Comp 5311 Database Management Systems

10. B+-trees and Dynamic Hashing

B^+-Tree Index Files

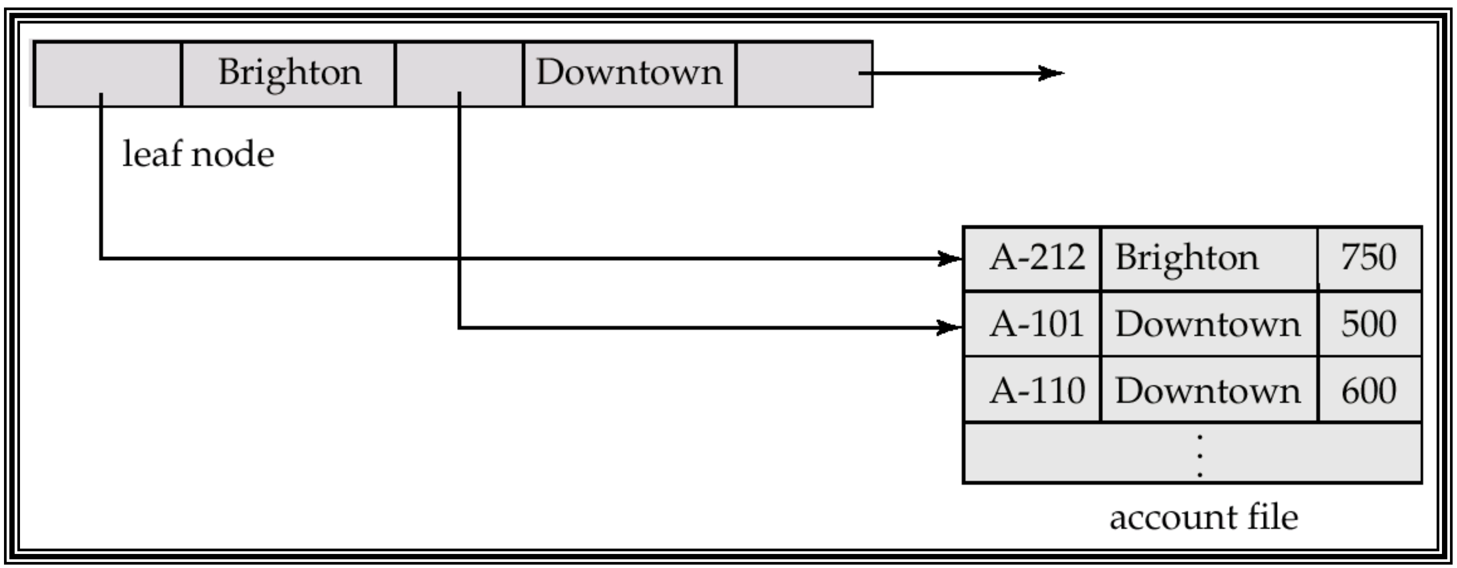
- Disadvantage of indexed-sequential files: performance degrades as file grows, since many overflow blocks get created. Periodic reorganization of entire file is required.
- Advantage of B^+-tree index files: automatically reorganizes itself with small, local, changes, in the face of insertions and deletions. Reorganization of entire file is not required to maintain performance.
- Disadvantage of B^+-trees: extra insertion and deletion overhead, space overhead.
- Advantages of B^+-trees outweigh disadvantages, and they are used extensively in all commercial products.

B^+-Tree Index Files (Cont.)

- All paths from root to leaf are of the same length (i.e., balanced tree)
- Each node has between $\lceil n/2 \rceil$ and n pointers. Each leaf node stores between $\lceil (n-1)/2 \rceil$ and $n-1$ values.
- n is called fanout (it corresponds to the maximum number of pointers/children). The value $\lceil (n-1)/2 \rceil$ is called order (it corresponds to the minimum number of values).
- Special cases:
  - If the root is not a leaf, it has at least 2 children.
  - If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and $(n-1)$ values.

Leaf Nodes in B^+-Trees

- For $i = 1, 2, \ldots, n-1$, pointer P_i either points to a file record with search-key value K_i, or to a bucket of pointers to file records, each record having search-key value K_i. If L_i, L_j are leaf nodes and $i < j$, L_i's search-key values are less than L_j's search-key values
- P_n points to next leaf node in search-key order (right sibling node)

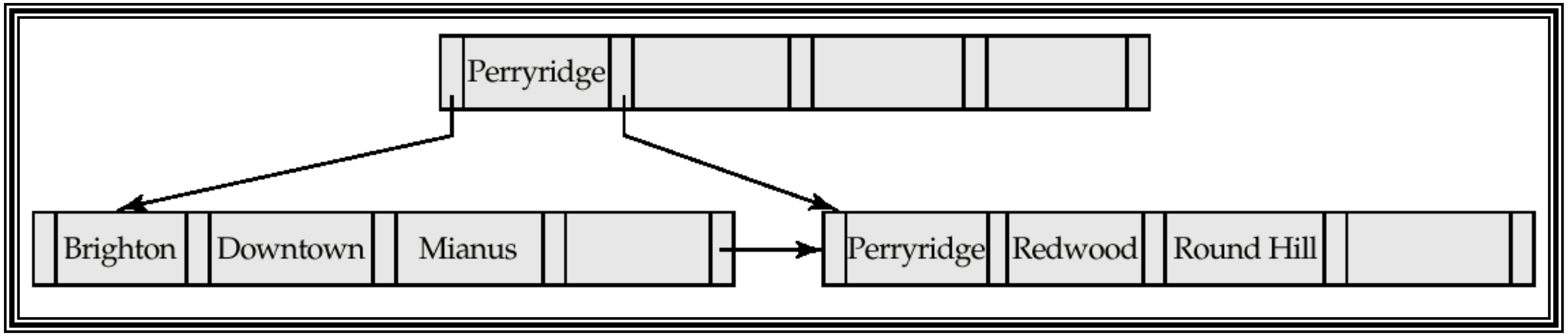


Non-Leaf Nodes in B$^+$-Trees

- Non leaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with *m* pointers:
  - All the search-keys in the subtree to which P_1 points are less than K_1
  - For $2 \leq i \leq n - 1$, all the search-keys in the subtree to which P_i points have values greater than or equal to K_{i-1} and less than K_i *(example : P_2 points to a node where the value v of each key is K_1 <=v< k_2)*

