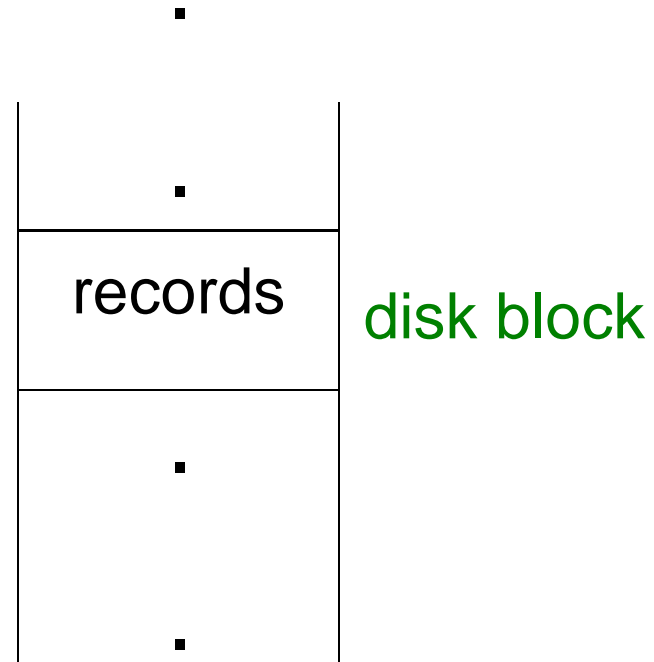
Slides based on
Notes 05: Hashing and more
for Stanford CS 245, fall 2002
by Hector Garcia–Molina

Hashing in a nutshell



Hashing as primary index

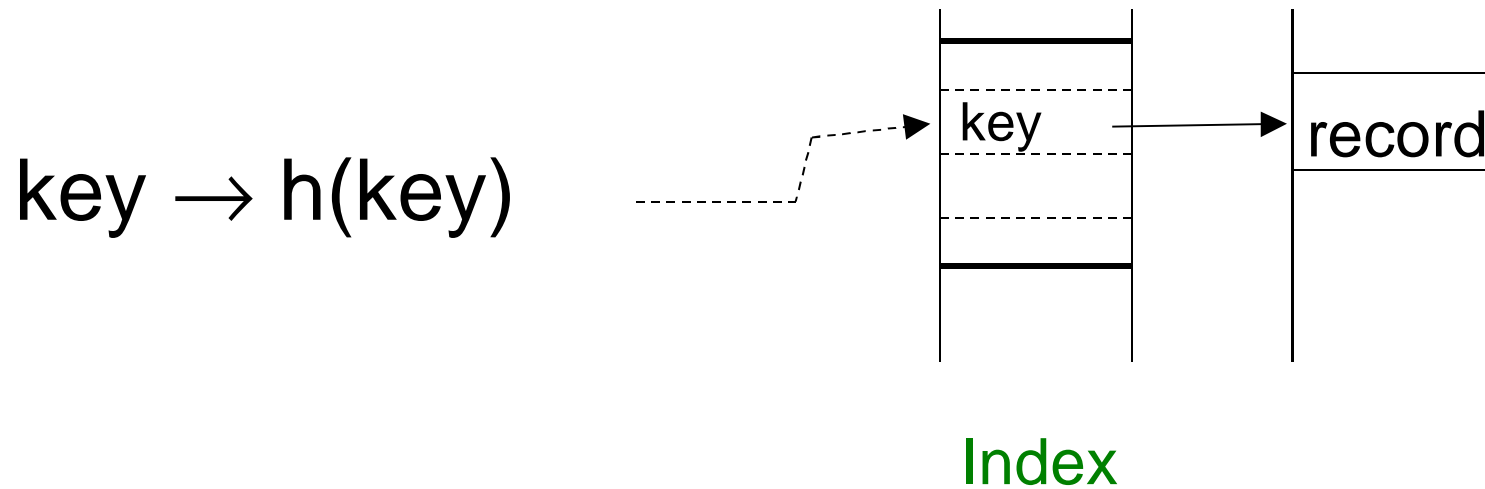
key \rightarrow h(key)



Note on terminology:

The word "indexing" is often used synonymously with "B-tree indexing".

Hashing as secondary index



Today we discuss hashing as **primary index**.
Can always be transformed to a secondary
index using indirection, as above.

Choosing a hash function

Book's suggestions (p. 650):

- Key = 'x₁ x₂ ... x_n', n byte character string:

$$h(\text{Key}) = (x_1 + x_2 + \dots + x_n) \bmod b$$

- Key is an integer:

$$h(\text{Key}) = \text{Key} \bmod b$$

PROBLEM SESSION

Find examples of key sets that make these functions behave badly

Choosing a randomized function

Another approach (not mentioned in book):

- Choose **h at random** from some set of functions.
- This can make the hashing scheme behave well **regardless** of the key set.
- E.g., "**universal hashing**" makes chained hashing perform well (in theory and practice).
- Details out of scope for this course...

