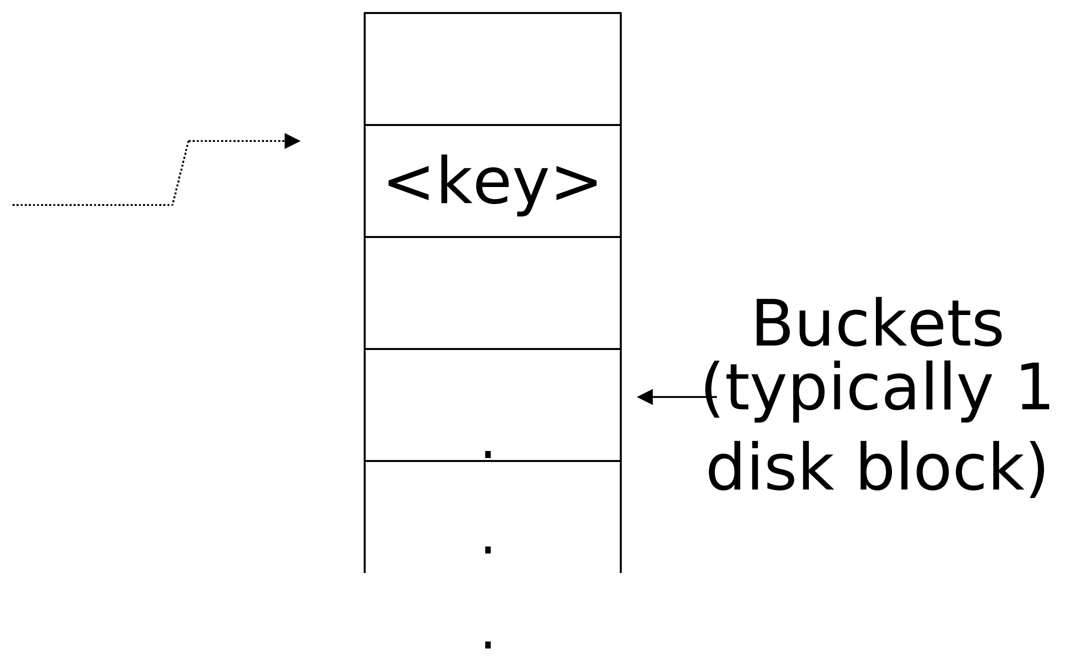
Outline/summary

- Conventional Indexes
  - Sparse vs. dense
  - Primary vs. secondary
- B trees
  - B+trees vs. indexed sequential
- Hashing schemes --> Next

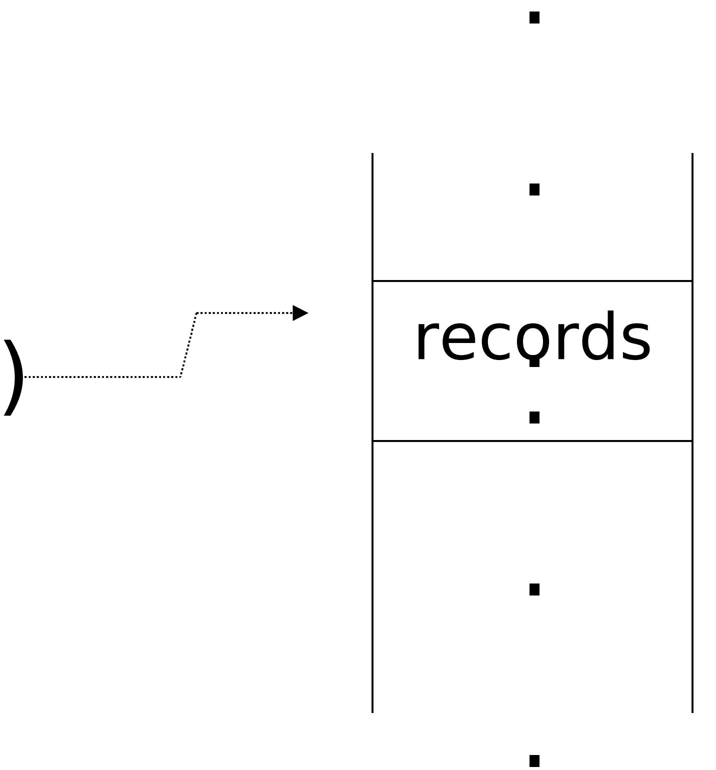
Hashing

key → h(key)