P_1	K_1	P_2	$\ldots$	P_{n-1}	K_{n-1}	P_n

Example of B^+-tree



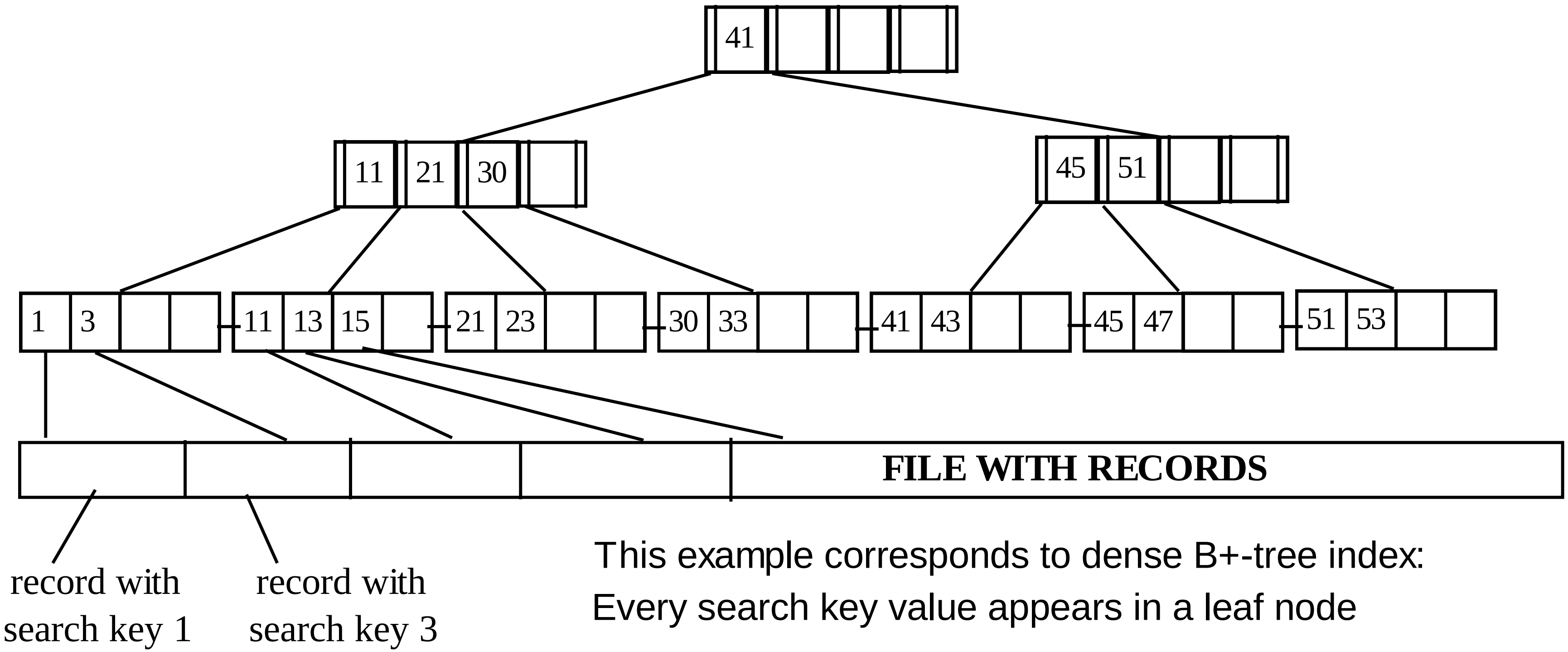
B^+-tree for *account* file (n = 5)

- Leaf nodes must have between 2 and 4 values ($\lceil (n-1)/2 \rceil$ and $n-1$, with $n = 5$).
- Non-leaf nodes other than root must have between 3 and 5 children ($\lceil (n/2) \rceil$ and n with $n = 5$).
- Root must have at least 2 children.

Observations about B^+-trees

- Since the inter-node connections are done by pointers, the close blocks need not be "physically" close (i.e., no need for sequential storage).
- The non-leaf levels of the B^+-tree form a hierarchy of sparse indices.
- The B^+-tree contains a relatively small number of levels (logarithmic in the size of the main file), thus search can be conducted efficiently.
- Insertions and deletions to the main file can be handled efficiently, as the index can be restructured in logarithmic time (as we shall see).

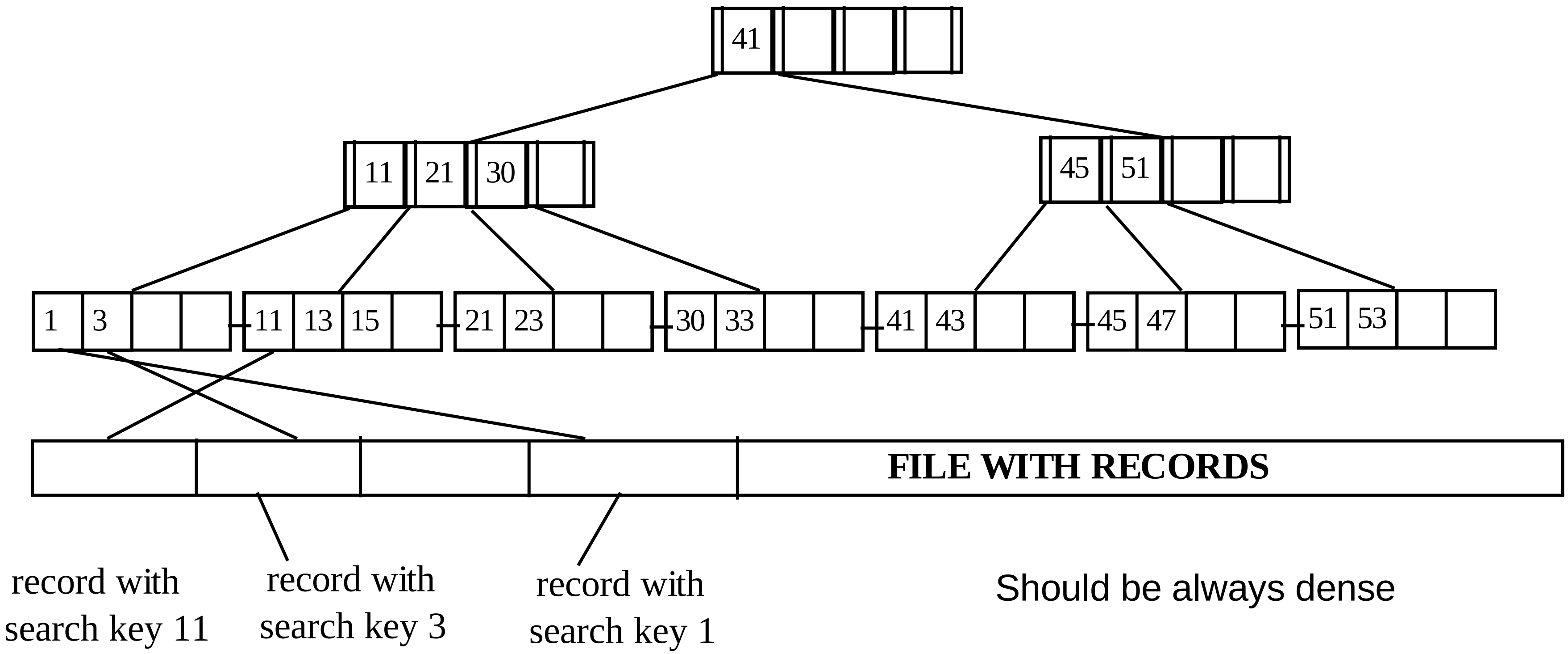
Example of clustering (primary) B+-tree on candidate key



This example corresponds to dense B+-tree index:
Every search key value appears in a leaf node