Insertions and overflows

INSERT:

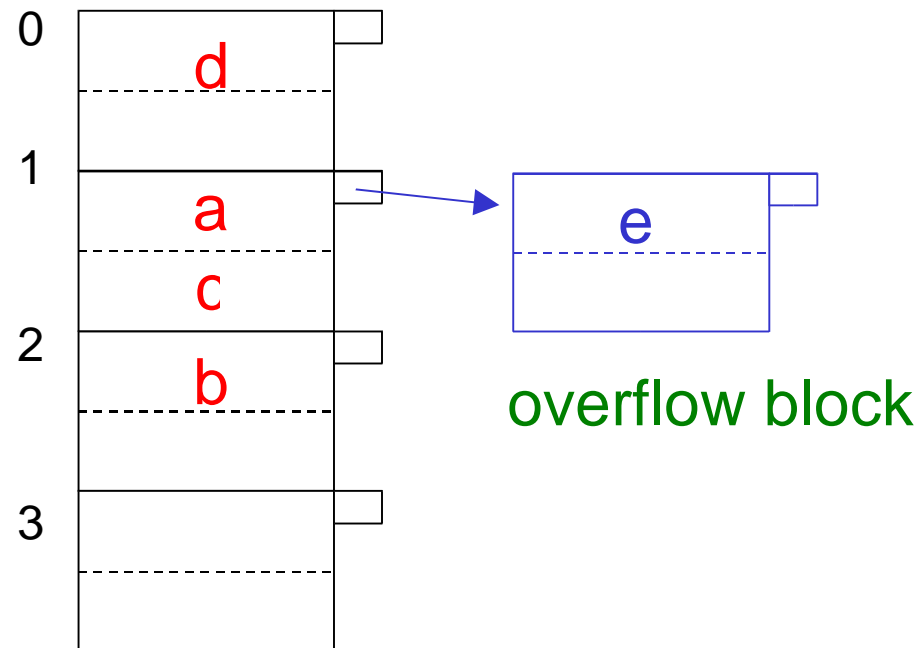
$h(a) = 1$

$h(b) = 2$

$h(c) = 1$

$h(d) = 0$

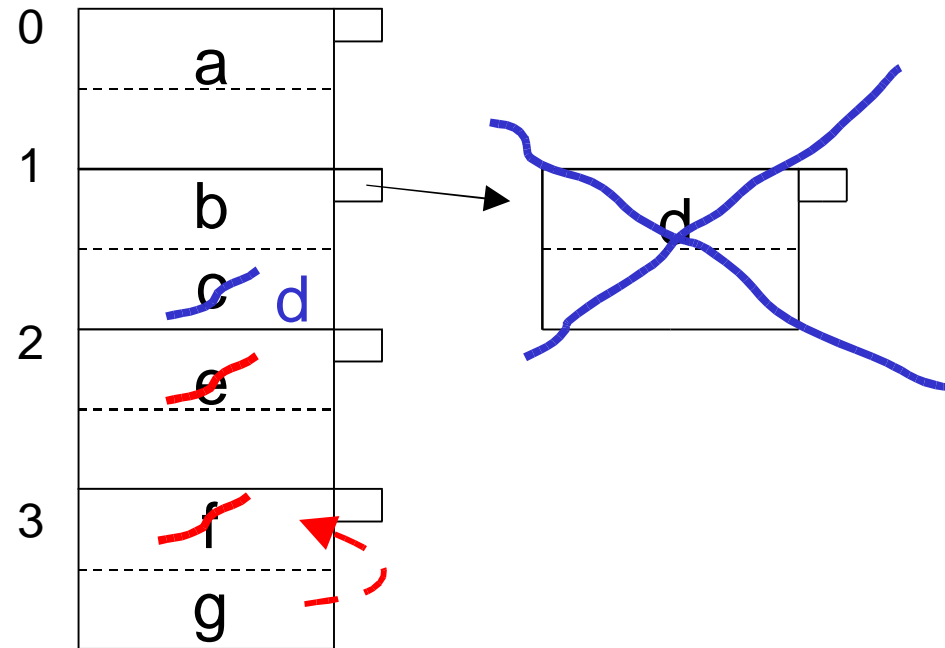
$h(e) = 1$



Deletions

Delete:

e
f
c



Analysis – external chained hashing

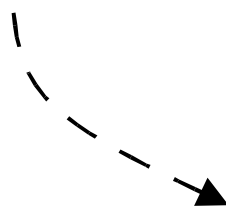
(assuming truly random hash functions)

- N keys inserted, each block (bucket) in the hash table can hold B keys.
- Suppose the hash table has size $N/\alpha B$, i.e., "is a fraction α full".
- Expected number of overflow blocks:
 $(1-\alpha)^{-2} \cdot 2^{-\Omega(B)} N$ (proof omitted!)
- Good to have many keys in each bucket (an advantage of secondary indexes).

