

Advanced Database Technology  
Anna Östlin Pagh and Rasmus Pagh  
IT University of Copenhagen  
Spring 2004

March 4, 2004

# INDEXING II

Lecture based on [GUW, 13.3-13.4] and [Pagh03, 3.0-3.2+4]

Slides based on  
**Notes 04: Indexing**  
**Notes 05: Hashing and more**  
for Stanford CS 245, fall 2002  
by Hector Garcia-Molina

# Today

- Recap of indexes
- B-trees
- Analysis of B-trees
- B-tree variants and extensions
- Hash indexes

# Why indexing?

- Support more efficiently queries like:  
`SELECT * FROM R WHERE a=11`  
`SELECT * FROM R WHERE 0<= b and b<42`
- Indexing an attribute (or set of attributes) speeds up finding tuples with specific values.

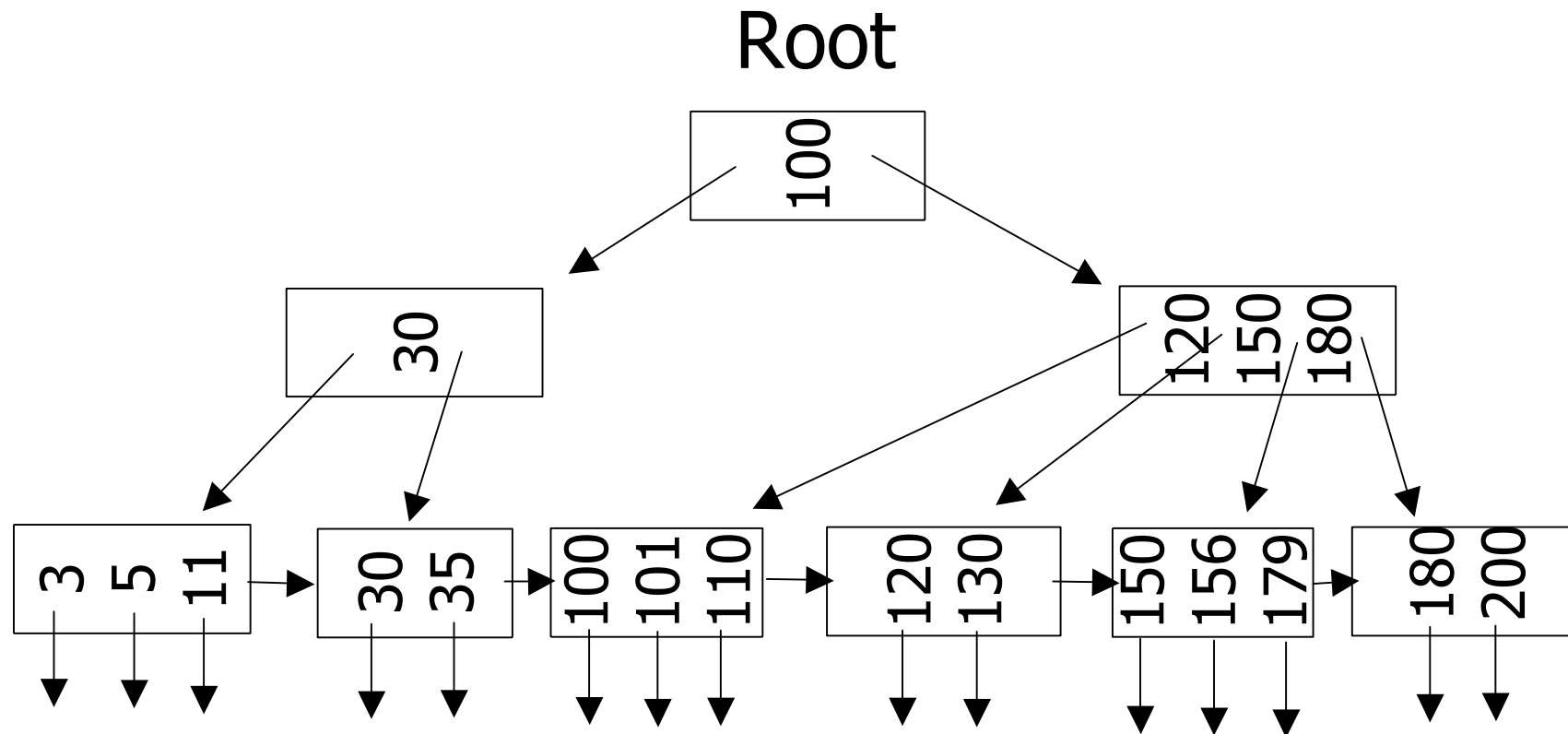
# Indexes in last lecture

- Dense indexes (primary or secondary)
- Sparse indexes (always primary)
- Multi-level indexes
- **Updates** (inserting or deleting a key) caused problems

# B-trees

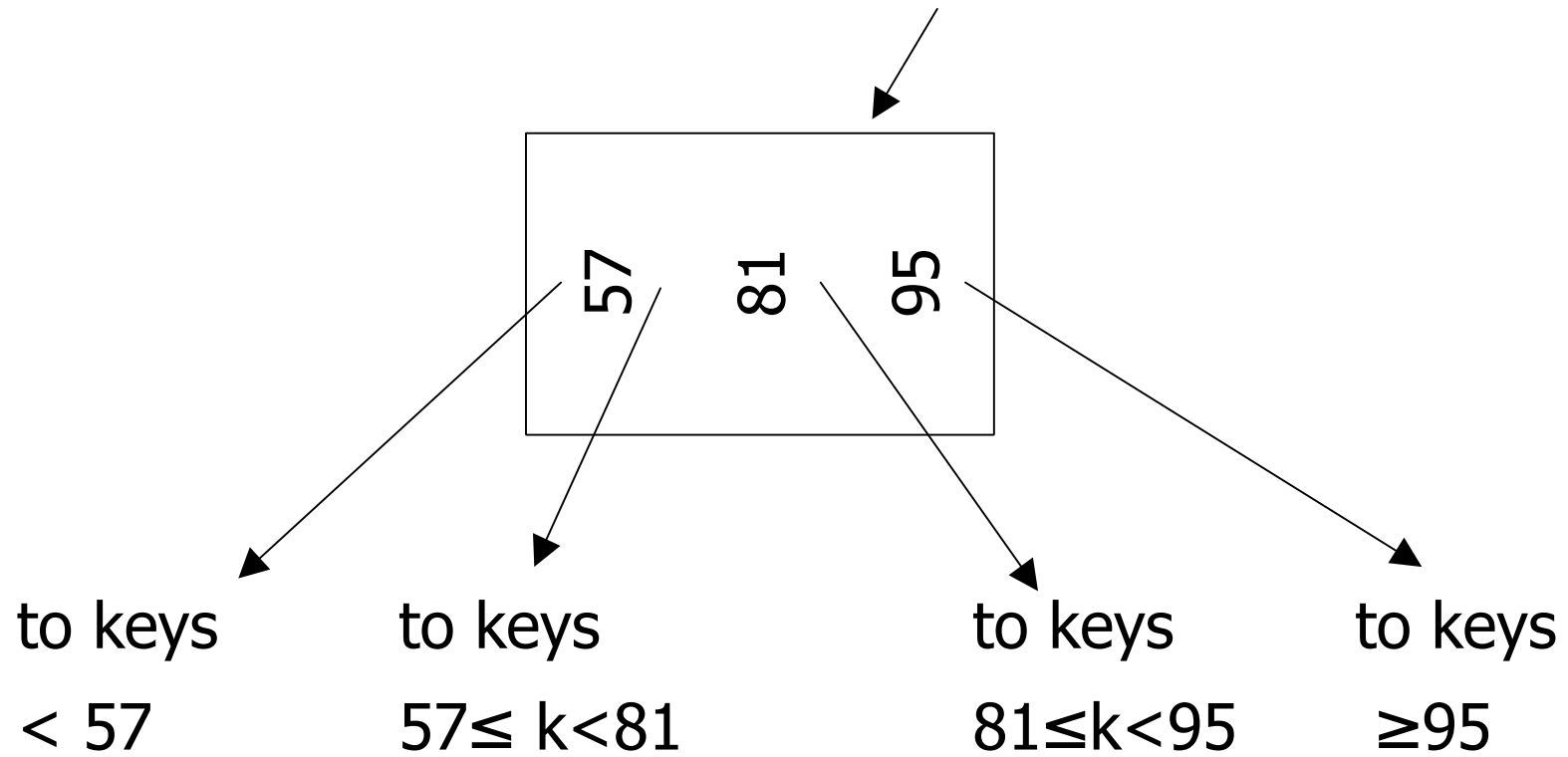
- Can be seen as a general form of multi-level indexes.
- Generalize usual (binary) search trees.  
*(Do you remember?)*
- Allow efficient insertions and deletions at the expense of using slightly more space.
- Popular variant: B<sup>+</sup>-tree