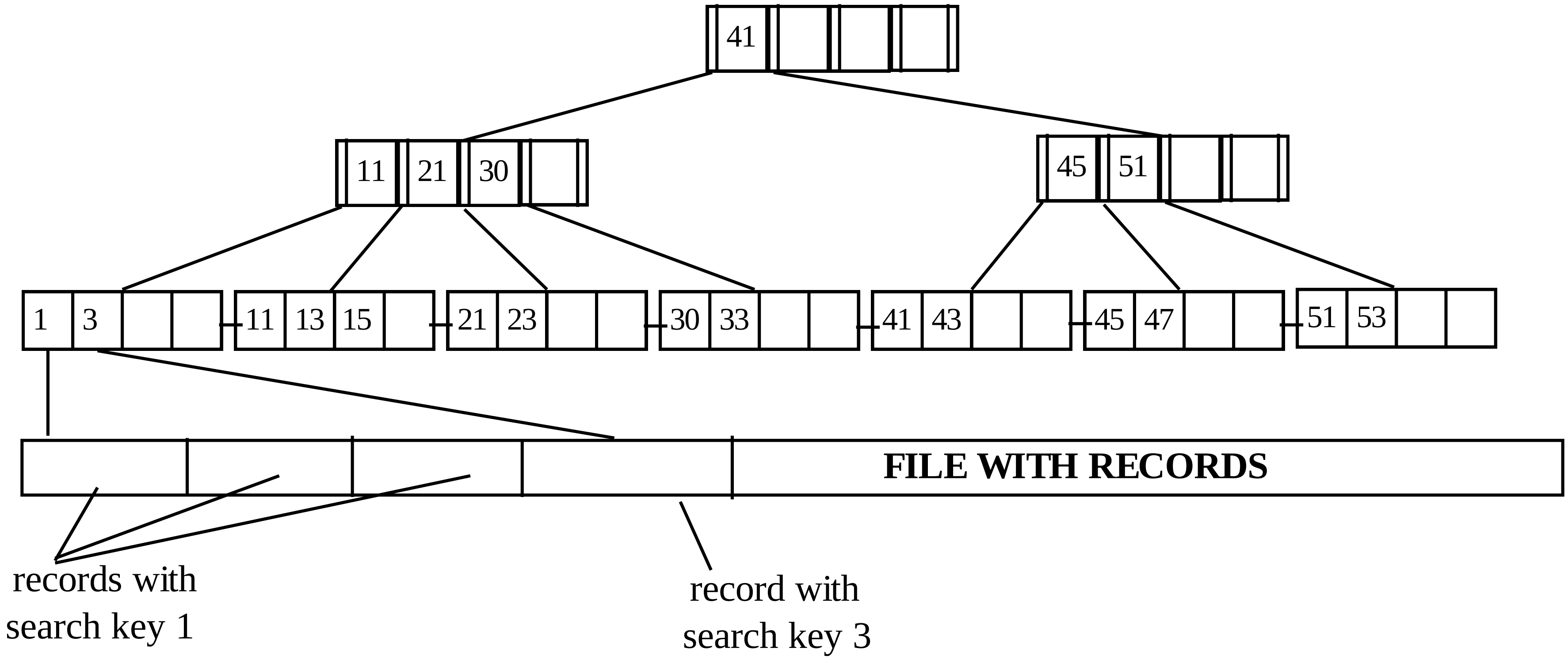
You may also have sparse B+-tree, e.g., entries in leaf nodes correspond to pages

Example of non-clustering (secondary) B+-tree on candidate key

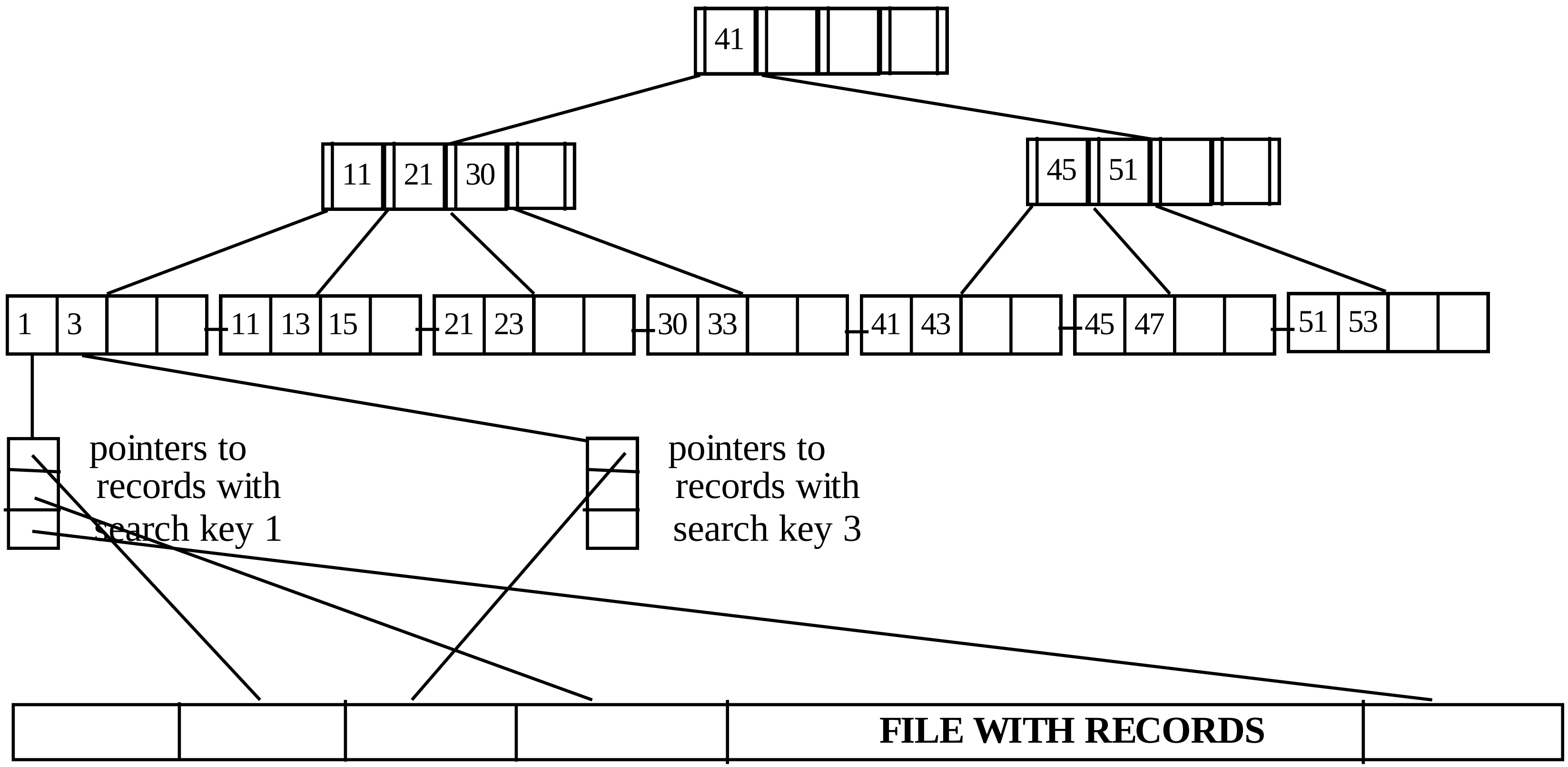


Should be always dense

Example of clustering B+-tree on non-candidate key



Example of non-clustering B+-tree on non-candidate key



Queries on B^+-Trees

- Find all records with a search-key value of *k.*
  - Start with the root node
    - If there is an entry with search-key value $K_j = k$, *follow pointer* P_{j+1}
    - Otherwise, if $k < K_{m-1}$ (there are *m* pointers in the node, i.e., *k* is not the larger than all values in the node) *follow pointer* P_j, where K_j is the smallest search-key value > *k*.
    - Otherwise, if $k \geq K_{m-1}$, follow P_m to the child node.
  - If the node reached by following the pointer above is not a leaf node, repeat the above procedure on the node, and follow the corresponding pointer.
  - Eventually reach a leaf node. If for some *i*, key $K_i = k$ follow pointer P_i to the desired record or bucket. Else no record with search-key value *k* exists.

Queries on B^{+-}Trees (Cont.)

- In processing a query, a path is traversed in the tree from the root to some leaf node.
- If there are *K* search-key values in the file, the path is no longer than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$.
- A node is generally the same size as a disk page, typically 4 kilobytes, and *n* is typically around 100 (40 bytes per index entry).
- With 1 million search key values and *n* = 100, at most
  $log_{50}(1{,}000{,}000) = 4$ nodes are accessed in a lookup.