Sometimes life is easy...

- If B is sufficiently large compared to N , all overflow blocks can be kept in internal memory.
- Lookup in 1 I/O.
- Update in 2 I/Os.

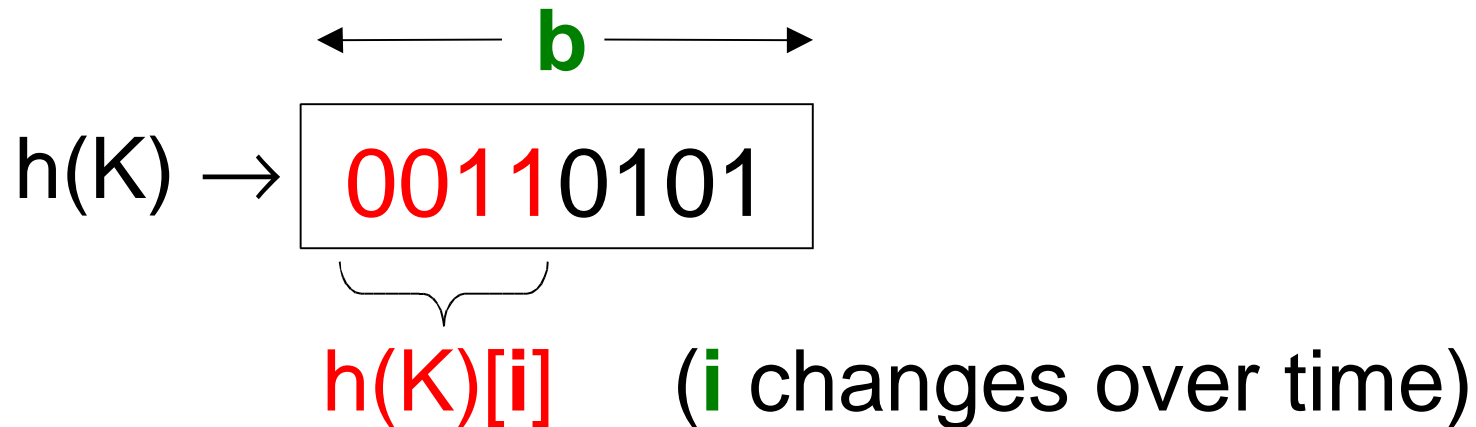
Coping with growth

- Overflows and global rebuilding
 - Dynamic hashing
- 
- Extendible hashing
 - Linear hashing
 - Uniform rehashing

Extendible hashing

Two ideas:

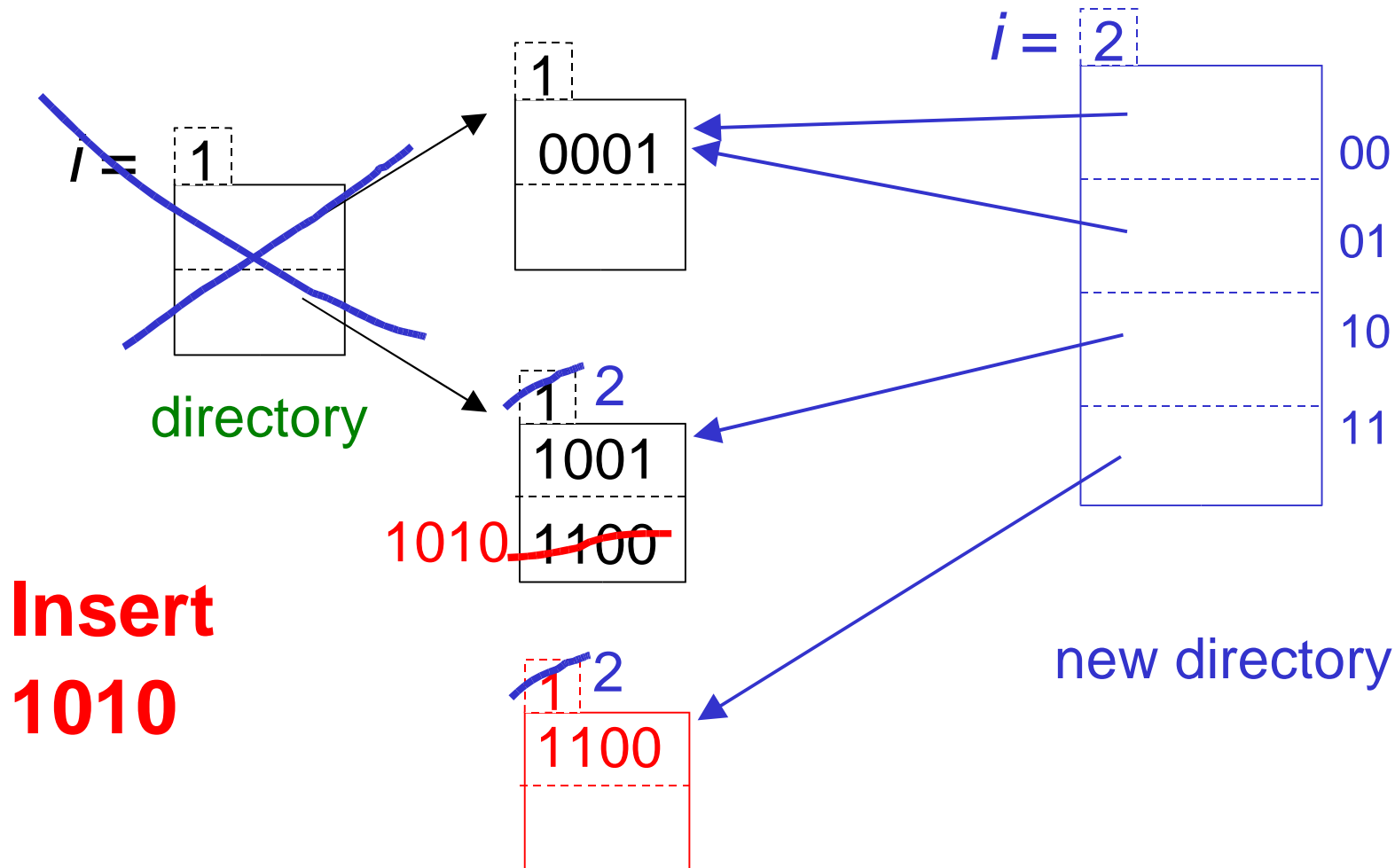
(a) Use i of b bits output by hash function



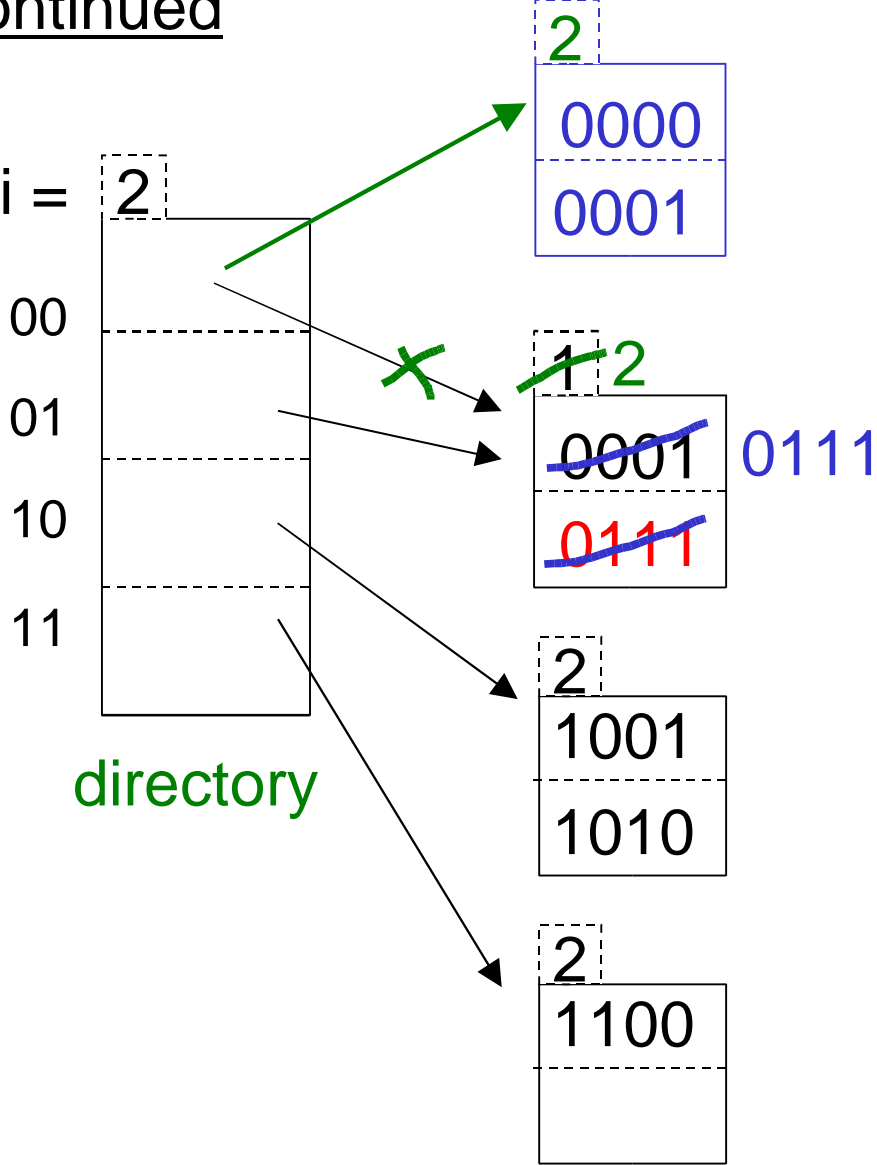
(b) Look up pointer to record in a **directory**.