# B<sup>+</sup>-tree Example

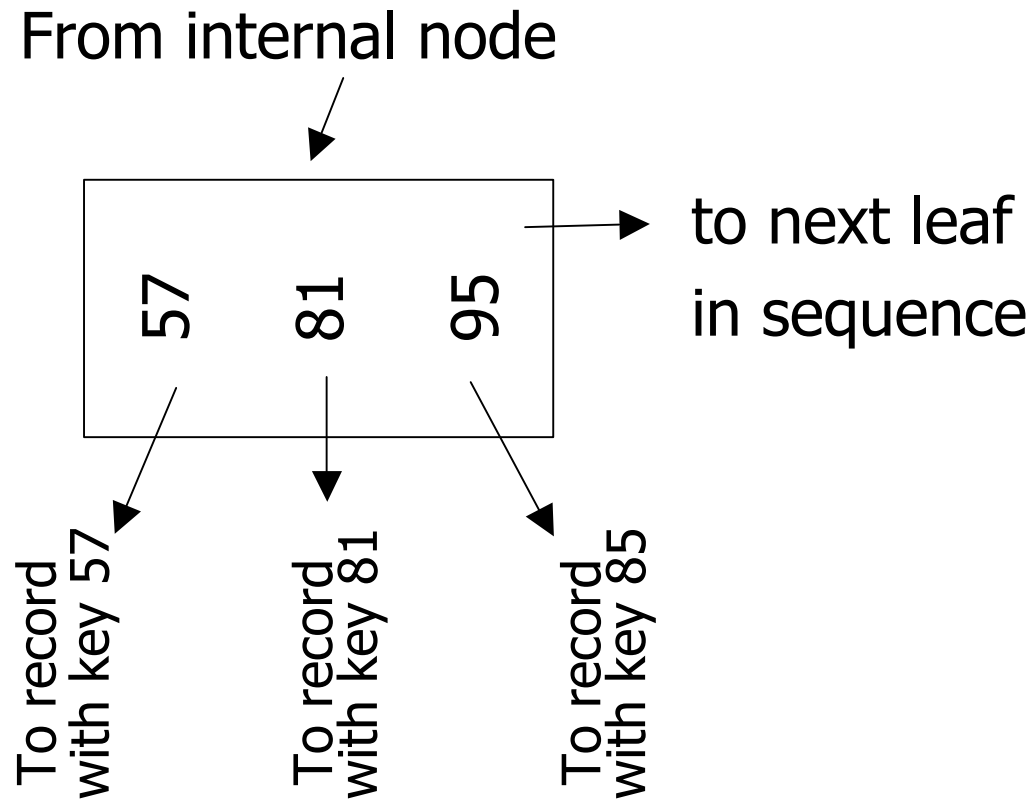


***Each node stored in one disk block***

# Sample internal node

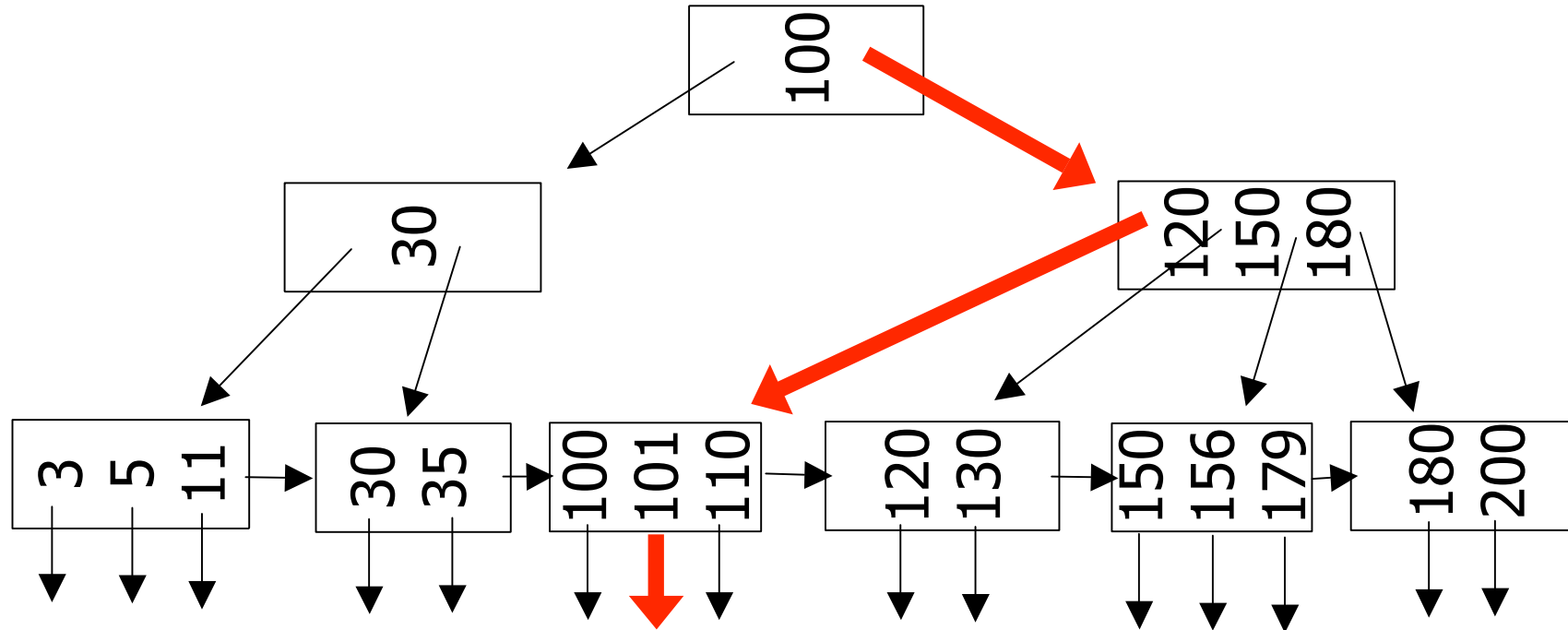


## Sample leaf node:



***Alternative: Records in leaves***

# Searching a B<sup>+</sup>-tree

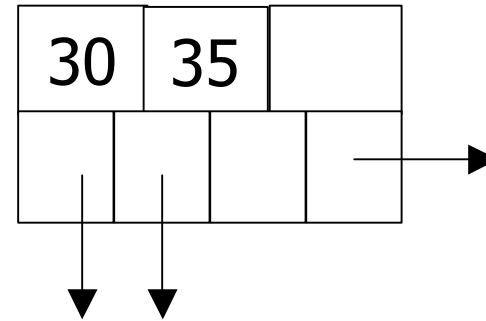
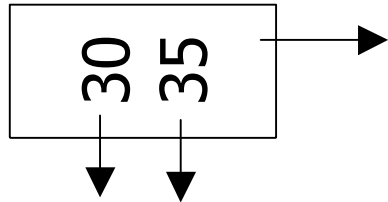


Above: Search path for tuple with key 101.

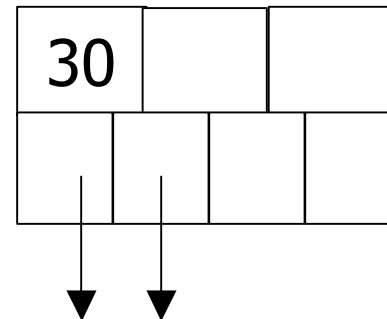
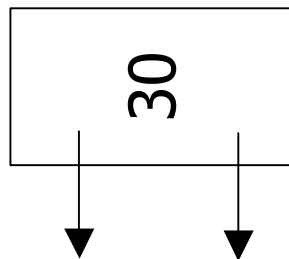
**Question:** How does one search for a **range** of keys?

# In textbook's notation

Leaf:



Internal node:



## B<sup>+</sup>-tree invariants on nodes

- Suppose a node (stored in a block) has space for  $n$  keys and  $n+1$  pointers.
- Don't want block to be too empty: Should have at least  $\lfloor (n+1)/2 \rfloor$  non-null pointers.
- Exception: The root, which may have only 2 non-null pointers.

## Other B<sup>+</sup>-tree invariants

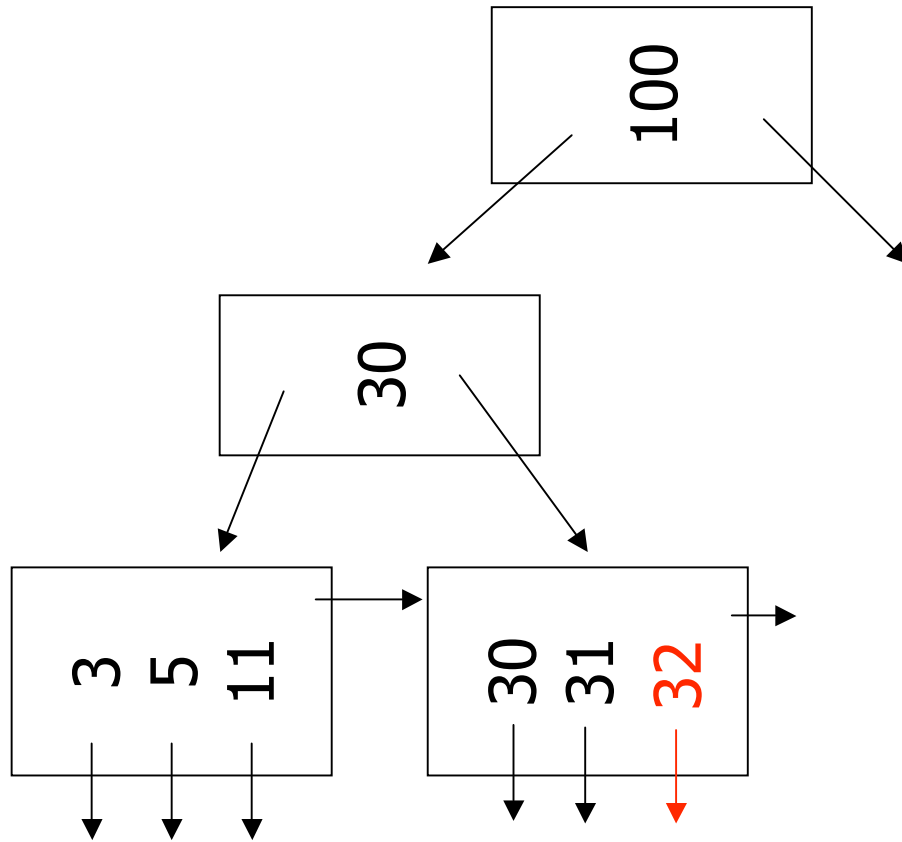
- (1) All leaves at same lowest level  
(perfectly balanced tree)
- (2) Pointers in leaves point to records  
except for *sequence pointer*

# Insertion into B<sup>+</sup>-tree

- (a) simple case
  - space available in leaf
- (b) leaf overflow
- (c) non-leaf overflow
- (d) new root