Inserting a Data Entry into a B+ Tree

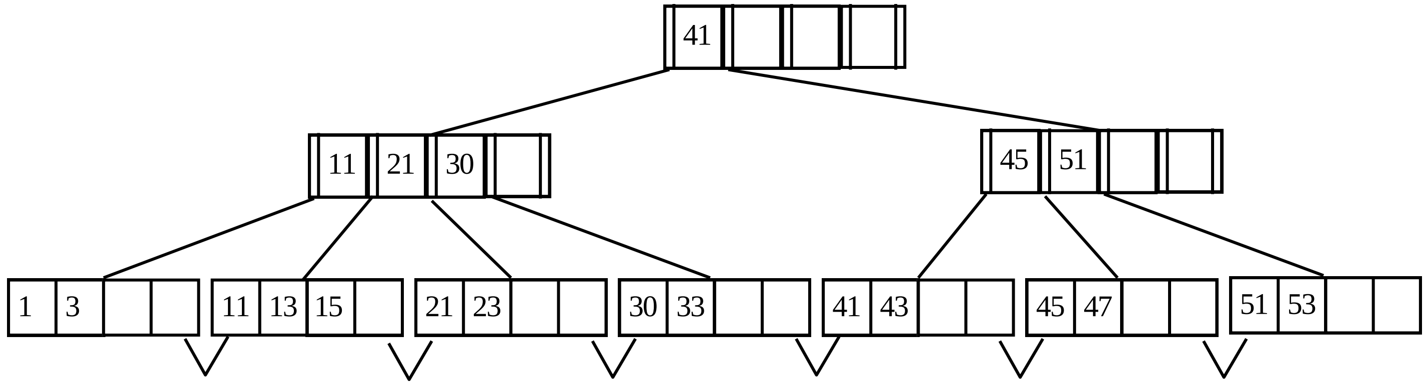
- Find correct leaf *L.*
- Put data entry onto *L*.
  - If *L* has enough space, *done*!
  - Else, must *split* *L (into L and a new node L2)*
    - Redistribute entries evenly, **copy up** middle key.
    - Insert index entry pointing to *L2* into parent of *L*.
- This can happen recursively
  - To split index node, redistribute entries evenly, but **push up** middle key. (Contrast with leaf splits.)
- Splits "grow" tree; root split increases height.
  - Tree growth: gets *wider* or *one level taller at top.*

Deleting a Data Entry from a B+ Tree

- Start at root, find leaf *L* where entry belongs.
- Remove the entry.
  - If L is at least half-full, *done!*
  - If L less than half-full,
    - Try to re-distribute, borrowing from *sibling* *(adjacent node to the right)*.
    - If re-distribution fails, *merge* *L* and sibling.
- If merge occurred, must delete entry (pointing to *L* or sibling) from parent of *L*.
- Merge could propagate to root, decreasing height.

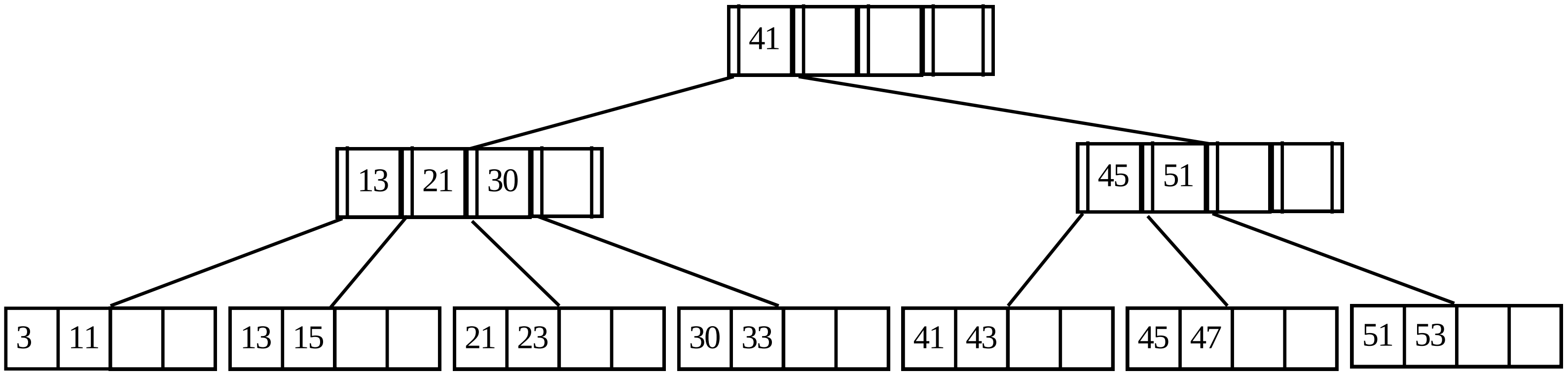
B+-tree Updates

Consider the B+-tree below with order 2 (each node except for the root must contain at least two search key values – and 3 pointers).



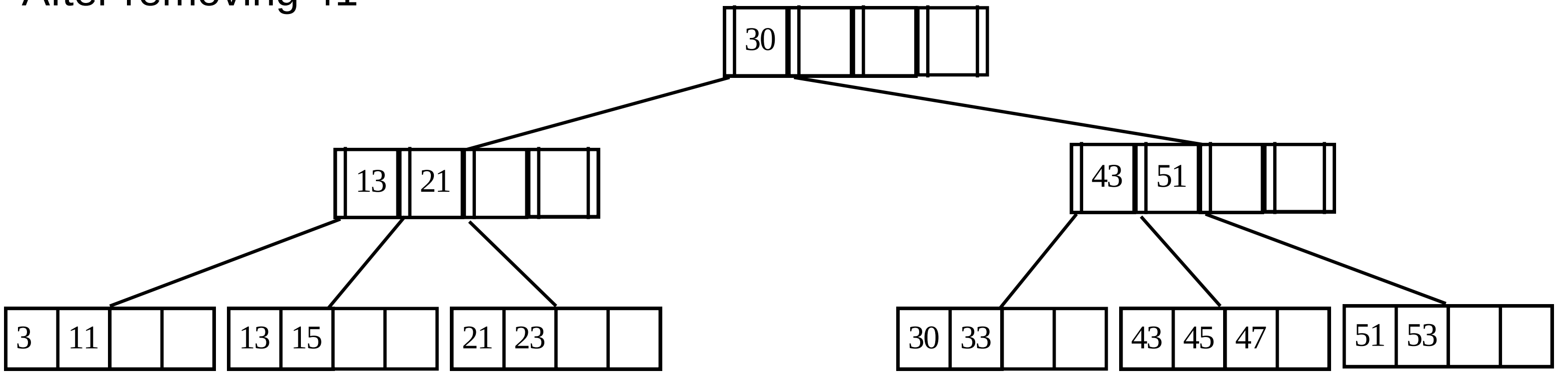
Remove 1

Remove 41



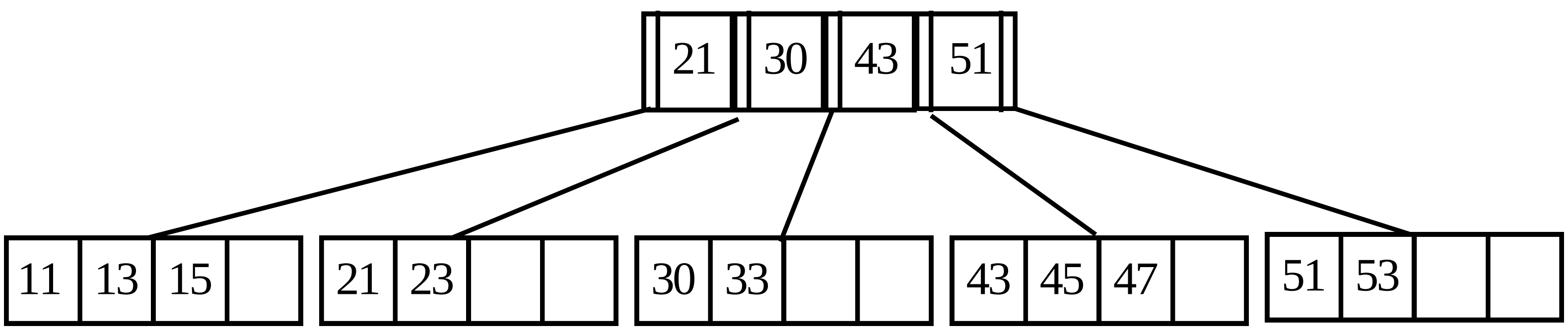
B+-tree Updates (cont)