Extendible hashing example

$h(K)$ is 4 bits, 2 keys/bucket. In examples we assume $h(K)=K$.



Example continued

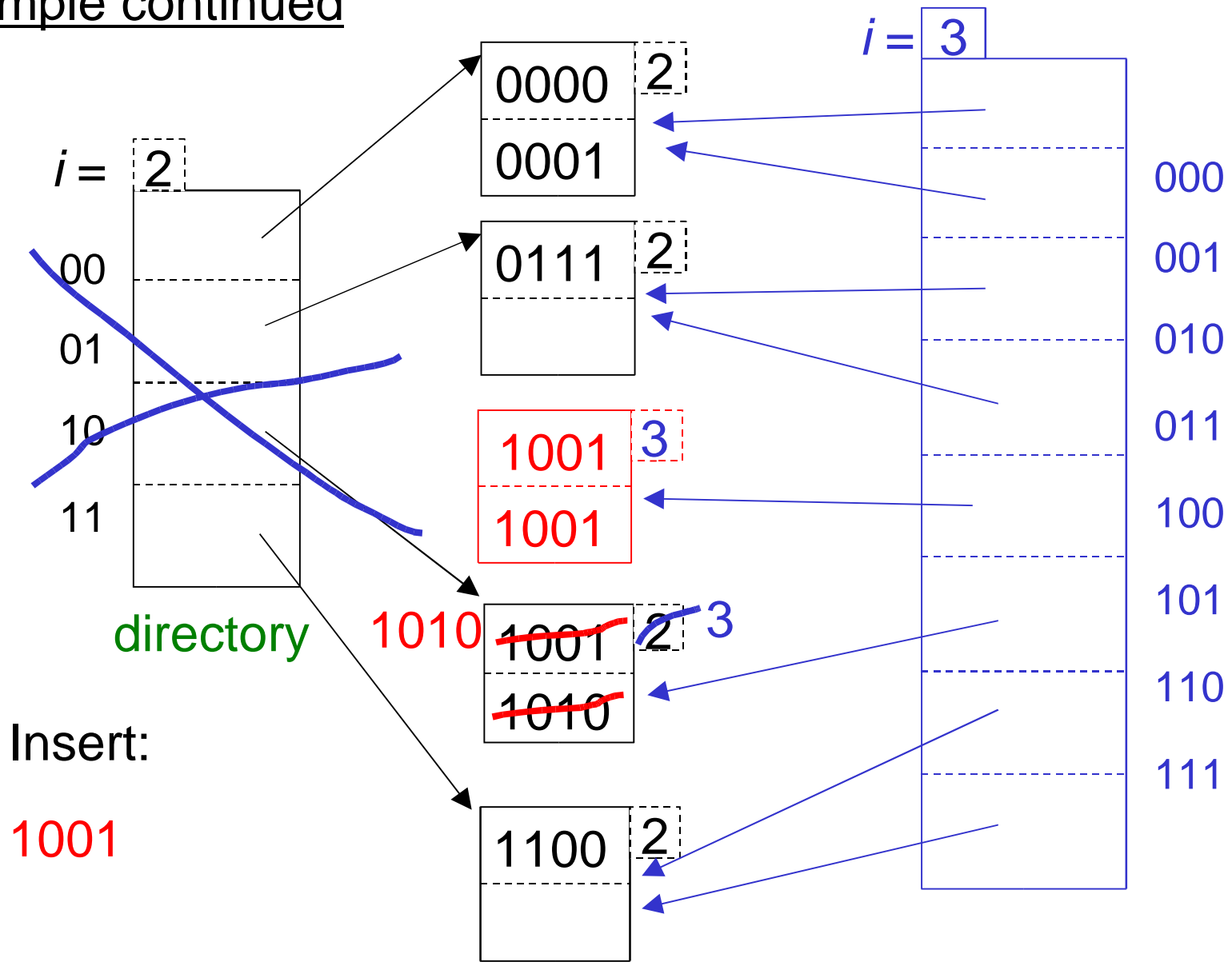


Insert:

0111

0000

Example continued



Extendible hashing deletion

Straightforward:

Merge blocks, and halve directory if possible (reverse insert procedure).