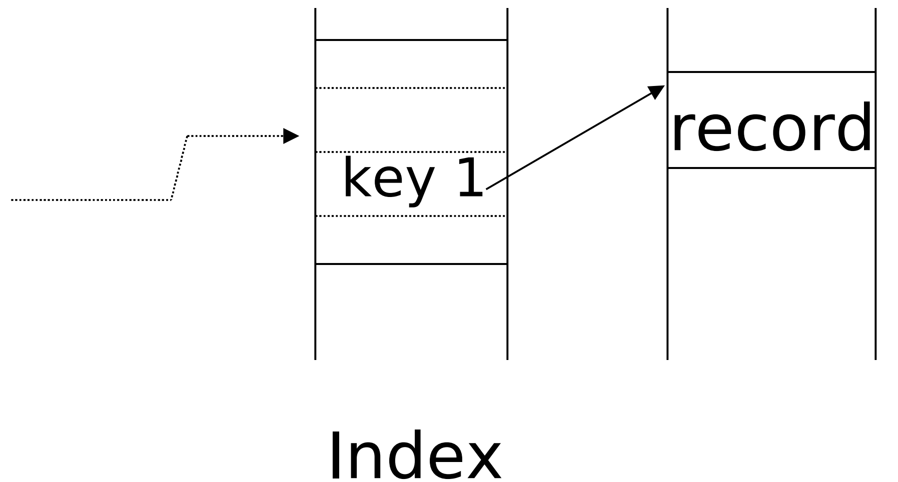
Two alternatives

(1) key $\rightarrow$ h(key)



Two alternatives

(2) key $\rightarrow$ 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 $x_1 + x_2 + \ldots\ldots x_n$
  - 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:

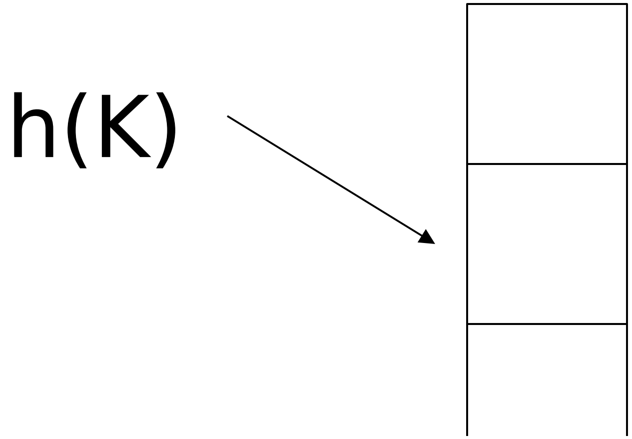
- ☞ Expected number of keys/bucket is the same for all buckets

Within a bucket:

- Do we keep keys sorted?

- Yes, if CPU time critical
  & Inserts/Deletes not too frequent

<u>Next:</u> example to illustrate
inserts, overflows, deletes



EXAMPLE 2 records/bucket

INSERT:

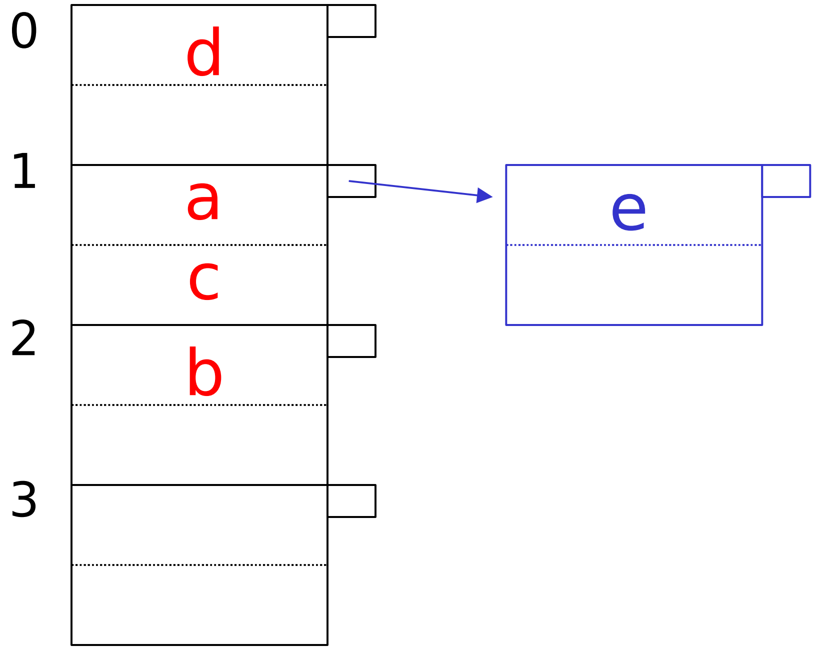
h(a) = 1

h(b) = 2

h(c) = 1

h(d) = 0

h(e) = 1



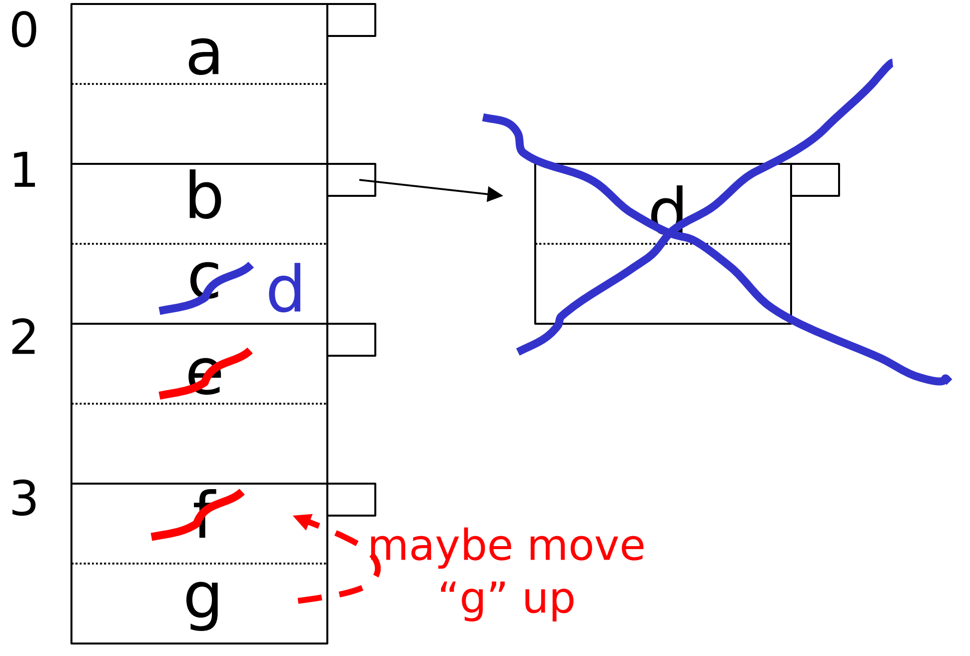
EXAMPLE: deletion

Delete:

e

f

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
  - Extensible
  - Linear

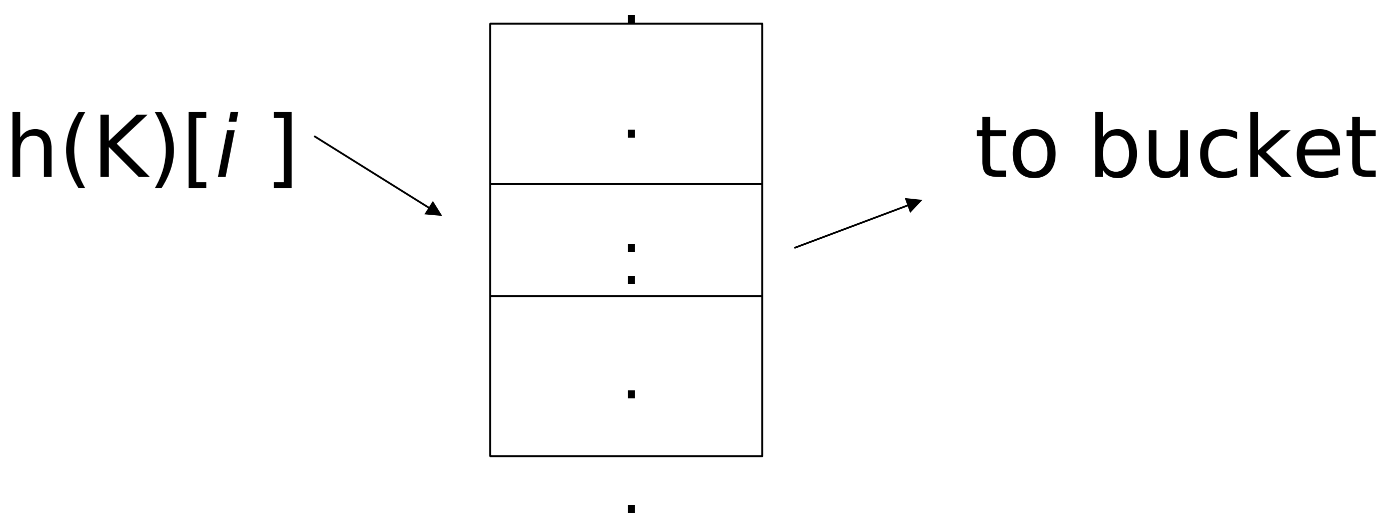
<u>Extensible hashing:</u> two ideas

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

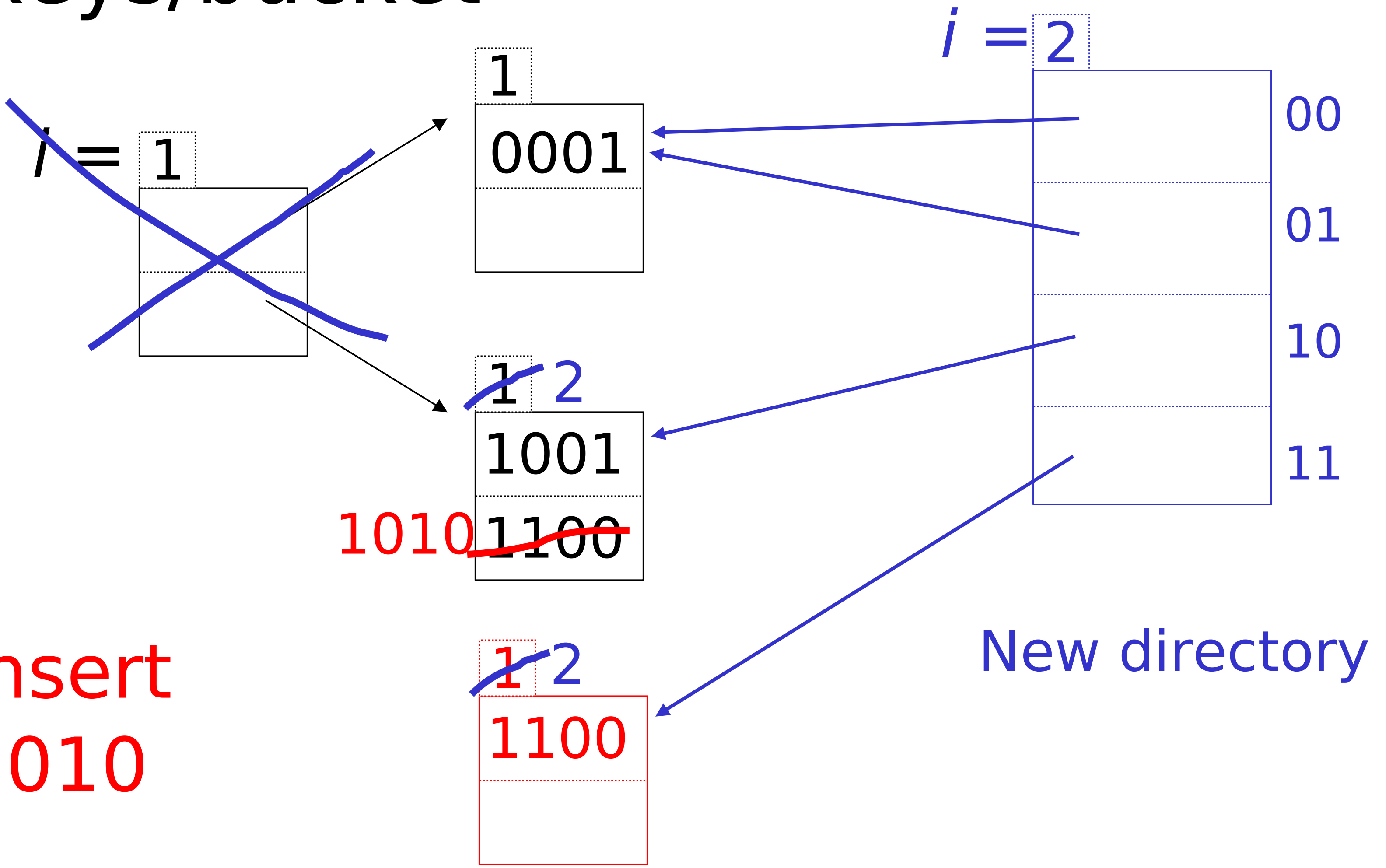
h(K) → 00110101

use *i* → grows over time….