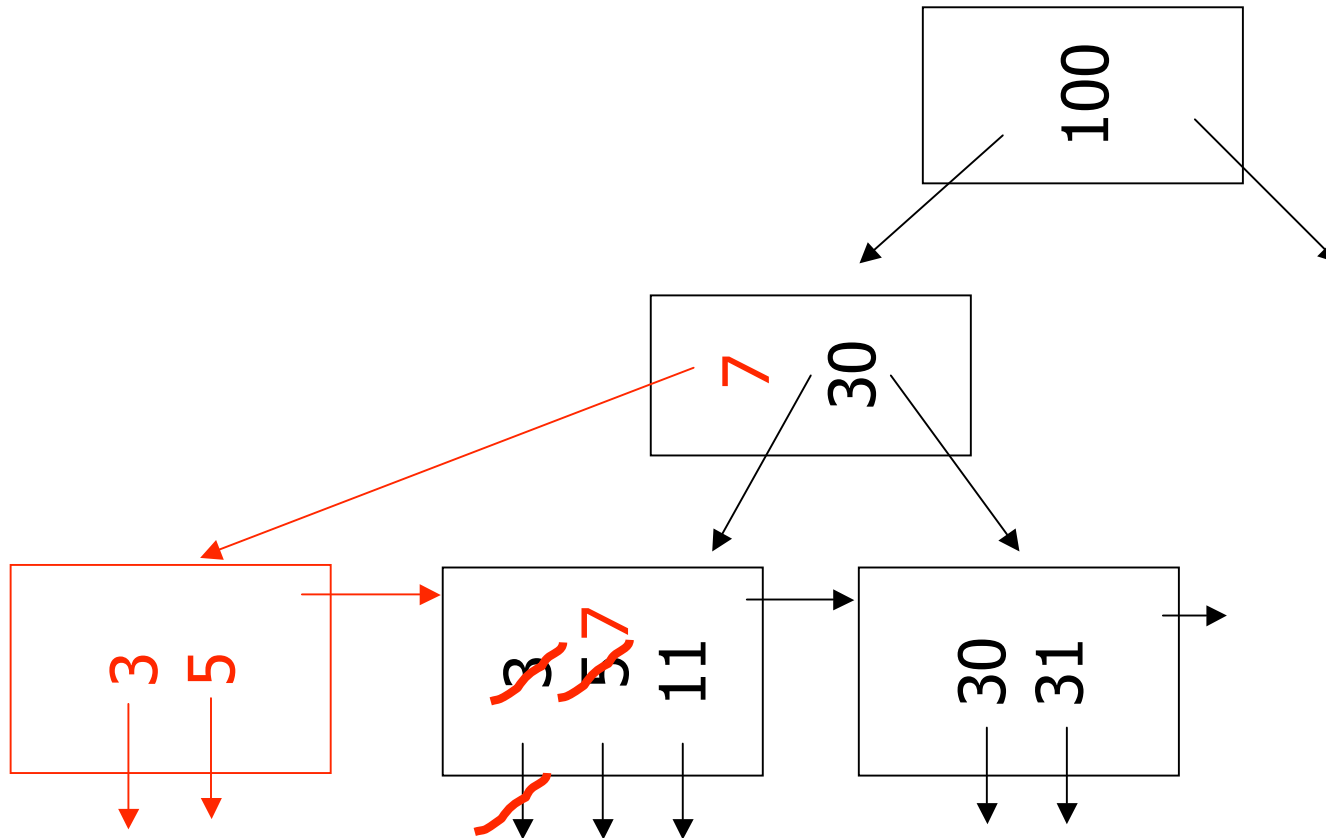
(a) Insert key = 32

n=3



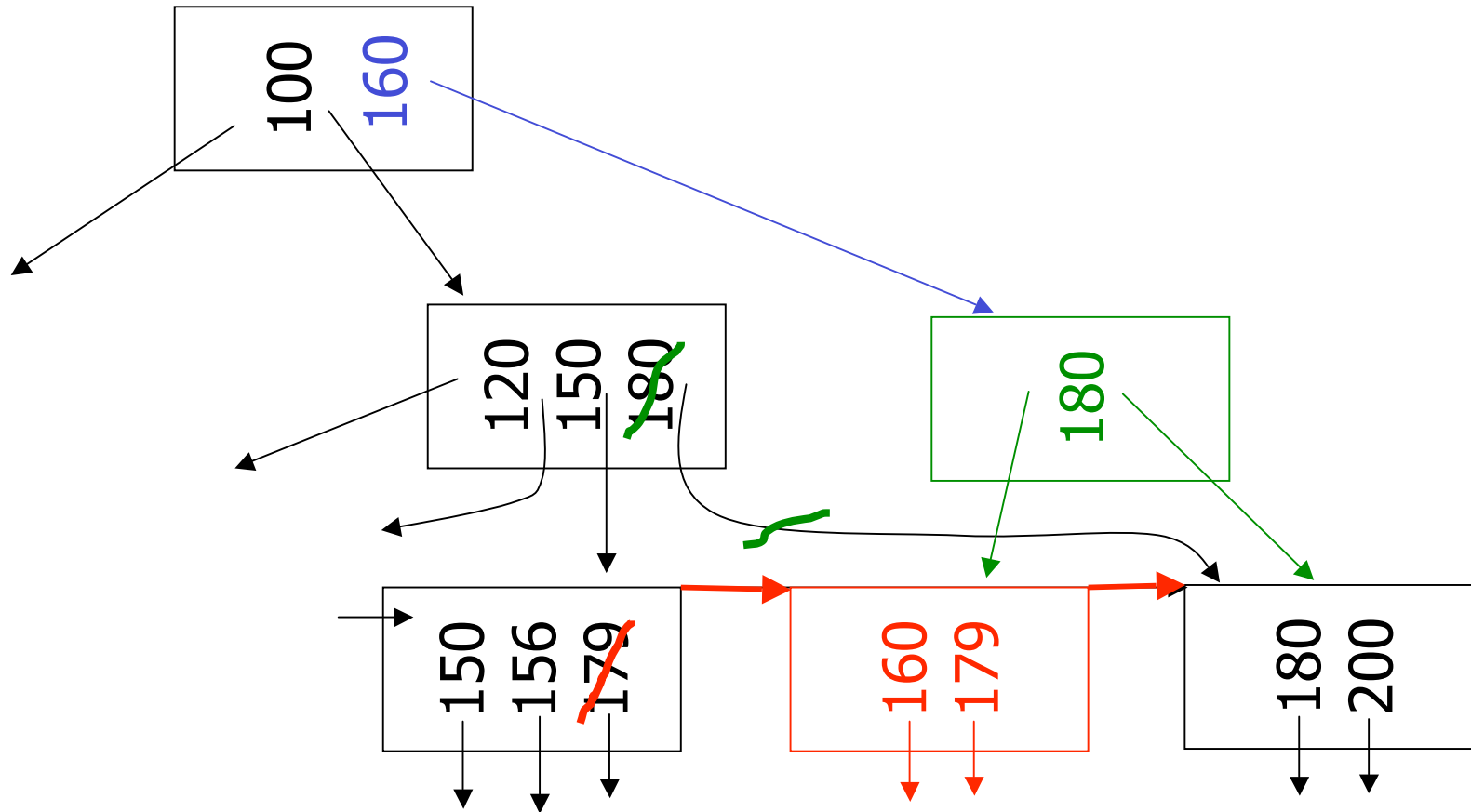
(b) Insert key = 7

n=3



(c) Insert key = 160

n=3





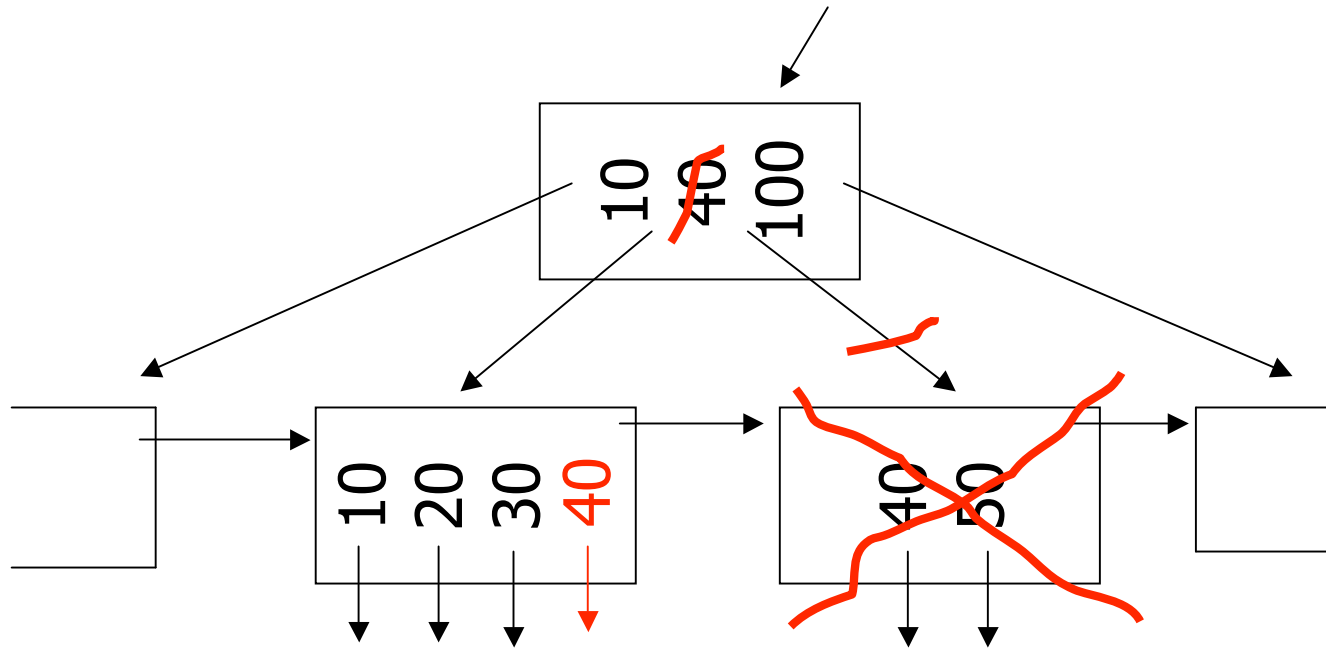
# Deletion from B<sup>+</sup>-tree

- (a) Simple case - no example
- (b) Coalesce with neighbour (sibling)
- (c) Re-distribute keys
- (d) Cases (b) or (c) at non-leaf