Analysis – extendible hashing

(assuming truly random hash functions)

- N keys inserted, each block (bucket) in the hash table can hold B keys.
- Blocks are about 69% full on average (proof omitted!)
- Expected size of directory is around $\frac{4N^{1+1/B}}{B}$ (proof omitted!)
- Again: Good to have large B .

Problem session

Compare extendible hashing to a sparse index:

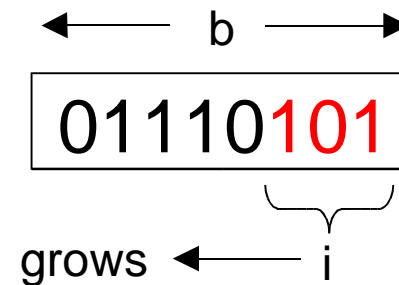
- When is one more efficient than the other?
- Consider various combinations of **N**, **B** and **M** (internal memory).

Linear hashing

–another dynamic hashing scheme

Two ideas:

(a) Use i (low order) bits of $h(K)$

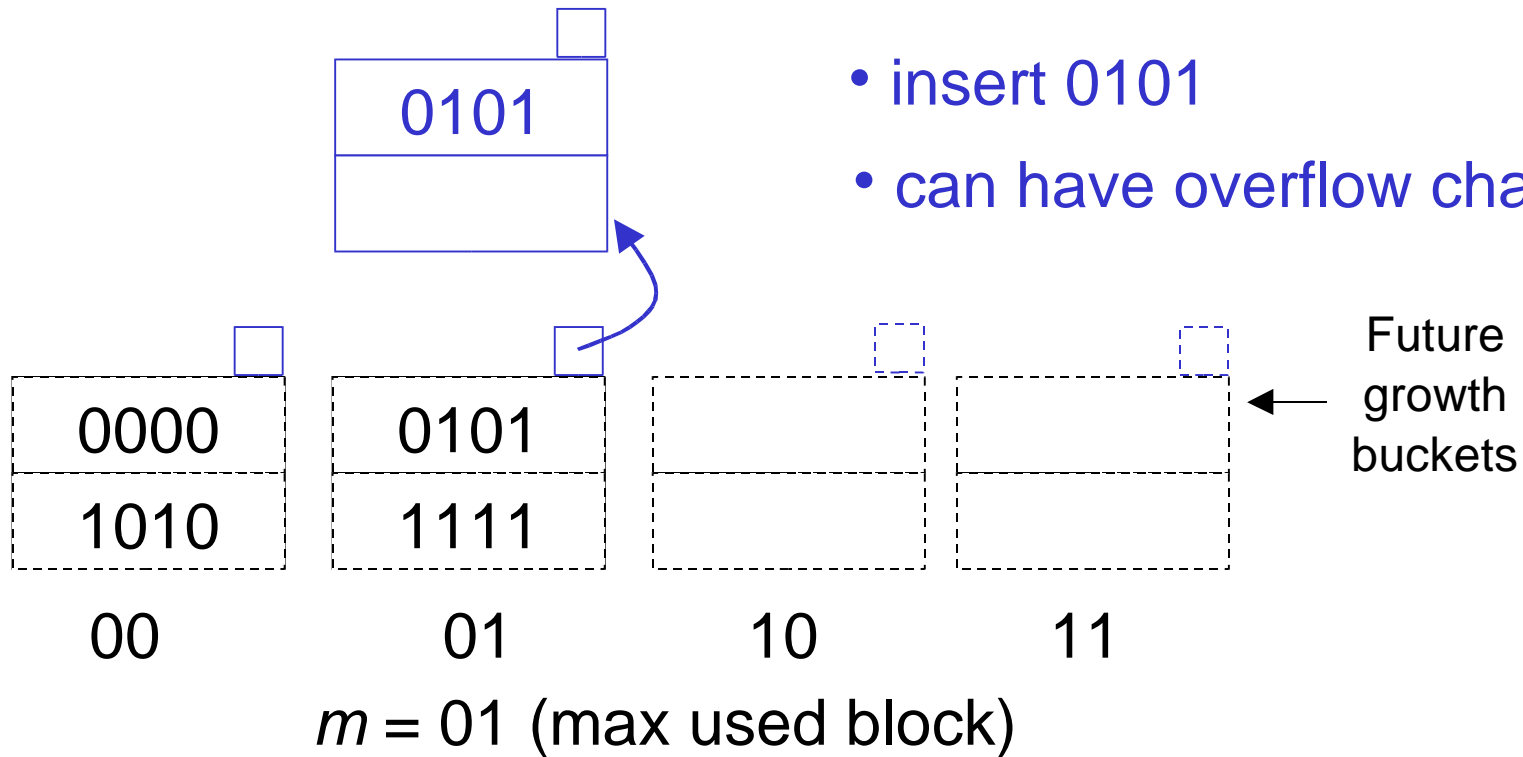


(b) Hash table grows one bucket at a time



Linear hashing example

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

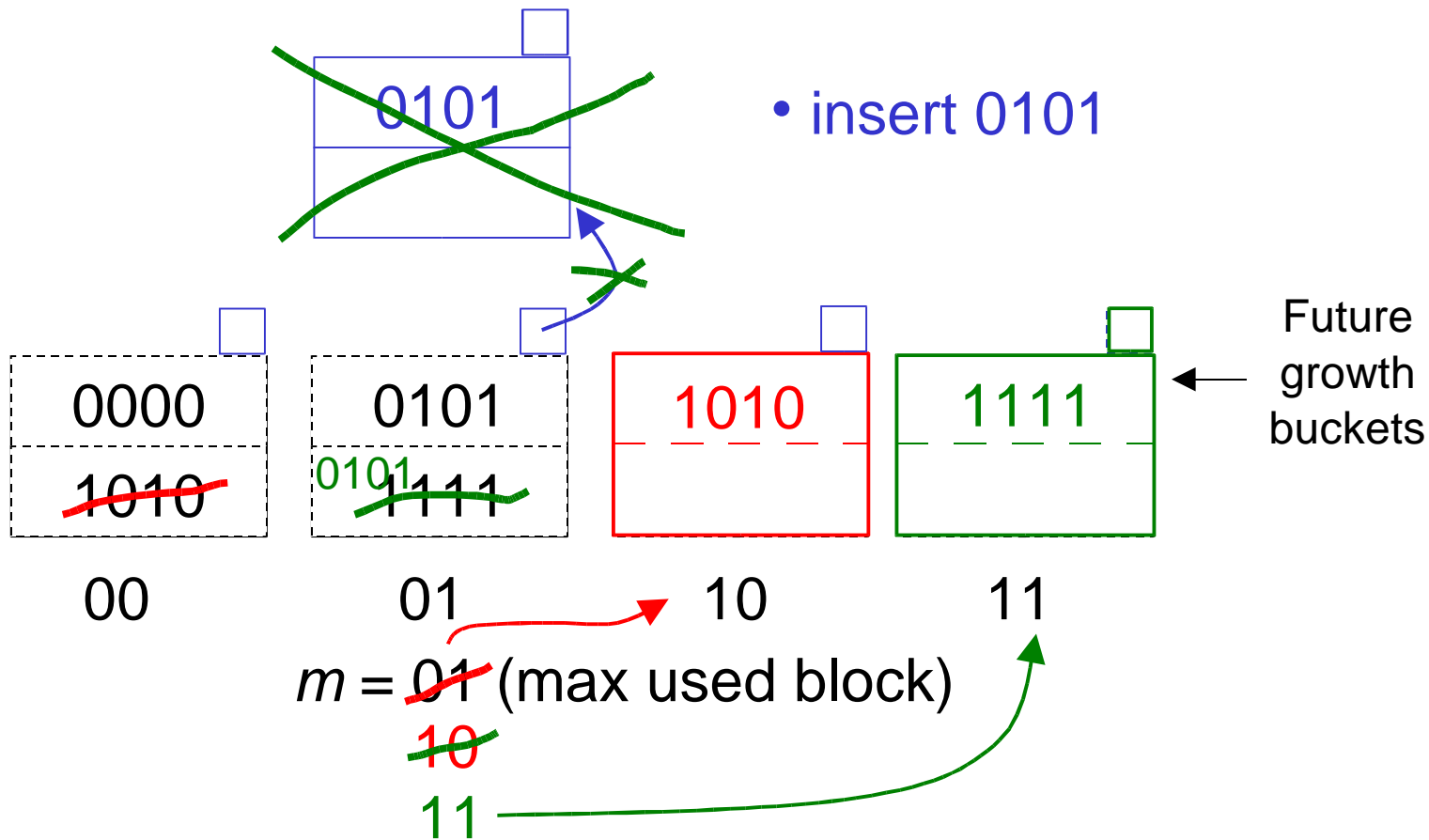


If $h(K)[i] \leq m$: Look at bucket $h(k)[i]$

Otherwise: Look at bucket $h(k)[i] - 2^{i-1}$

Linear hashing example, cont.