After removing 41



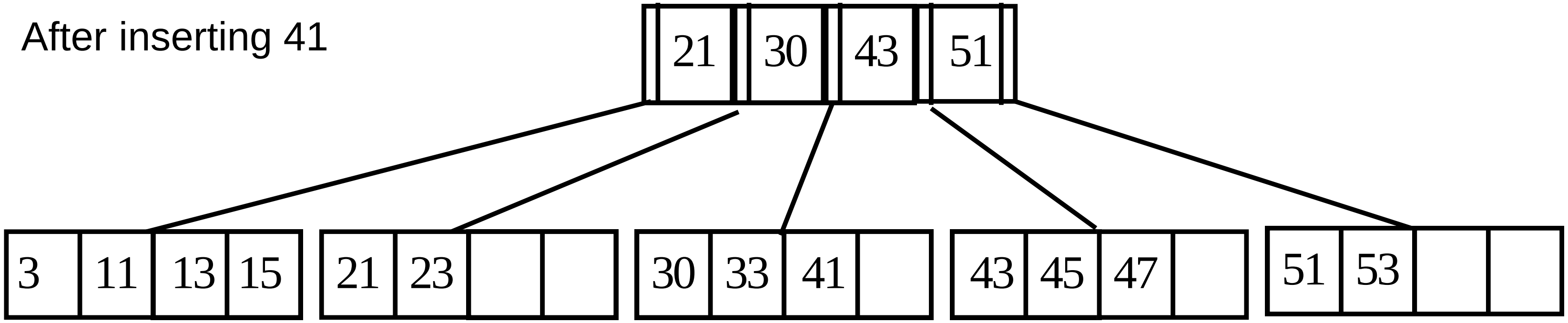
Remove 3

Insert 41

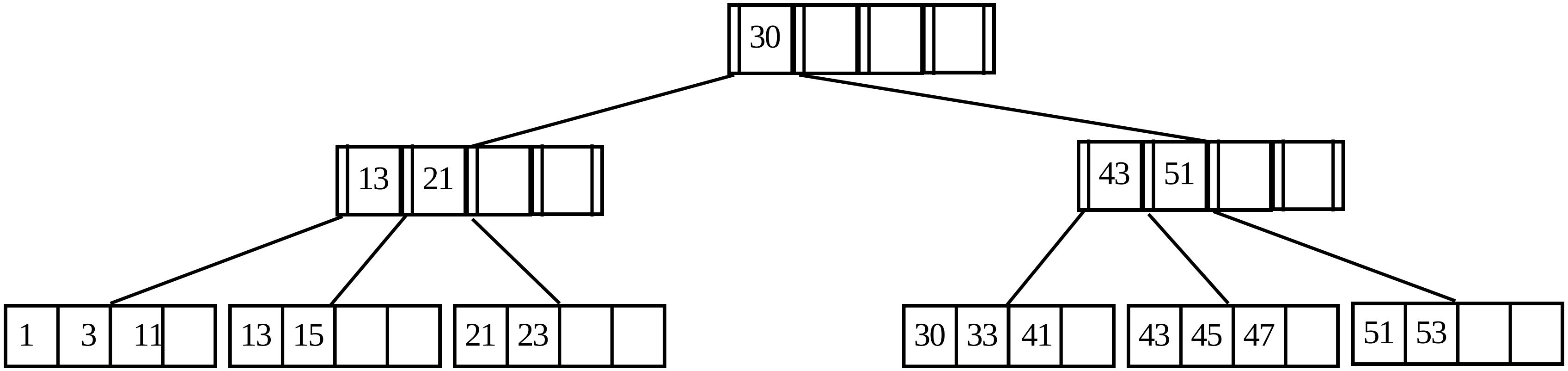


B+-tree Updates (cont)

After inserting 41



Insert 1

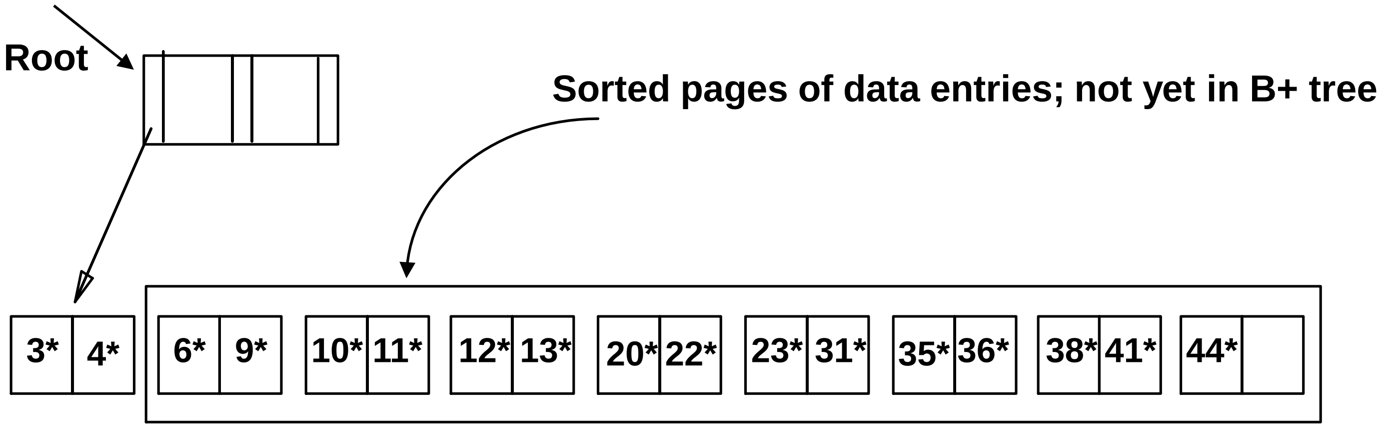


B^+-Tree File Organization

- Index file degradation problem is solved by using B^+-Tree indices. Data file degradation problem is solved by using B^+-Tree File Organization.
- The leaf nodes in a B^+-tree file organization store records, instead of pointers.
- Since records are larger than pointers, the maximum number of records that can be stored in a leaf node is less than the number of pointers in a nonleaf node.
- Leaf nodes are still required to be half full.
- Insertion and deletion are handled in the same way as insertion and deletion of entries in a B^+-tree index.