## (b) Coalesce with sibling

- Delete 50

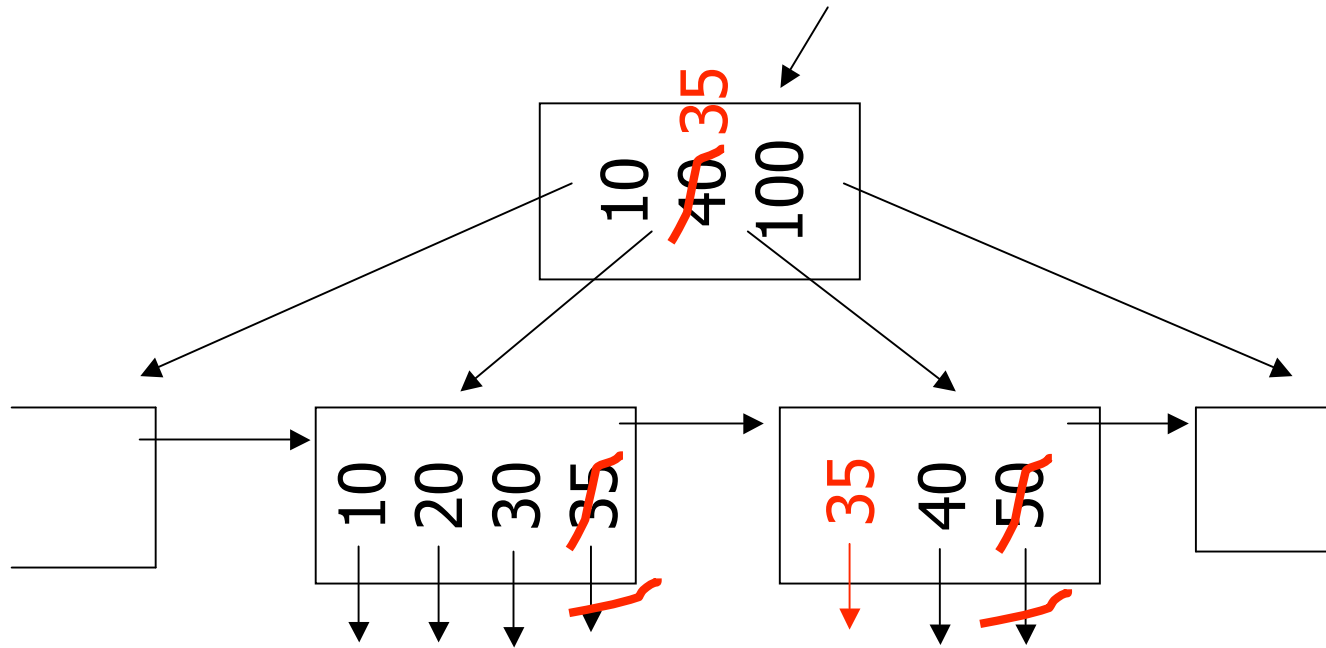
n=4



# (c) Redistribute keys

- Delete 50

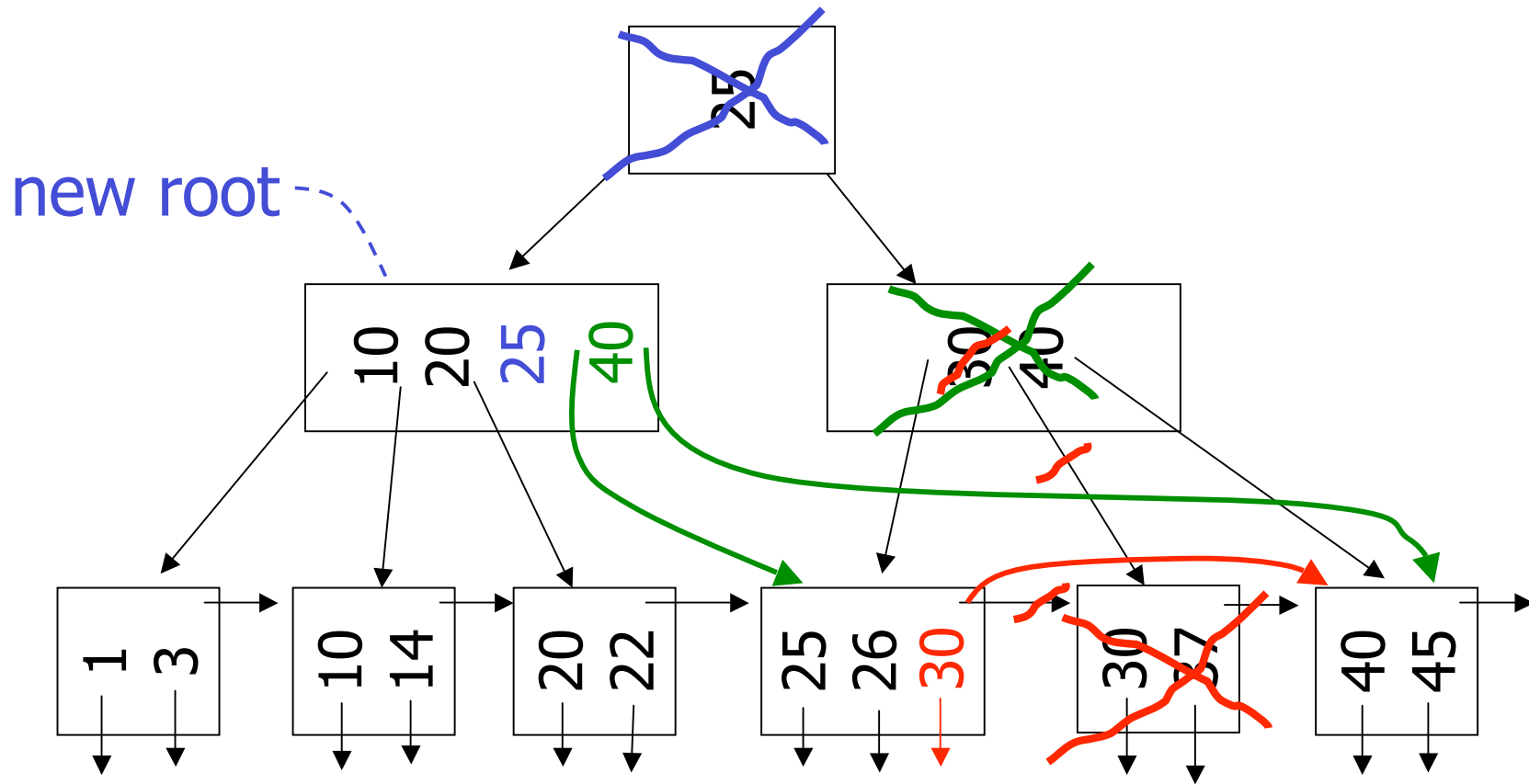
n=4



# (d) Non-leaf coalesce

- Delete 37

n=4



# Alternative B<sup>+</sup>-tree deletion

- In practice, coalescing is often **not** implemented (hard, and often not worth it)
- An alternative is to use tombstones.
- Periodic **global rebuilding** may be used to remove tombstones when they start taking too much space.

## Problem session: Analysis of B<sup>+</sup>-trees

- What is the height of a B<sup>+</sup>-tree with **N** leaves and room for **n** pointers in a node?
- What is the worst case I/O cost of
  - Searching?
  - Inserting and deleting?

## B<sup>+</sup>-tree summary

- Height  $\leq 1 + \log_{n/2} N$ , typically 3 or 4.
- Best search time we could hope for!  
(To be shown in exercises.)
- If keeping top node(s) in memory, the number of I/Os can be reduced.
- Updates: Same cost as search, except for **rebalancing**.

## More on rebalancing

- The book claims (on page 645):  
"It will be a rare event that calls for splitting or merging of blocks".
- This is true (in particular at the top levels), but a little hard to see.
- Easier seen for **weight-balanced B-trees**.

# Weight-balanced B-trees

(based on [Pagh03], where  $n$  corresponds to  $B/2$ )

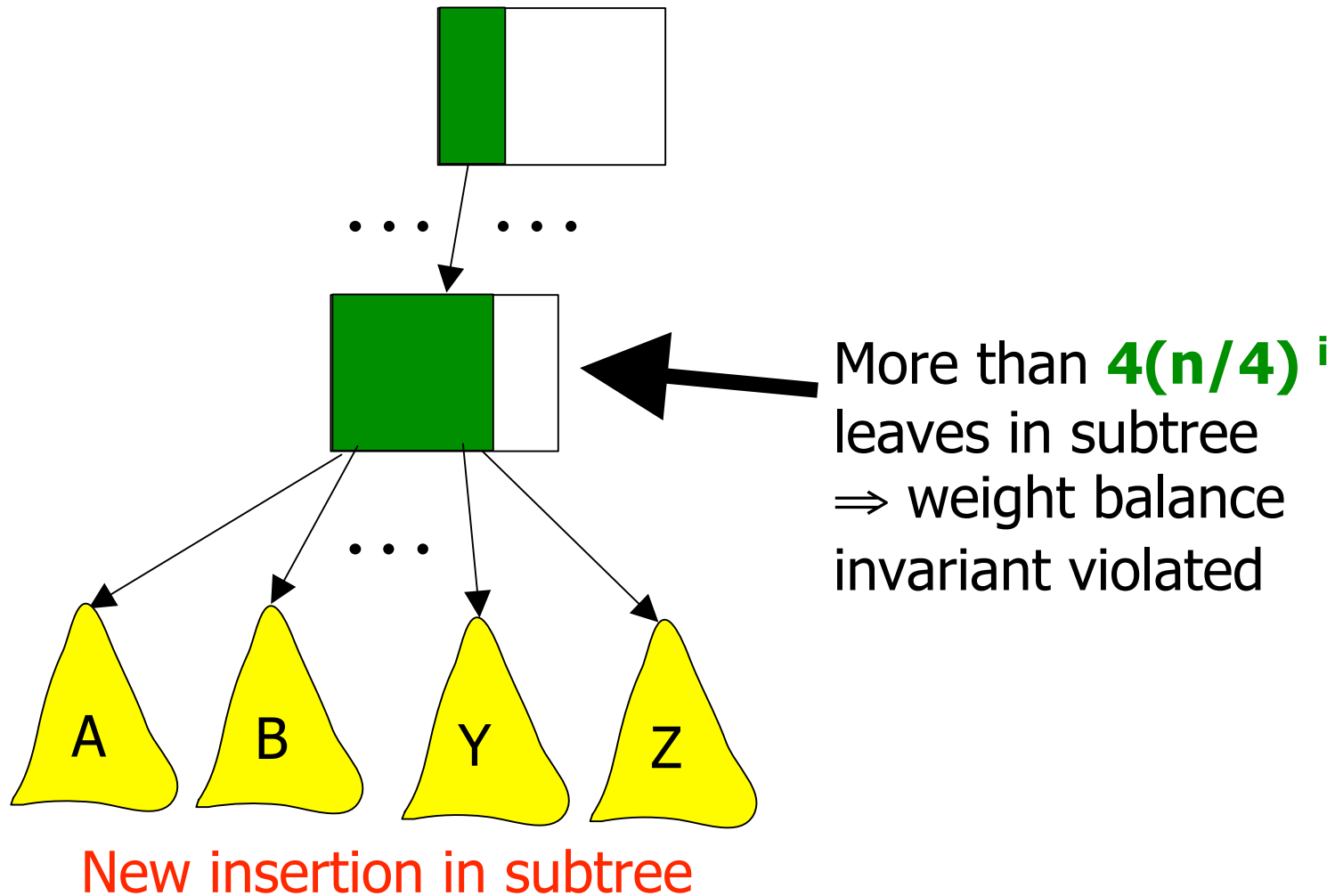
- Remove the  $B^+$ -tree invariant:  
There must be  $\lfloor (n+1)/2 \rfloor$  non-null pointers in a node.
- Add new **weight** invariant:  
A node at height  $i$  must have **weight** (number of leaves in the subtree below) that is between  $(n/4)^i$  and  $4(n/4)^i$ .  
(Again, the root is an exception.)