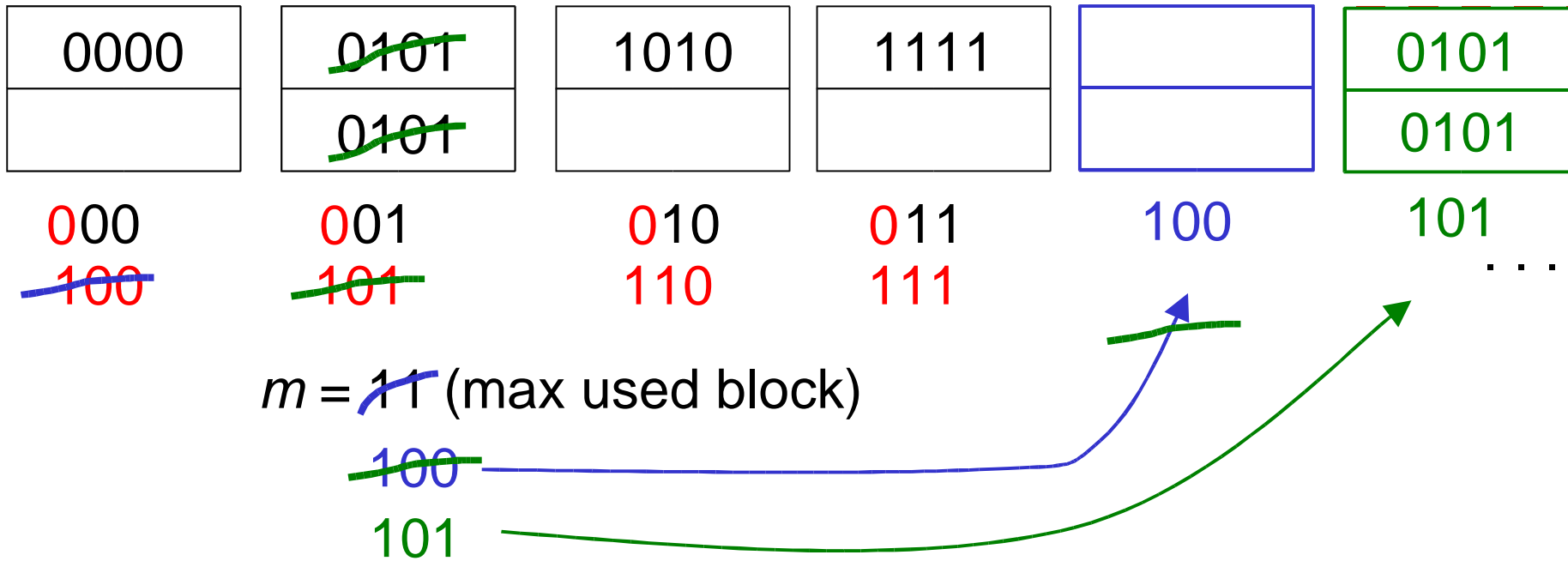
b=4 bits, i =2, 2 keys/bucket



Linear hashing example, cont.

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

$i = \cancel{2} 3$



When to expand the hash table?

- Keep track of the fraction $\alpha = (N/B)/m$
- If too close to 1 (e.g. $\alpha > 0.8$), increase m .

Performance of linear hashing

- Avoids using an index, lookup often 1 I/O.
- No good **worst-case** bound on lookups.
- Similar to chained hashing, except for the way hash values are computed.
- Unfortunately: Keys not placed uniformly in the table, so worse performance than in regular chained hashing.

Uniform rehashing

–yet another dynamic hashing scheme

Basic idea:

- Suppose $h(K) \in [0;1)$ –NB! A real number
- Look for key in bucket $\lfloor h(K) \cdot (m+1) \rfloor$
- Increase/decrease m by a **factor** $1 + \varepsilon$, where $\varepsilon > 0$ can be any constant – can be done by scanning the hash table.

See [Pagh03] for details on how to avoid real numbers.

B–tree vs hash indexes

- Hashing good to search given key, e.g.,
`SELECT * FROM R WHERE A = 5`
- Indexing (using B–trees) good for range searches, e.g.:
`SELECT * FROM R WHERE A > 5`
- More applications to come...

Hashing and range searching

Claim in book (p. 652):

"Hash tables do not support range queries"

– True, but they can be **used** to answer range queries in $O(1+Z/B)$ I/Os, where Z is the number of results. (Alstrup & Rauhe, 2001)

– Theoretical result, out of scope for ADBT.

Summary

- External hash tables support lookup of keys and updates in $O(1)$ I/Os, expected.
- The actual constant (typically 1, 2, or 3) is a major concern (compare to B-trees).
- Growth management:
Extendible hashing, linear hashing, uniform rehashing.