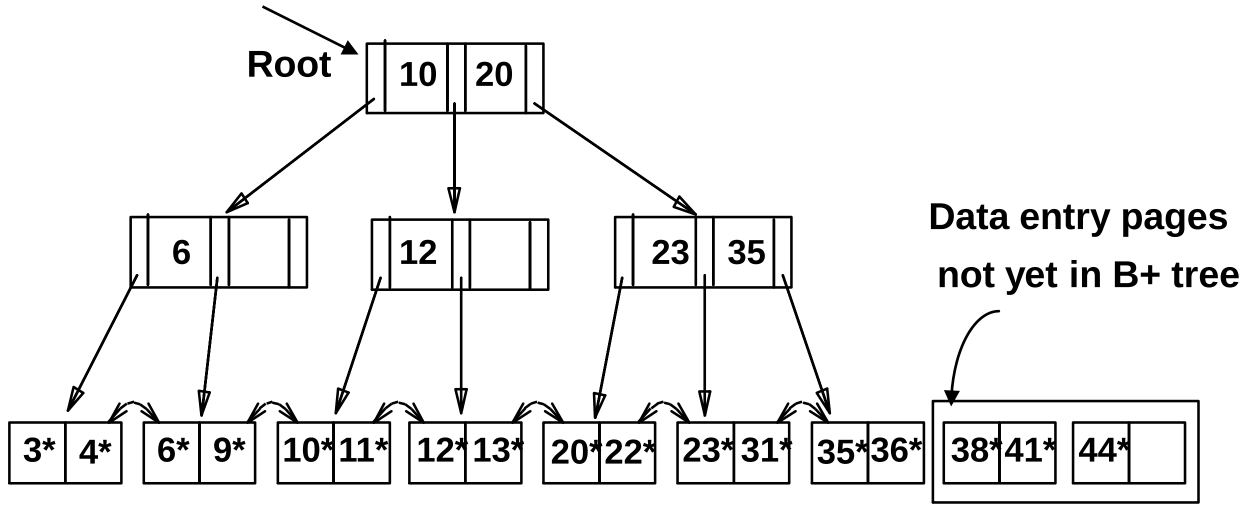
Bulk Loading of a B+ Tree

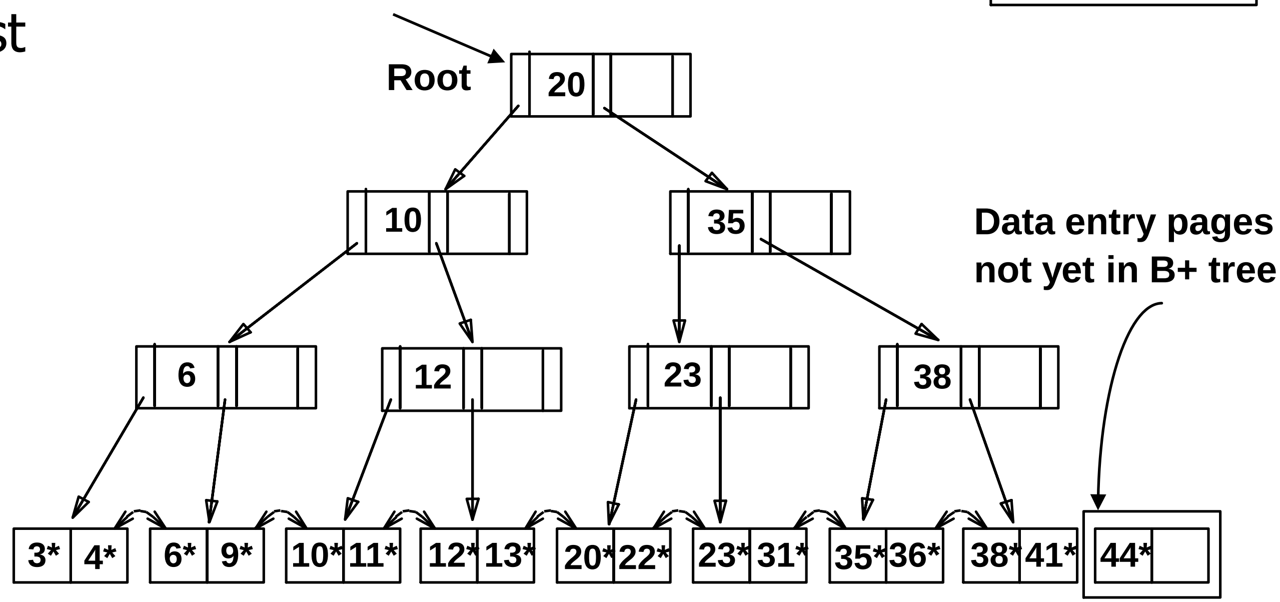
- If we have a large collection of records, and we want to create a B+ tree on some field, doing so by repeatedly inserting records is very slow.
- *Bulk Loading* can be done much more efficiently.
- *Initialization*: Sort all data entries (using external sorting – will be discussed in the next class), insert pointer to first (leaf) page in a new (root) page.



Bulk Loading (Cont.)

- Index entries for leaf pages always entered into right-most index page just above leaf level. When this fills up, it splits. (Split may go up right-most path to the root.)
- Much faster than repeated inserts!

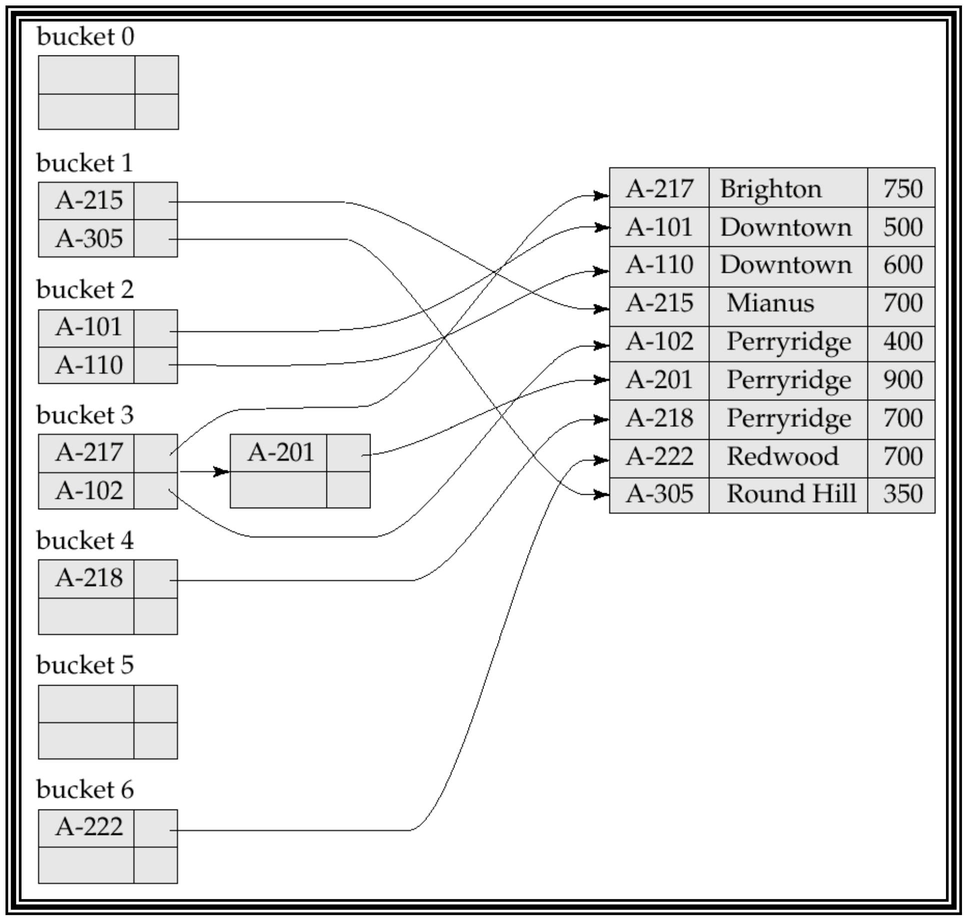




Hash Indices

- Hashing can be used not only for file organization, but also for index-structure creation.
- A **hash index** organizes the search keys, with their associated record pointers, into a hash file structure.
- Strictly speaking, hash indices are always secondary indices
  - if the file itself is organized using hashing, a separate primary hash index on it using the same search-key is unnecessary.
- The version that we discuss is for relatively static datasets
  - We want to build a hash index for an existing dataset - we expect the number of records not to change too much.