(b) Use directory

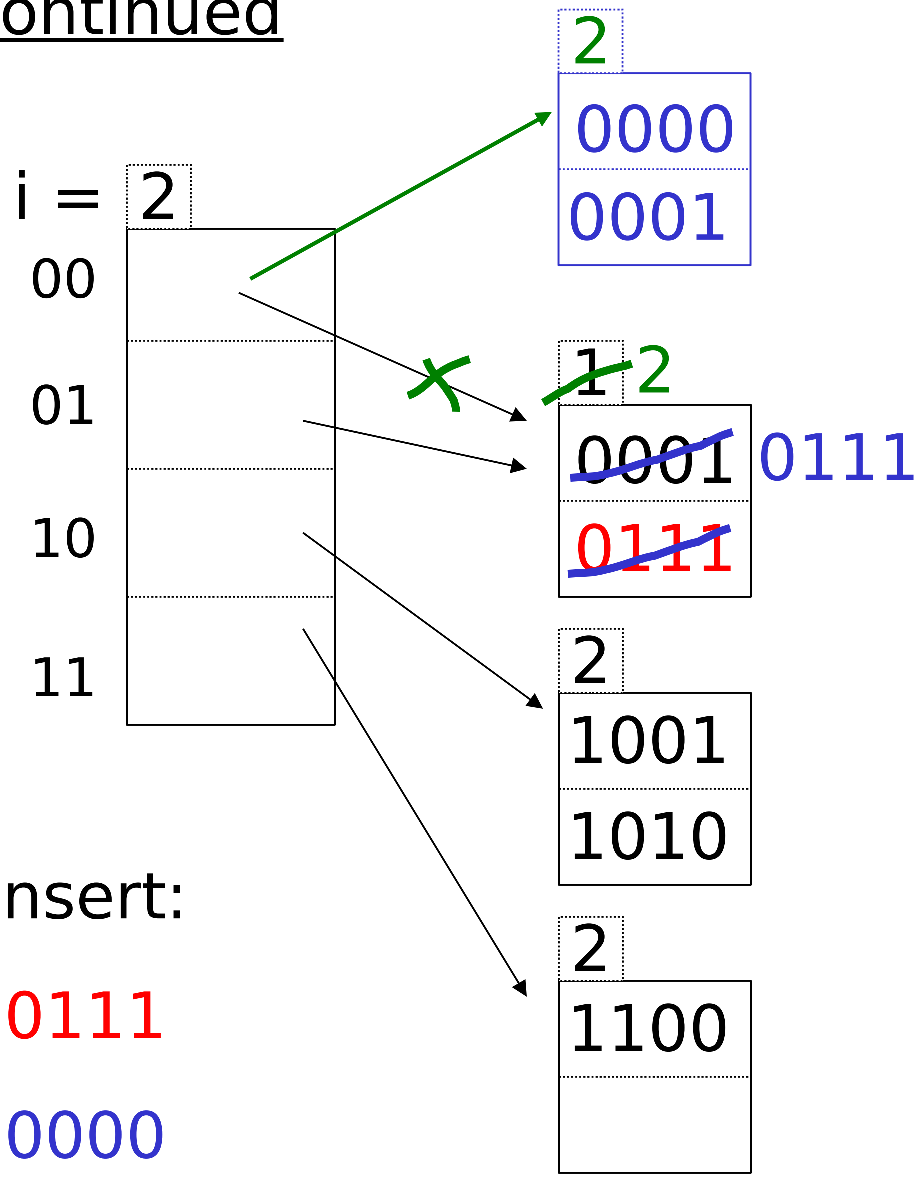


Example: h(k) is 4 bits; 2 keys/bucket



Insert 1010

Example continued

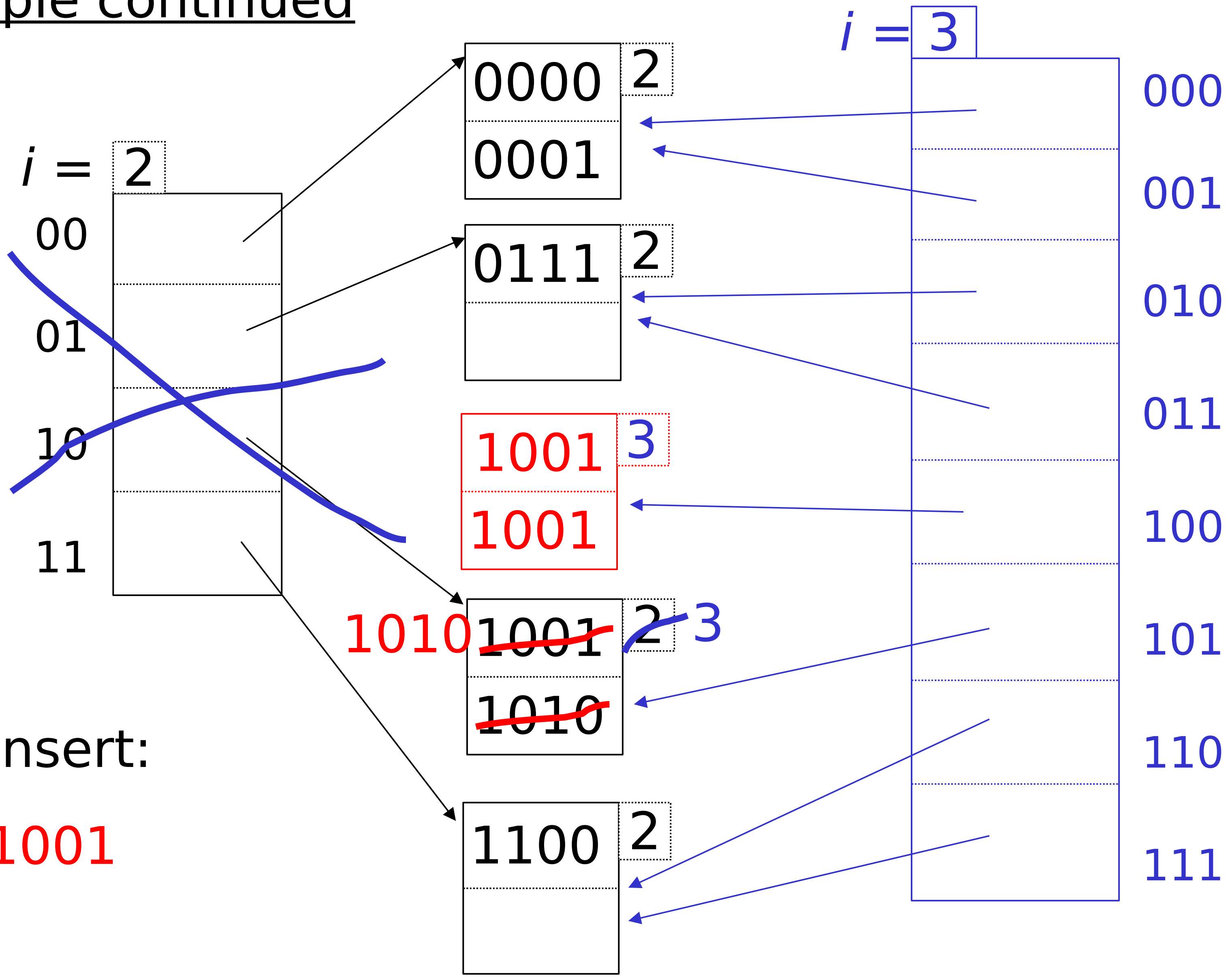


Insert:

0111

0000

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!