# Weight-balanced B-trees

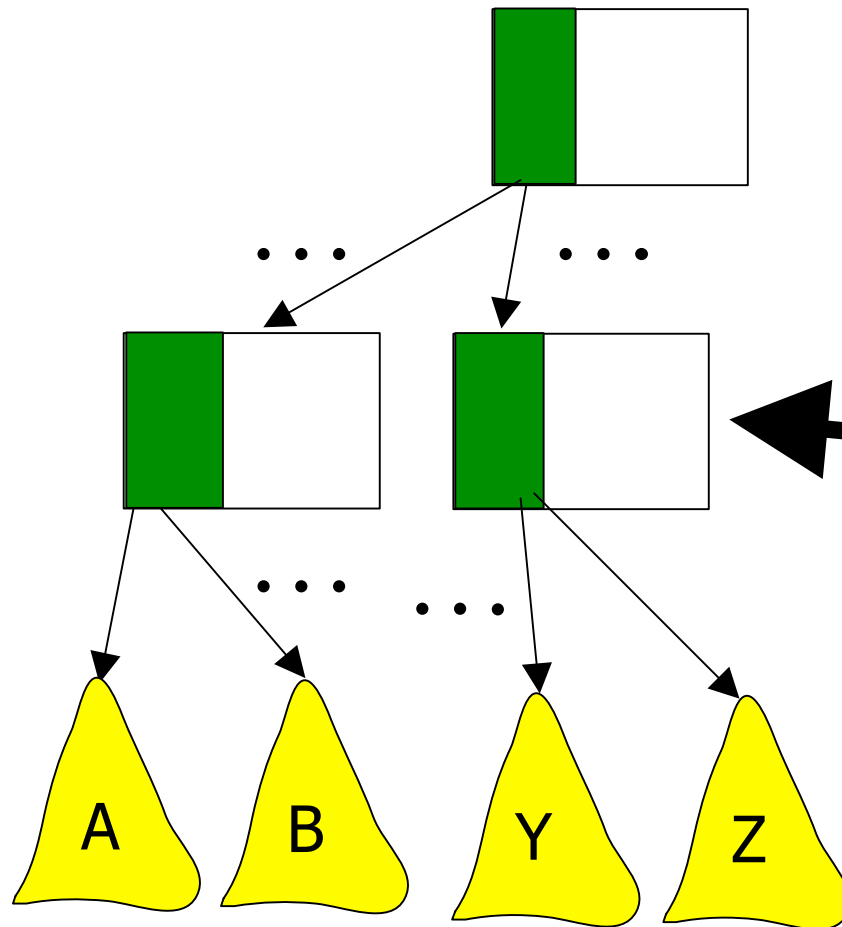
Consequences of the **weight** invariant:

- Tree height is  $\leq 1 + \log_{n/4} N$  (almost same)
- A node at height  $i$  with weight, e.g.,  $2(n/4)^i$  will not need rebalancing until there have been at least  $(n/4)^i$  updates in its subtree. **(Why?)**

# Rebalancing weight



# Rebalancing weight



Node is split into two nodes of weight around  $2(n/4)^i$ , i.e., **far** from violating the invariant (details in [Pagh03])

# Weight-balanced B-trees

## Summary of properties

- Deletions similar to insertions (or: use tombstones and global rebuilding).
- Search in time  $O(\log_n N)$ .
- A node at height  $i$  is rebalanced (costing  $O(1)$  I/Os) once for every  $\Omega((n/4)^i)$  updates in its subtree.

## Problem session

- In internal memory, sorting can be done in  $O(n \log n)$  time by inserting the keys into a balanced search tree.
- What is the time complexity of this strategy using B-trees?
- How does it compare with the optimal sorting algorithm we saw earlier?

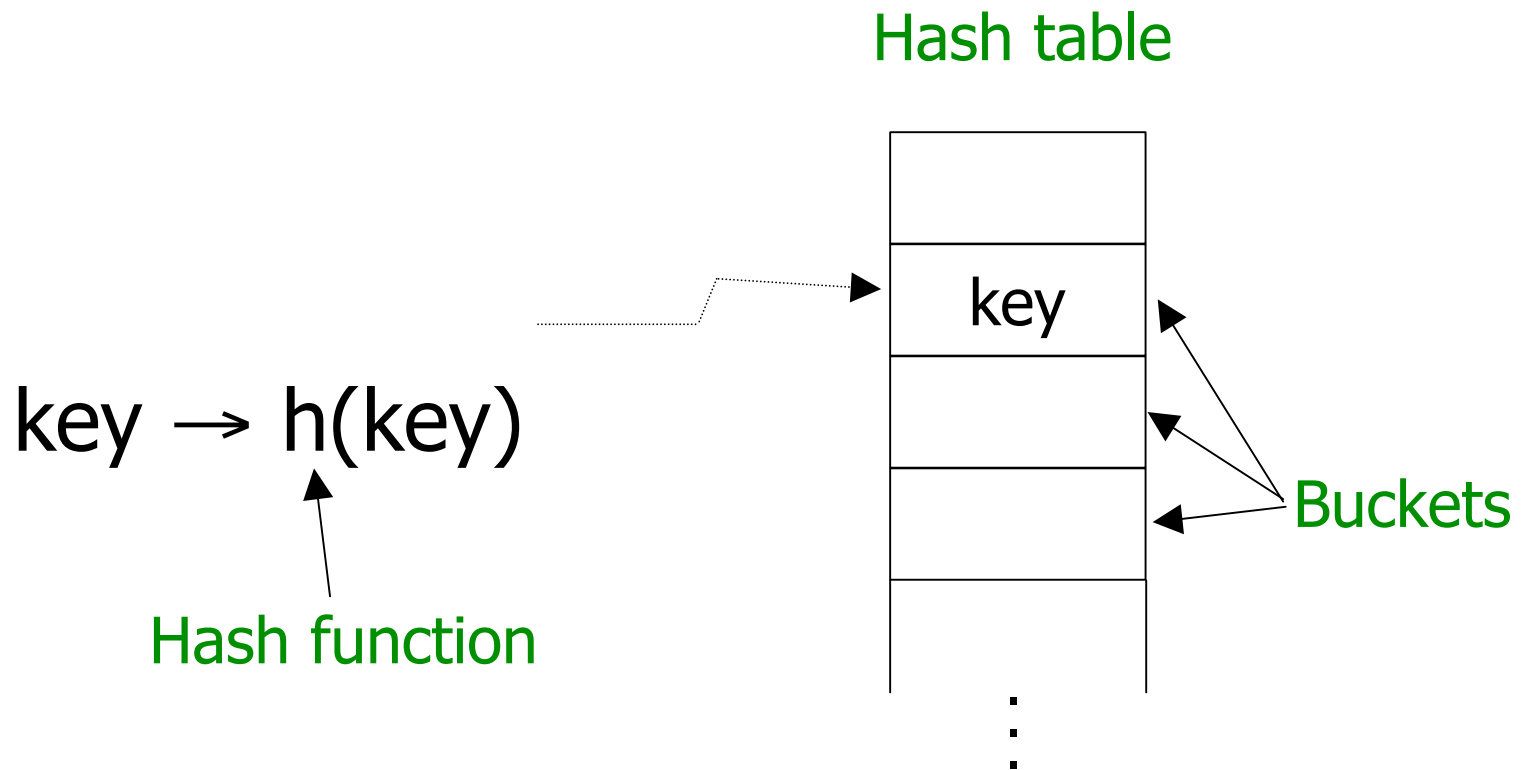
# Other kinds of B-trees

- **String B-trees:** Fast searches even if keys span many blocks. (April 3 lecture.)
- **Persistent B-trees:** Make searches in any previous version of the tree, e.g. "find x at time t". The time for a search is  $O(\log_B N)$ , where N is the **total** number of keys inserted in the tree.

## Next: Hash indexes

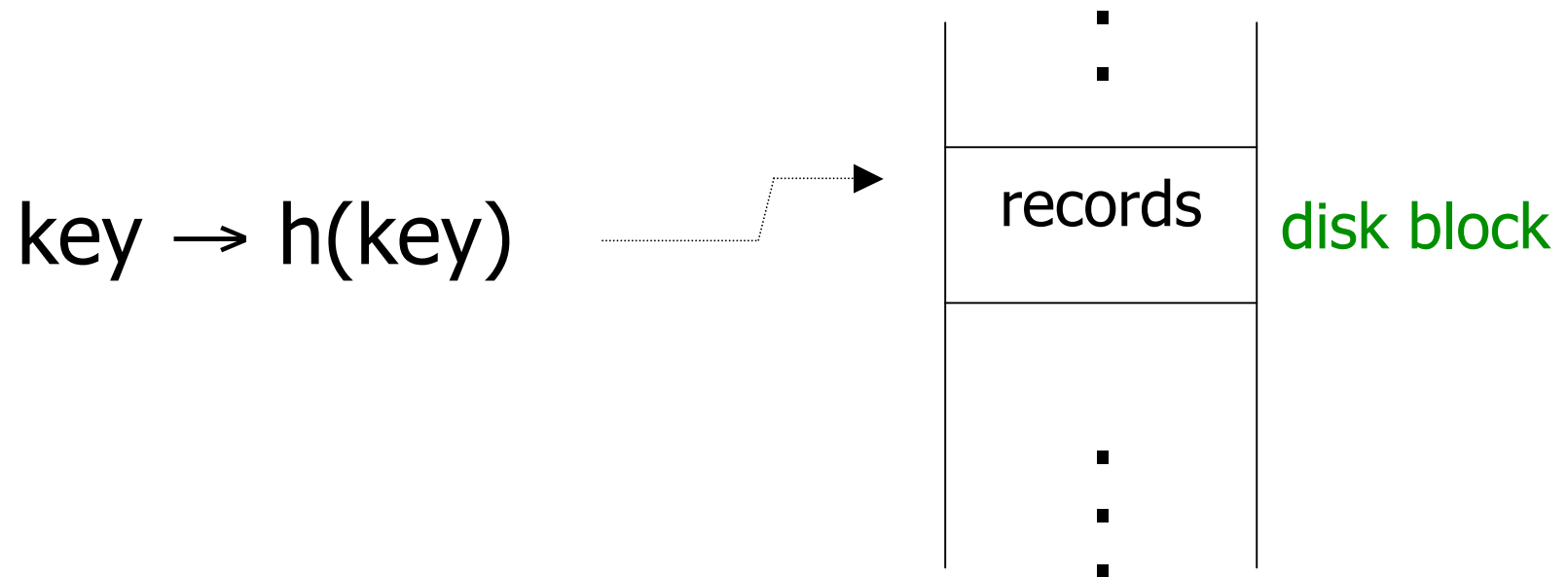
- You may recall that in internal memory, **hashing** can be used to quickly locate a specific key.
- The same technique can be used on external memory.
- However, advantage over search trees is smaller than internally.