Example of Hash Index



Hash Functions

- In the worst case, the hash function maps all search-key values to the same bucket; this makes access time proportional to the number of search-key values in the file.
- Ideal hash function is **random**, so each bucket will have the same number of records assigned to it irrespective of the *actual distribution* of search-key values in the file.
- Typical hash functions perform computation on the internal binary representation of the search-key.
  - For example, for a string search-key, the binary representations of all the characters in the string could be added and the sum modulo the number of buckets could be returned.

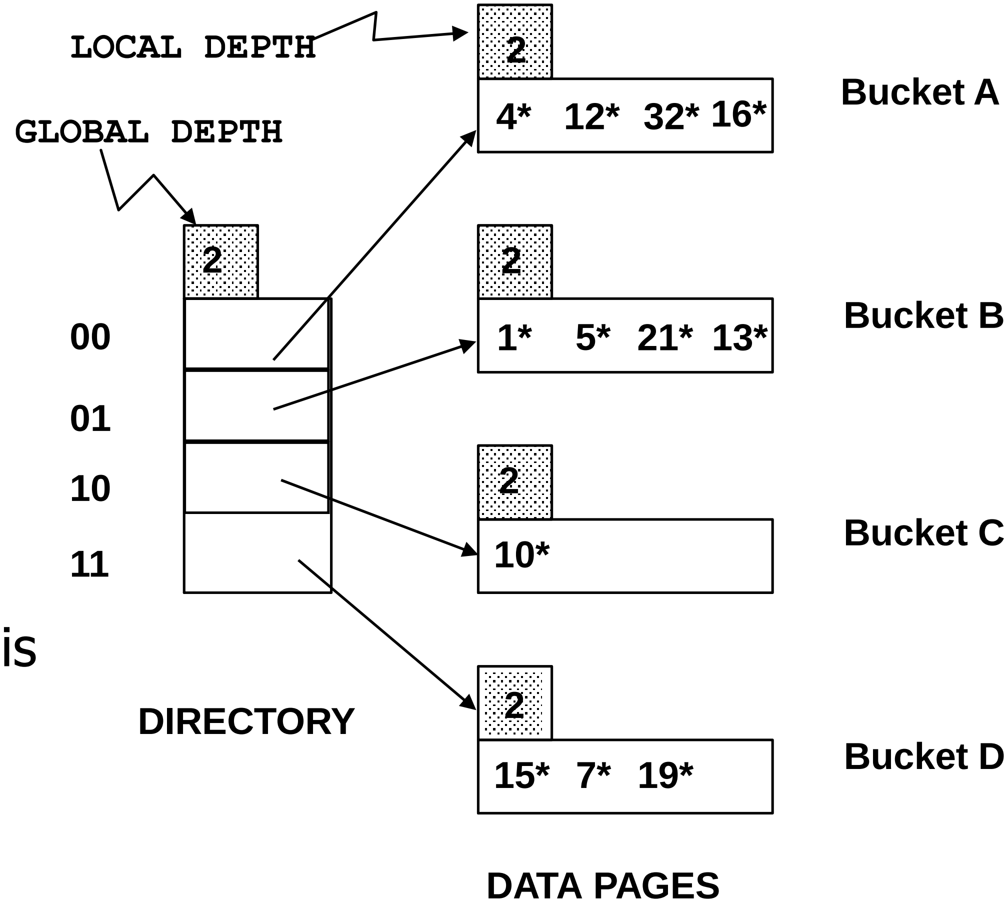
Deficiencies of Static Hashing

- In static hashing, function *h* maps search-key values to a fixed set of *B* of bucket addresses.
  - Databases grow with time. If initial number of buckets is too small, performance will degrade due to too much overflows.
  - If file size at some point in the future is anticipated and number of buckets allocated accordingly, significant amount of space will be wasted initially.
  - If database shrinks, again space will be wasted.
  - One option is periodic re-organization of the file with a new hash function, but it is very expensive.
- These problems can be avoided by using techniques that allow the number of buckets to be modified dynamically.

Extendible Hashing

- Situation: Bucket (primary page) becomes full. Why not re-organize file by *doubling* # of buckets?
  - Reading and writing all pages is expensive!
  - *Idea*: Use *directory of pointers to buckets*, double # of buckets by *doubling the directory,* splitting just the bucket that overflowed!
  - Directory much smaller than file, so doubling it is much cheaper. Only one page of data entries is split. *No overflow page*!
  - Trick lies in how hash function is adjusted!