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

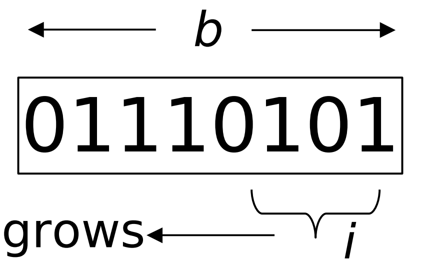
(Now it fits, now it does not)

Linear hashing

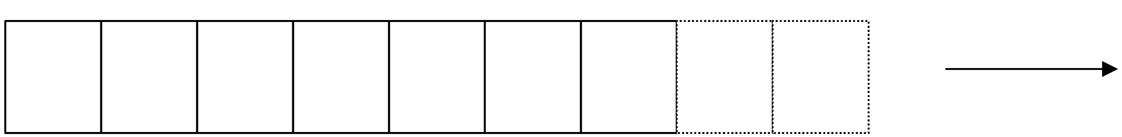
- Another dynamic hashing scheme

Two ideas:

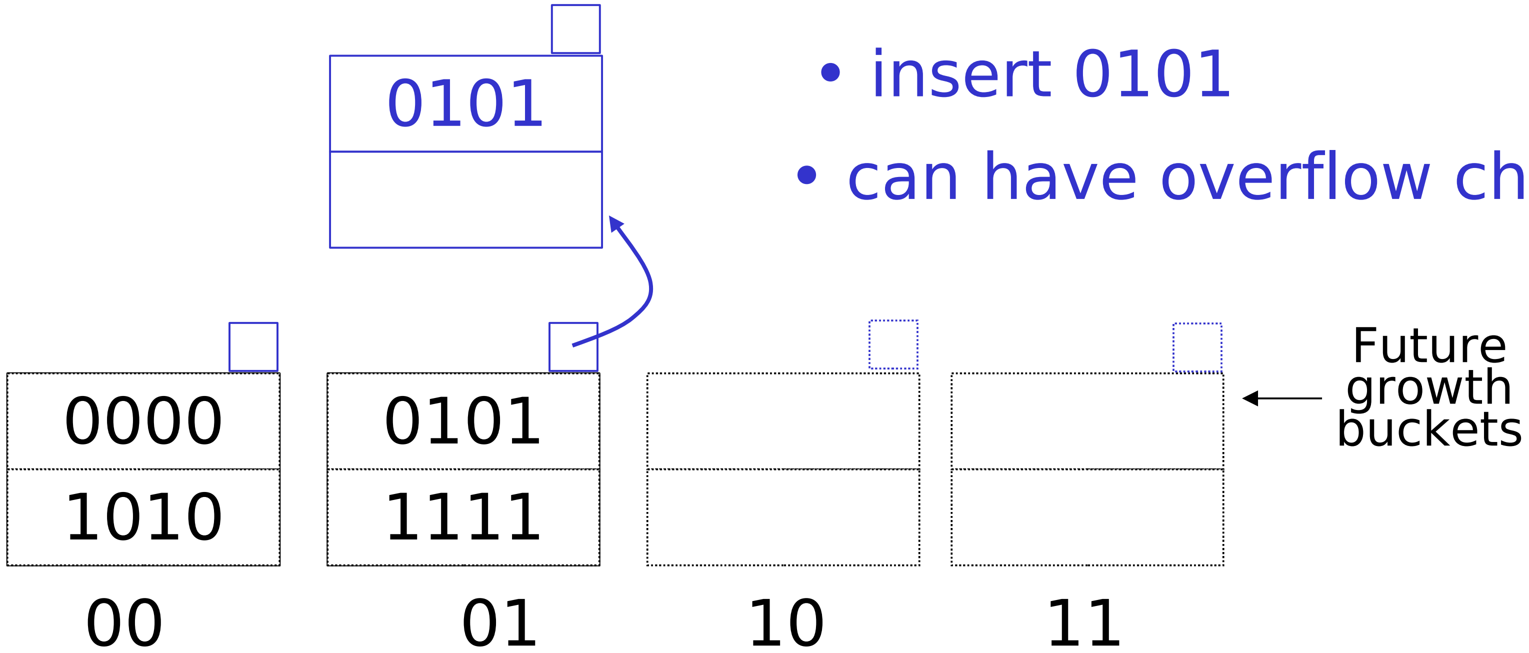
(a) Use *i* low order bits of hash



(b) Number n of buckets in use grows linearly



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



- insert 0101
- can have overflow chains!

n = 01 (number of buckets in use)