# Hashing in a nutshell



Typical implementation of buckets: Linked lists

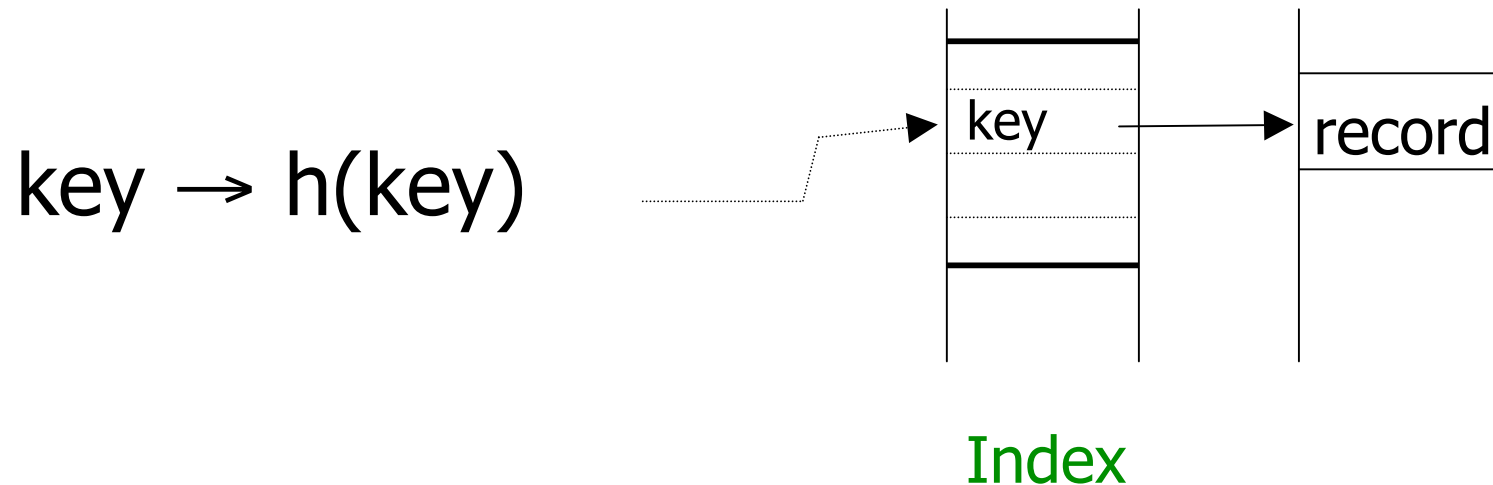
# Hashing as primary index



**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 =  $x_1 x_2 \dots x_n$ , n byte character string:  
 $h(\text{Key}) = (x_1 + x_2 + \dots + x_n) \bmod b$
- Key is an integer:  $h(\text{Key}) = \text{Key} \bmod b$

## **SHORT 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:

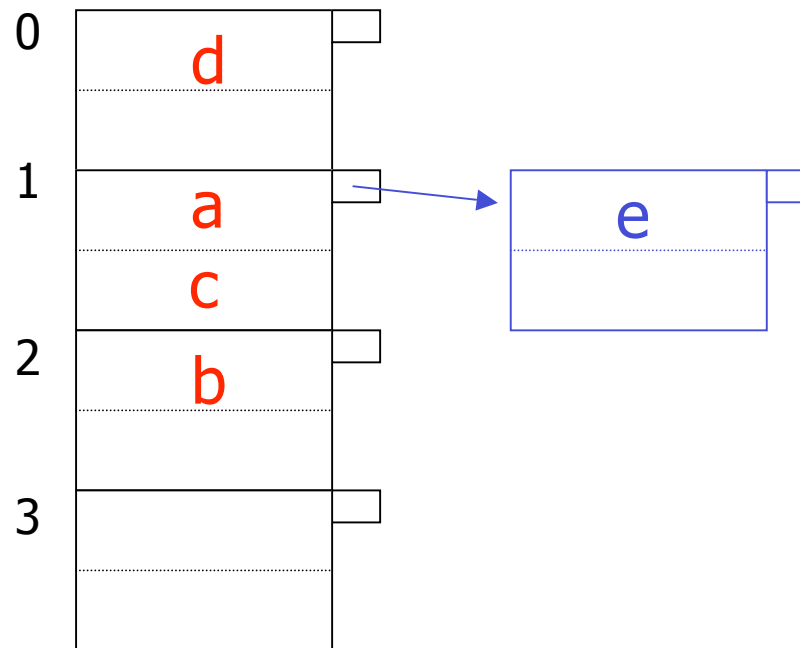
$h(a) = 1$

$h(b) = 2$

$h(c) = 1$

$h(d) = 0$

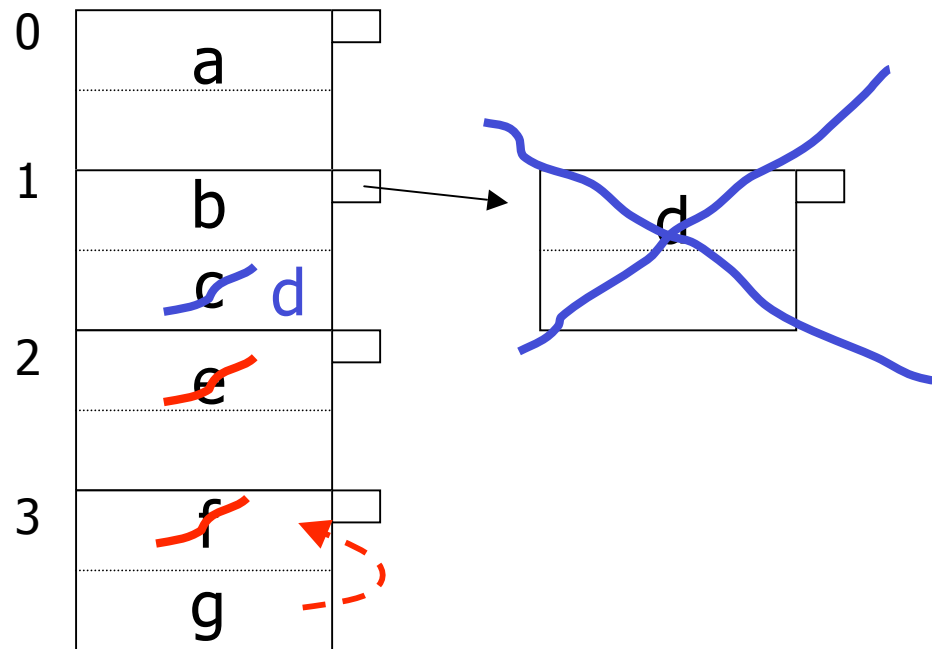
$h(e) = 1$



# Deletions

Delete:

e  
f  
c



## Analysis - external chained hashing (assuming truly random hash functions)

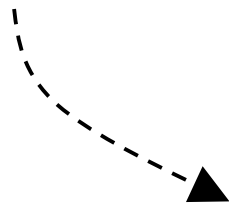
- $N$  keys inserted, each block (bucket) in the hash table can hold  $B$  keys.
- Suppose the hash table has size  $N/\alpha B$ , i.e., "is a fraction  $\alpha$  full".
- Expected number of overflow blocks:  
 $(1-\alpha)^{-2} \cdot 2^{-\Omega(B)} N$  (proof omitted!)
- Good to have many keys in each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

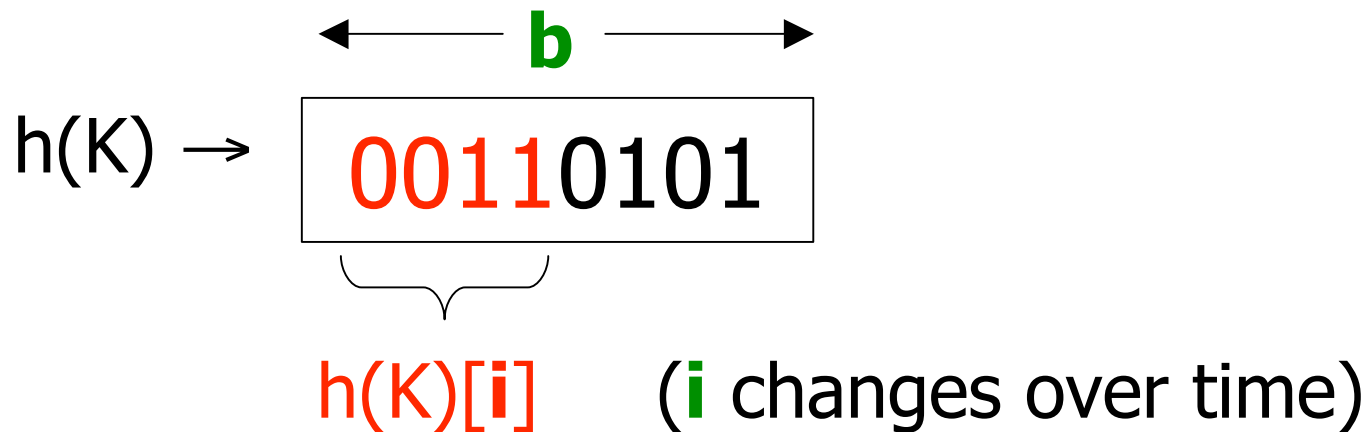


- Extendible hashing
- Linear hashing
- Uniform rehashing

# Extendible hashing

Two ideas:

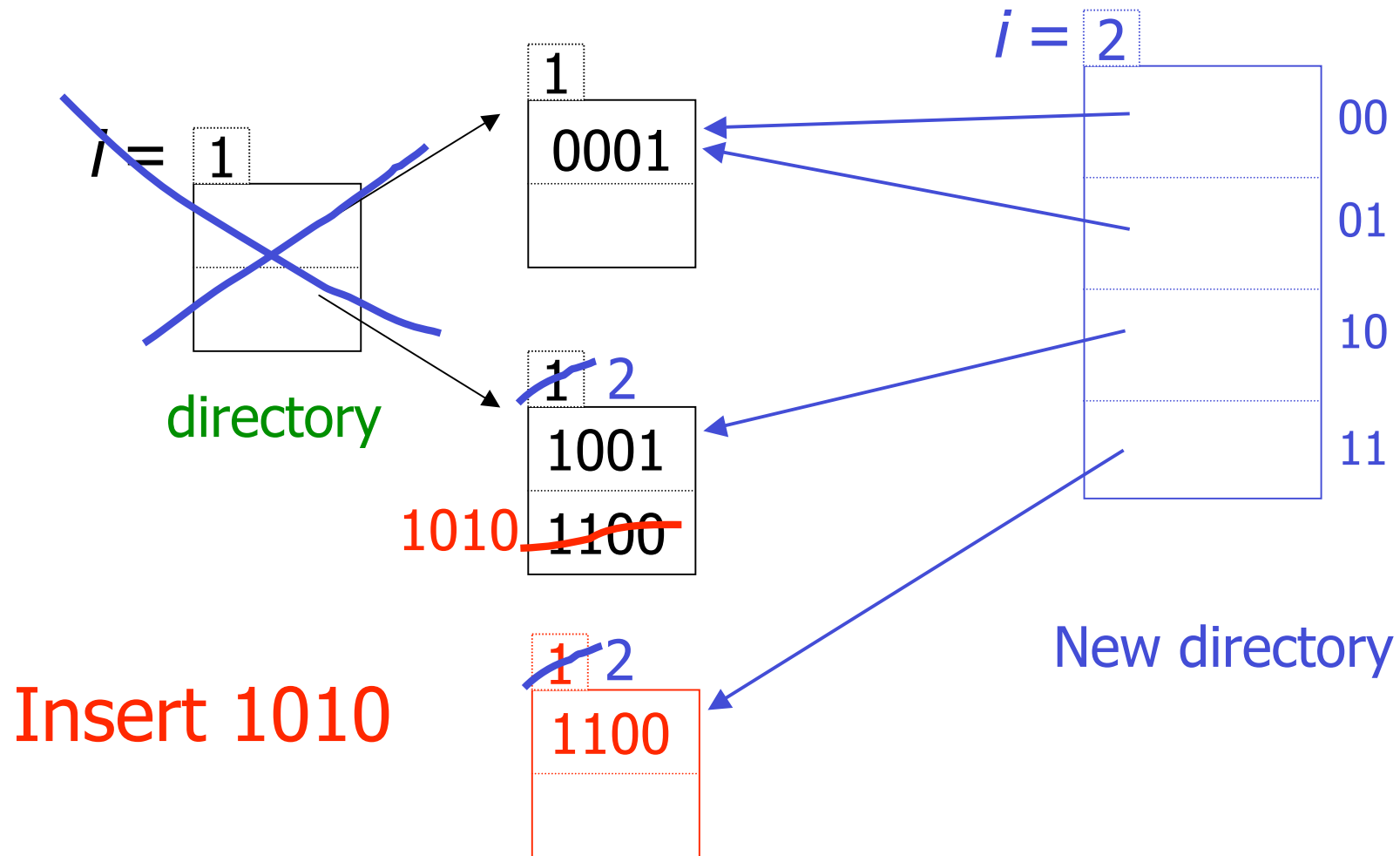
(a) Use  $i$  of  $b$  bits output by hash function



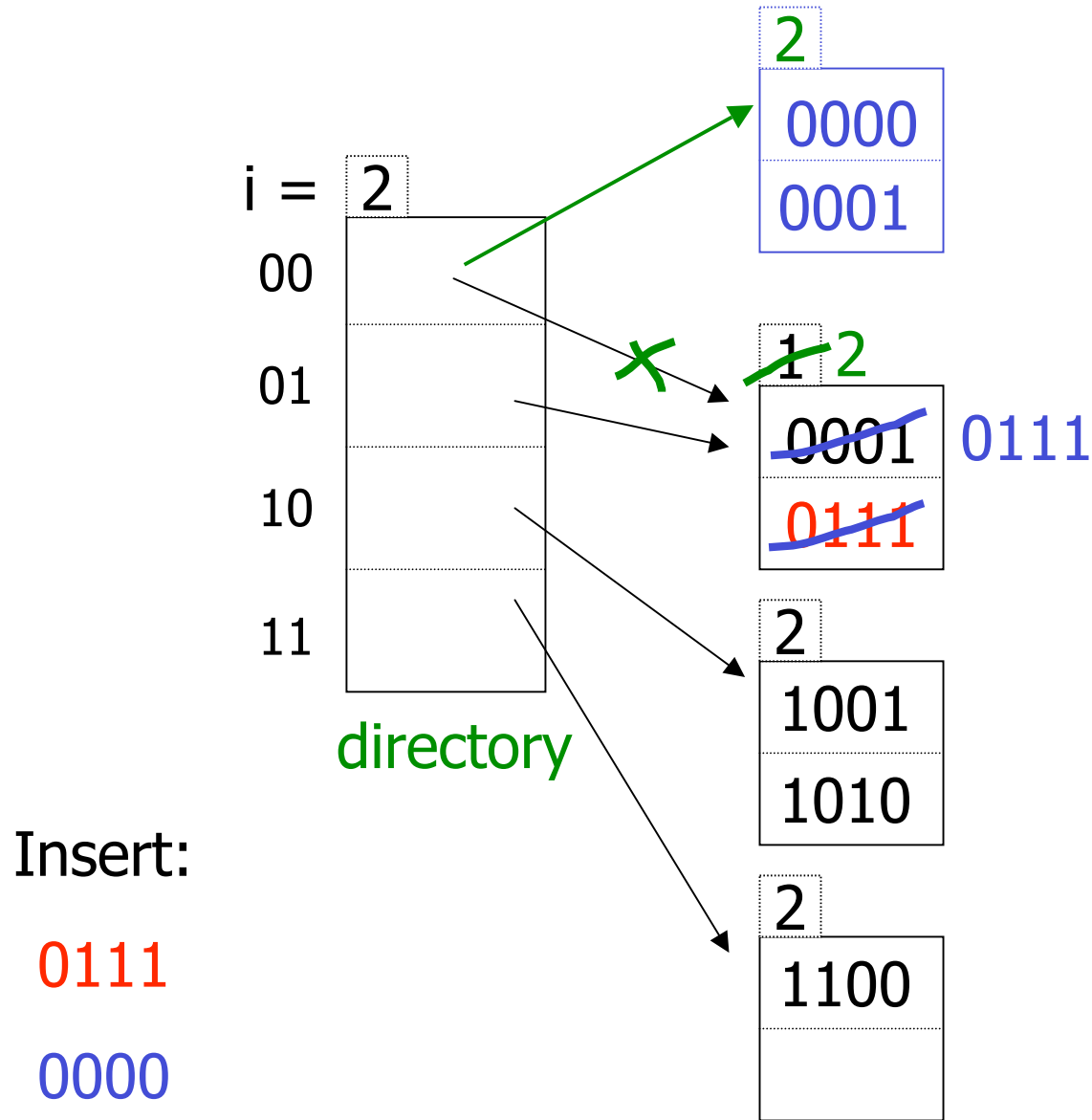
(b) Look up pointer to record in a **directory**.

# Extendible hashing example

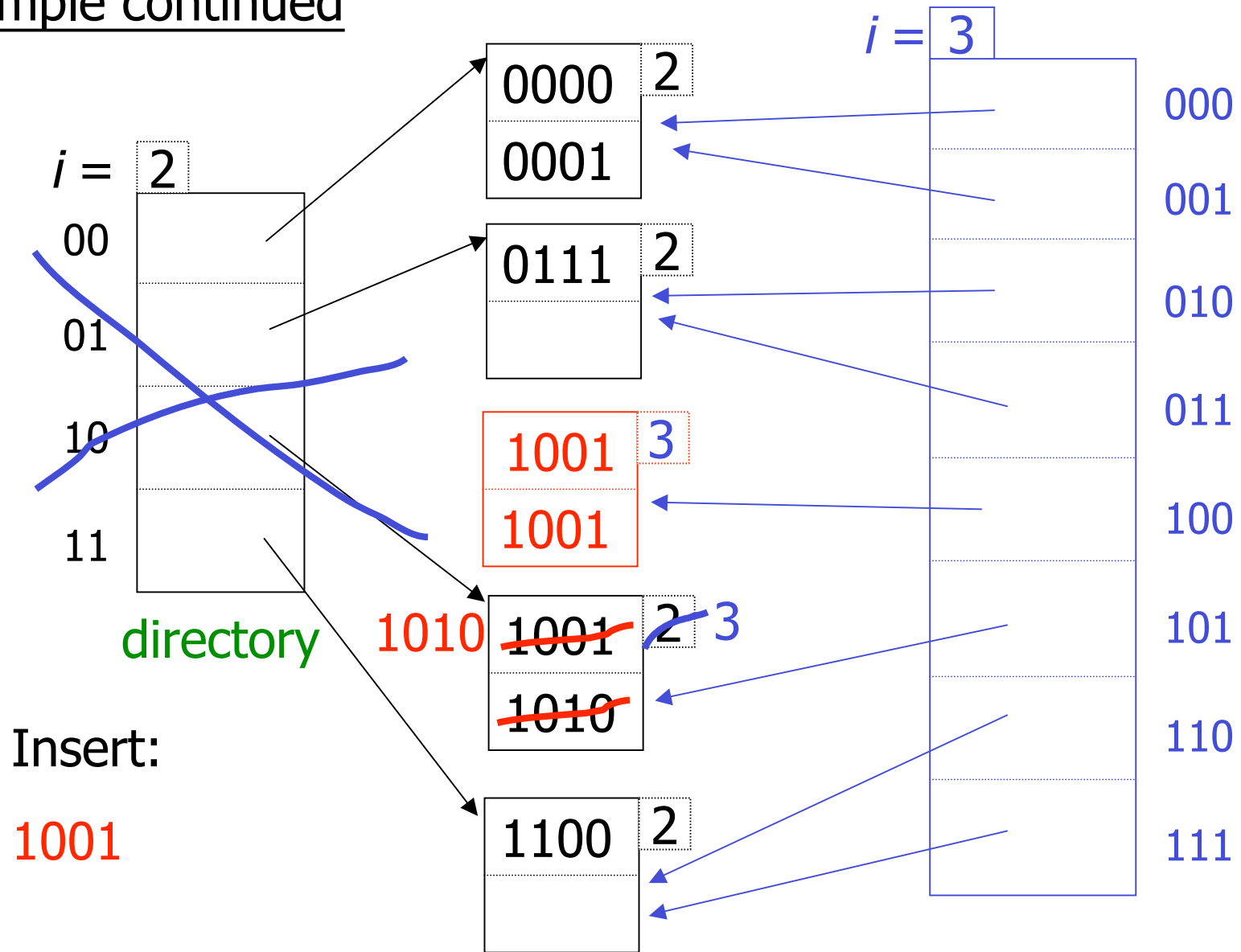
$h(K)$  is 4 bits, 2 keys/bucket. In examples we assume  $h(K)=K$ .



# Example continued



# 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  $\frac{4N^{1+1/B}}{B}$  blocks (proof omitted!)
- Again: Good to have large  $B$ .

# Problem session

Compare extendible hashing to a sparse index:

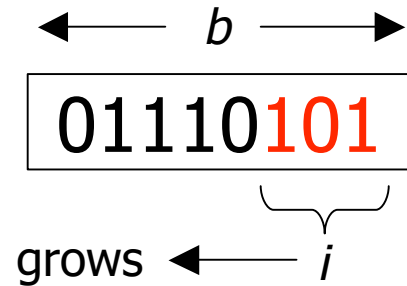
- When is one more efficient than the other?
- Consider various combinations of  $N$ ,  $B$  and  $M$  (internal memory).