Rule

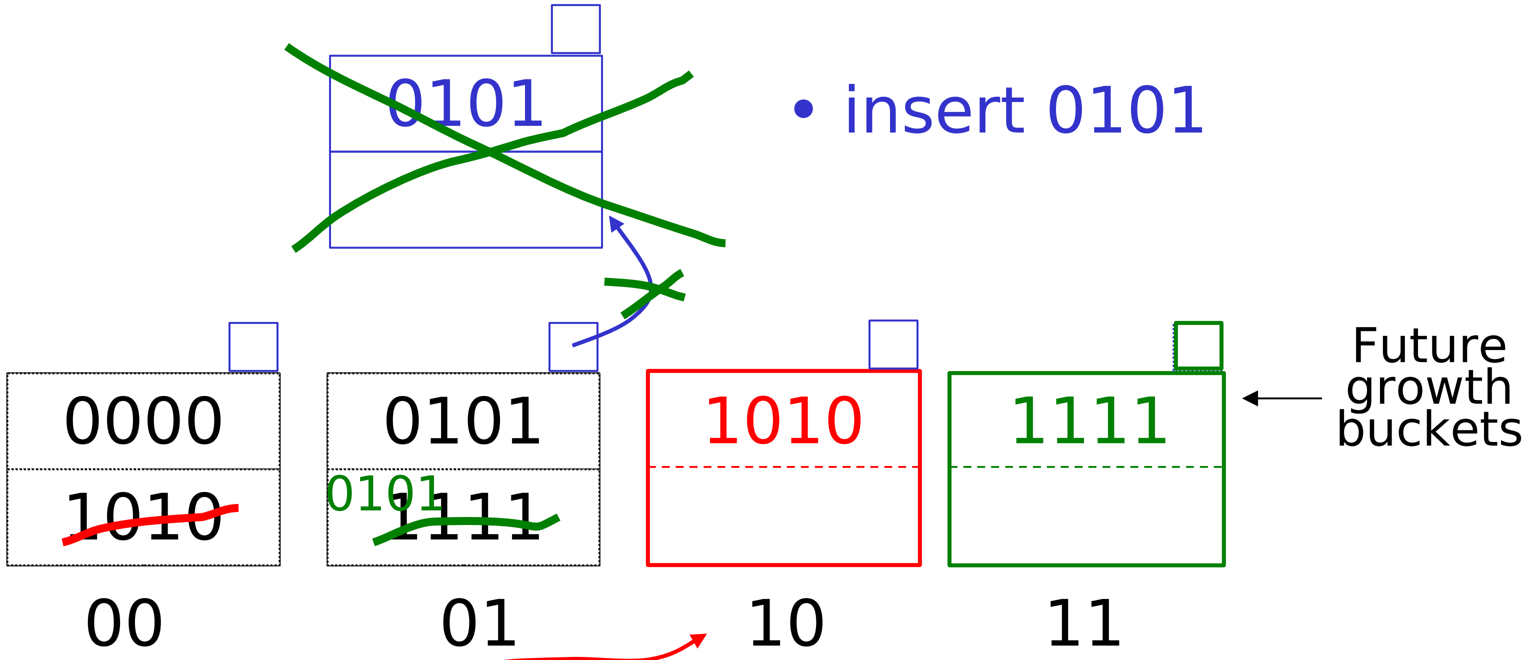
If h(k)[i] ≤ n, then

look at bucket h(k)[i]

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

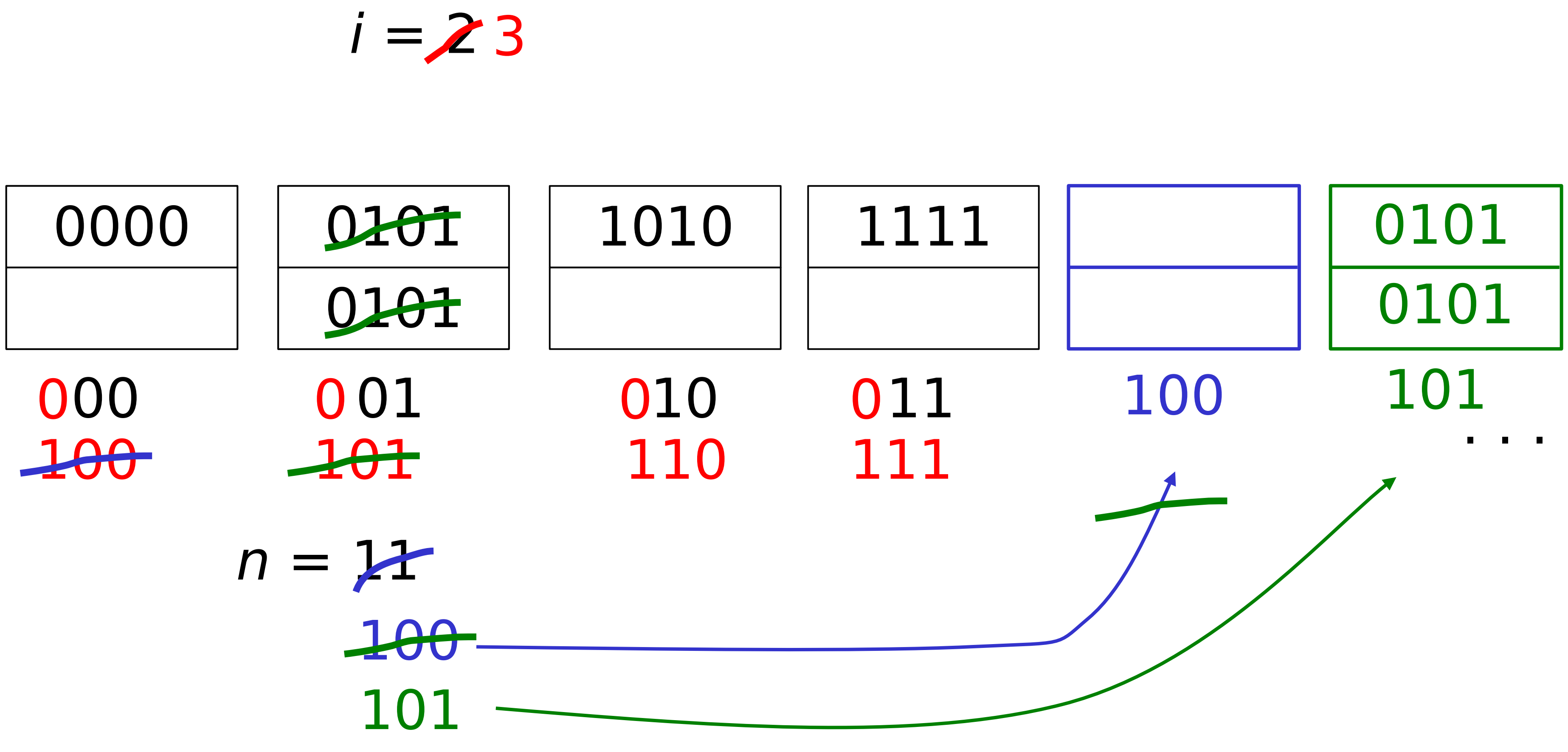
<u>Example</u> b=4 bits, i =2, 2 keys/bucket