# Linear hashing

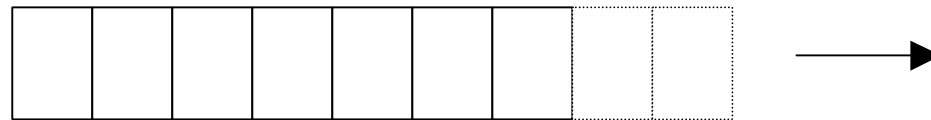
-another dynamic hashing scheme

## Two ideas:

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

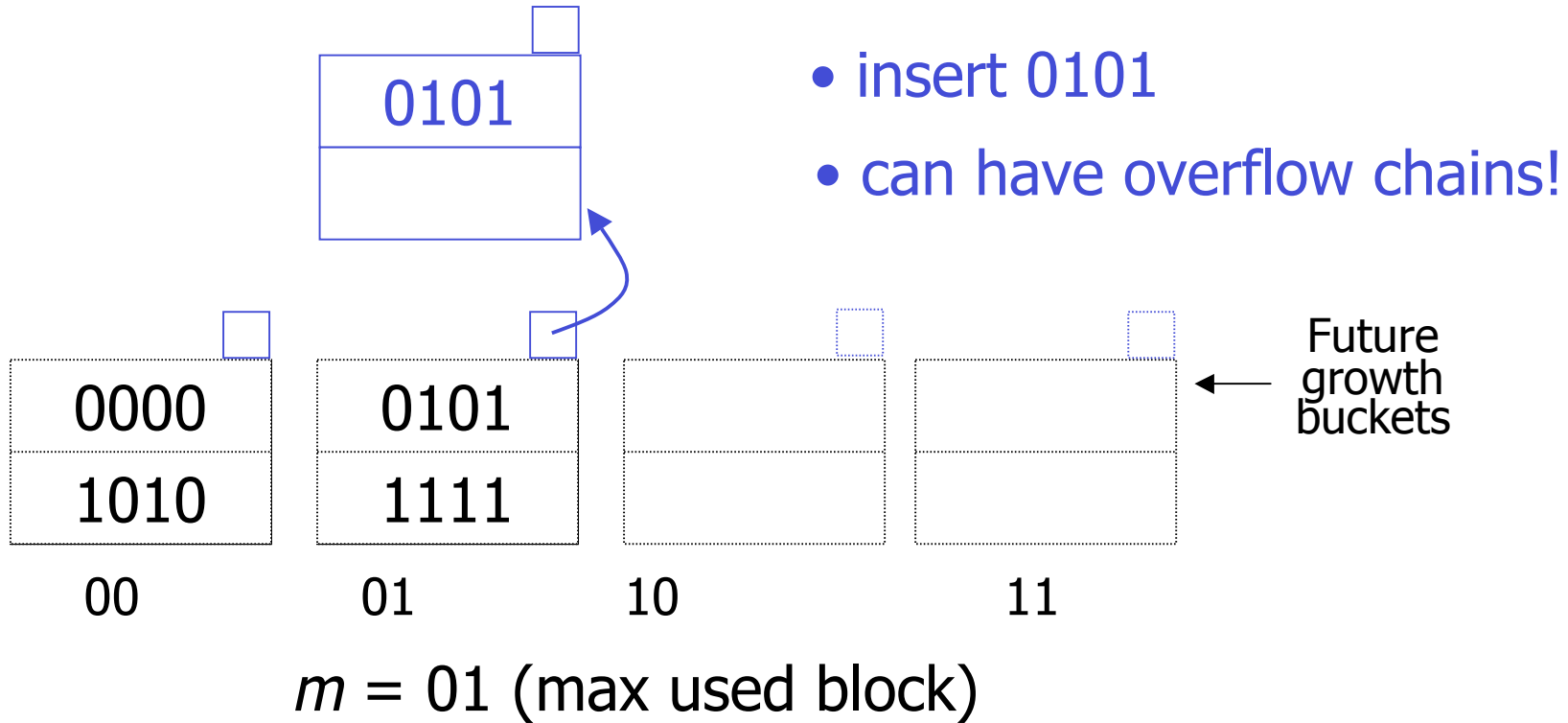


(b) Hash table grows one bucket at a time



# Linear hashing example

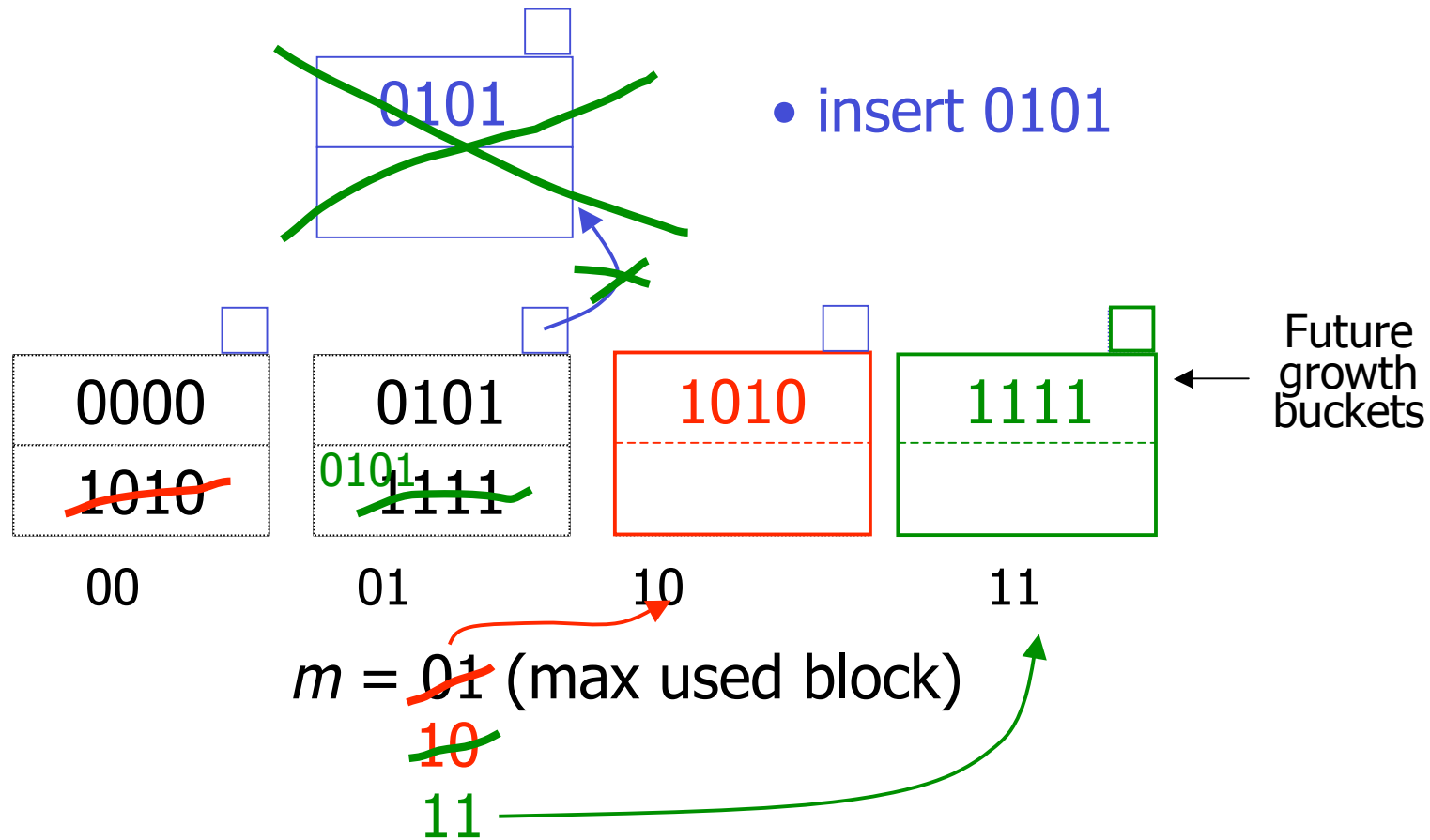
$b=4$  bits,  $i=2$ , 2 keys/bucket



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.

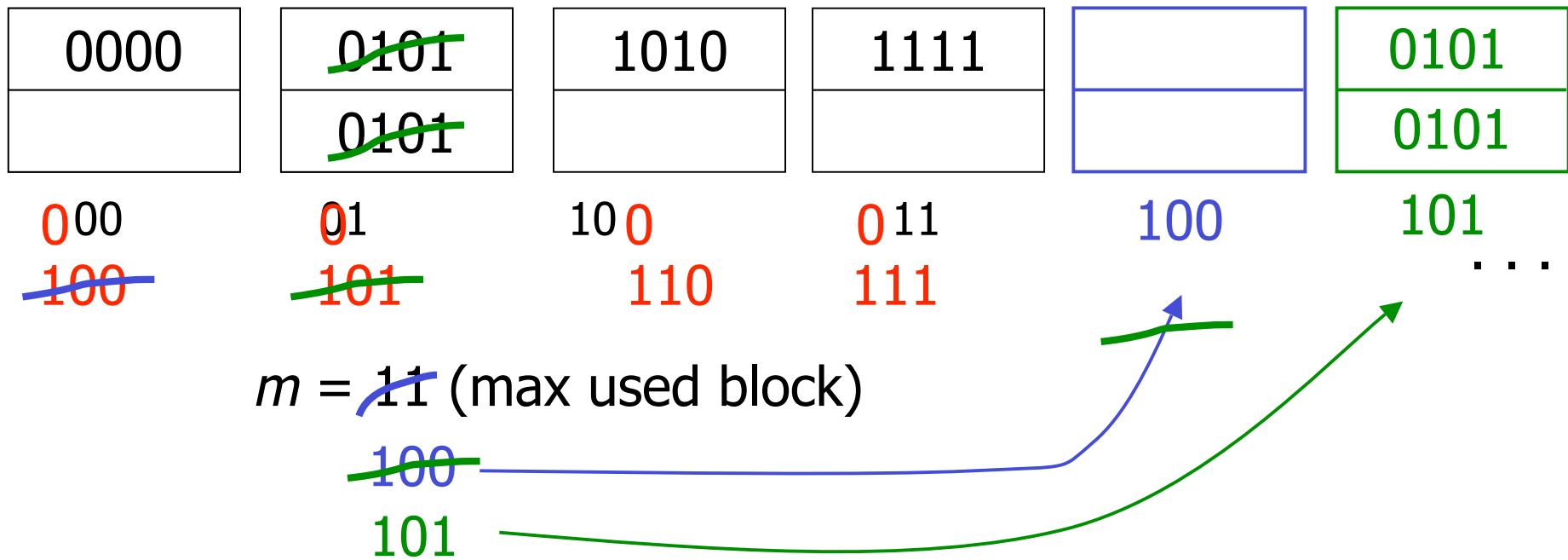
$b=4$  bits,  $i=2$ , 2 keys/bucket



# Linear hashing example, cont.

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

$i =$ ~~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  $\lfloor (m+1) h(K) \rfloor$
- 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, Brodal, Rauhe, 2001; Brodal, Mortensen, Pagh 2004)
- Theoretical result, out of scope for ADBT.

# Summary I

- **Indexing** is a "key" database technology.
- **Conventional indexes** sufficient if updates are few.
- **B-trees** (and variants) are more flexible
  - The choice of most DBMSs.
  - Theoretically "optimal":  $O(\log_B N)$  I/Os per operation.
  - Support range queries.

# Summary II

- **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.