- insert 0101



n = 01
10
11

Example Continued: How to grow beyond this?



- When do we expand file?

- Keep track of: $\frac{\#\ records}{\#\ buckets} = U$

- If U > threshold then increase *n*

  (and maybe *i*)

Summary Linear Hashing

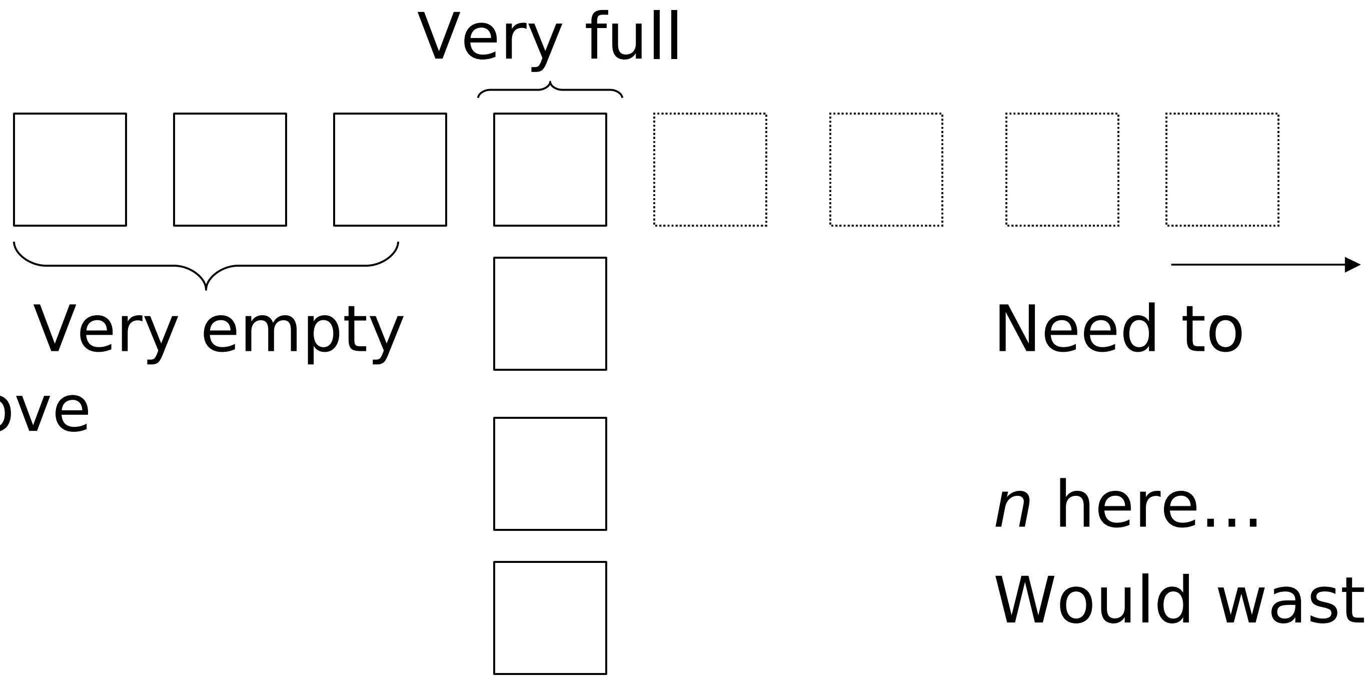
(+) Can handle growing files

- with less wasted space
- with no full reorganizations

(+) No indirection like extensible hashing

(-) Can still have overflow chains

Example: BAD CASE



Need to

move

n here…

Would waste space…

Summary

Hashing

- How it works
- Dynamic hashing
  - Extensible
  - Linear

B+trees vs Hashing

- Hashing good for probes given key

    e.g., SELECT …

    FROM R

    WHERE R.A = 5

B+Trees vs Hashing

- INDEXING (Including B Trees) good for

    Range Searches:

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