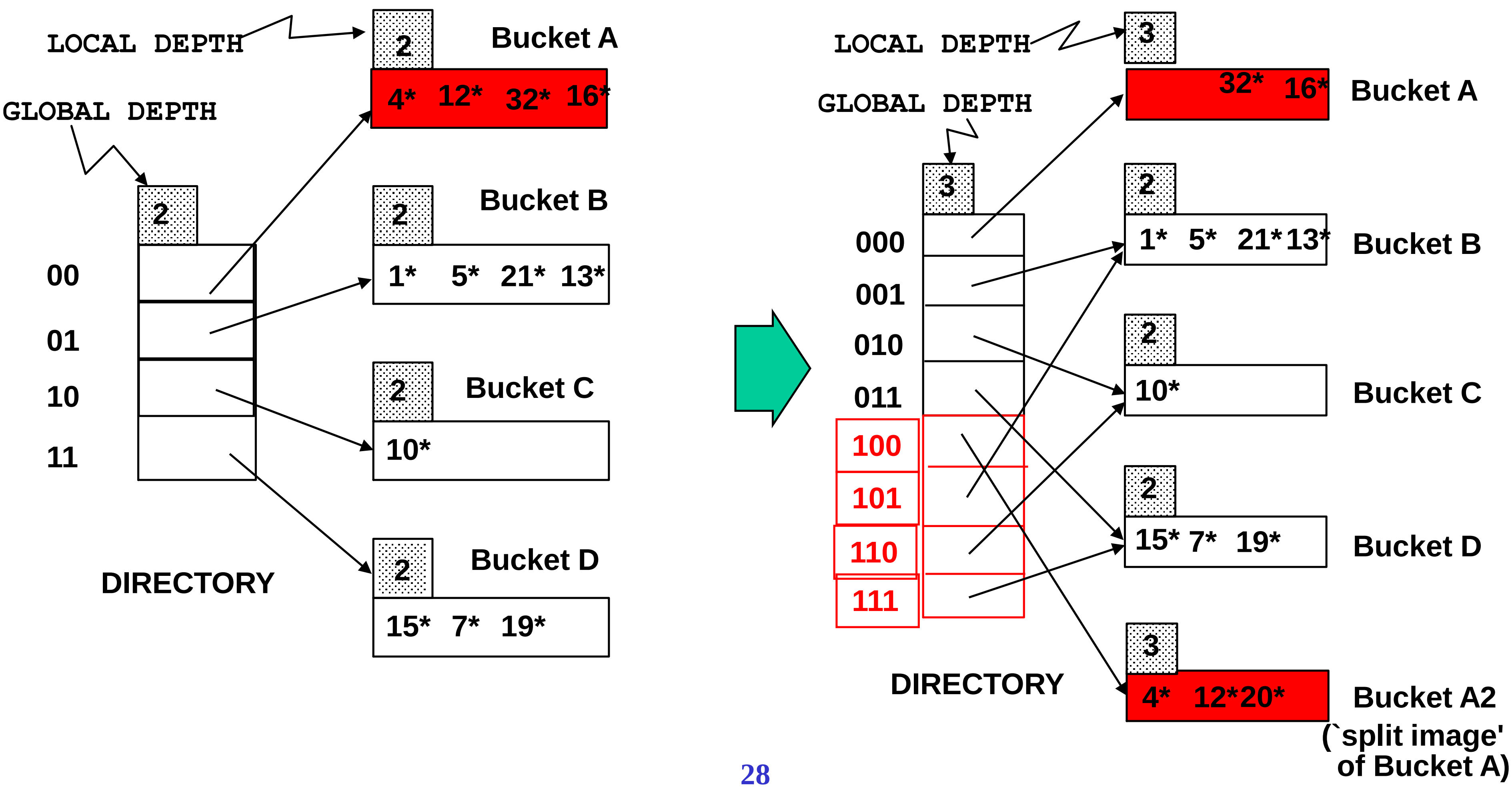
Example



- Directory is array of size 4.
- To find bucket for *r*, take last *global depth* bits of **h**(*r*); we denote *r* by **h**(*r*).
  - If **h**(*r*) = 5 = binary 101, it is in bucket pointed to by 01.

- **Insert**: If bucket is full, *split* it (*allocate new page, re-distribute*).
- *If necessary*, double the directory. (As we will see, splitting a bucket does not always require doubling; we can tell by comparing *global depth* with *local depth* for the split bucket.)

Insert **h**(r)=20 (10100)

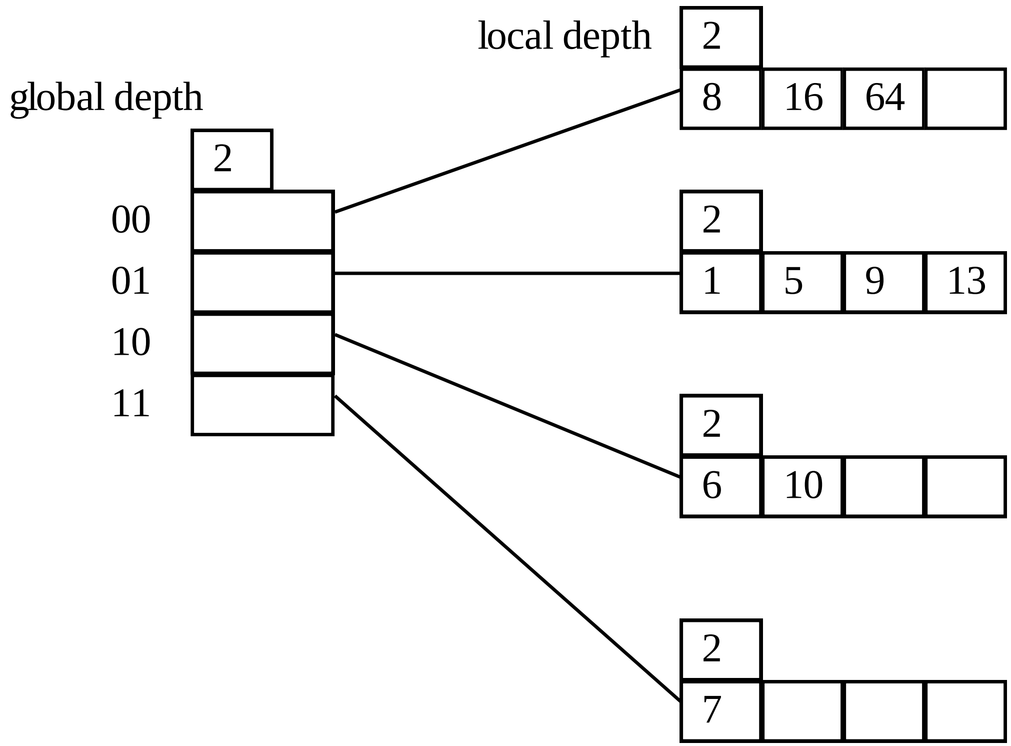


Points to Note

- 20 = binary 10100. Last **2** bits (00) tell us *r* belongs in A or A2. Last **3** bits needed to tell which.
  - *Global depth of directory*: Max # of bits needed to tell which bucket an entry belongs to.
  - *Local depth of a bucket*: # of bits used to determine if an entry belongs to this bucket.
- When does bucket split cause directory doubling?
  - Before insert, *local depth* of bucket = *global depth*. Insert causes *local depth* to become > *global depth*; directory is doubled by *copying it over* and `fixing' pointer to split image page. (Use of least significant bits enables efficient doubling via copying of directory!)

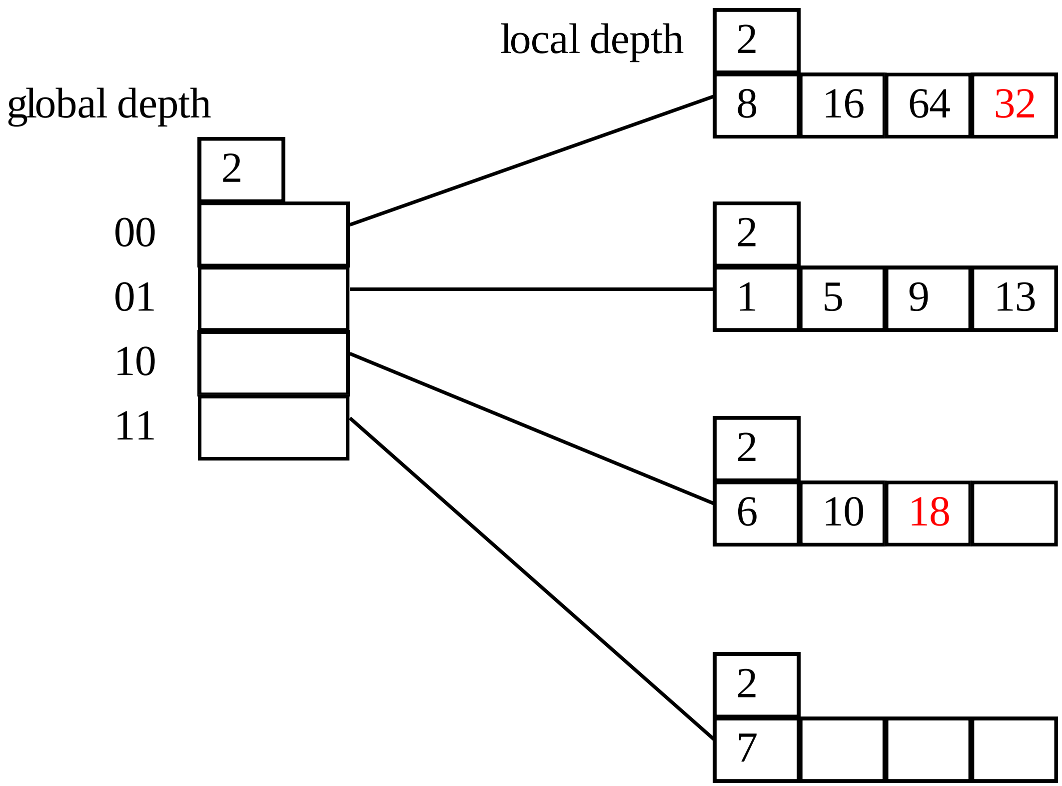
Insert 18 (010010), 32 (100000)

- Assume the following hash index where the hash function is determined by the least significant bits.



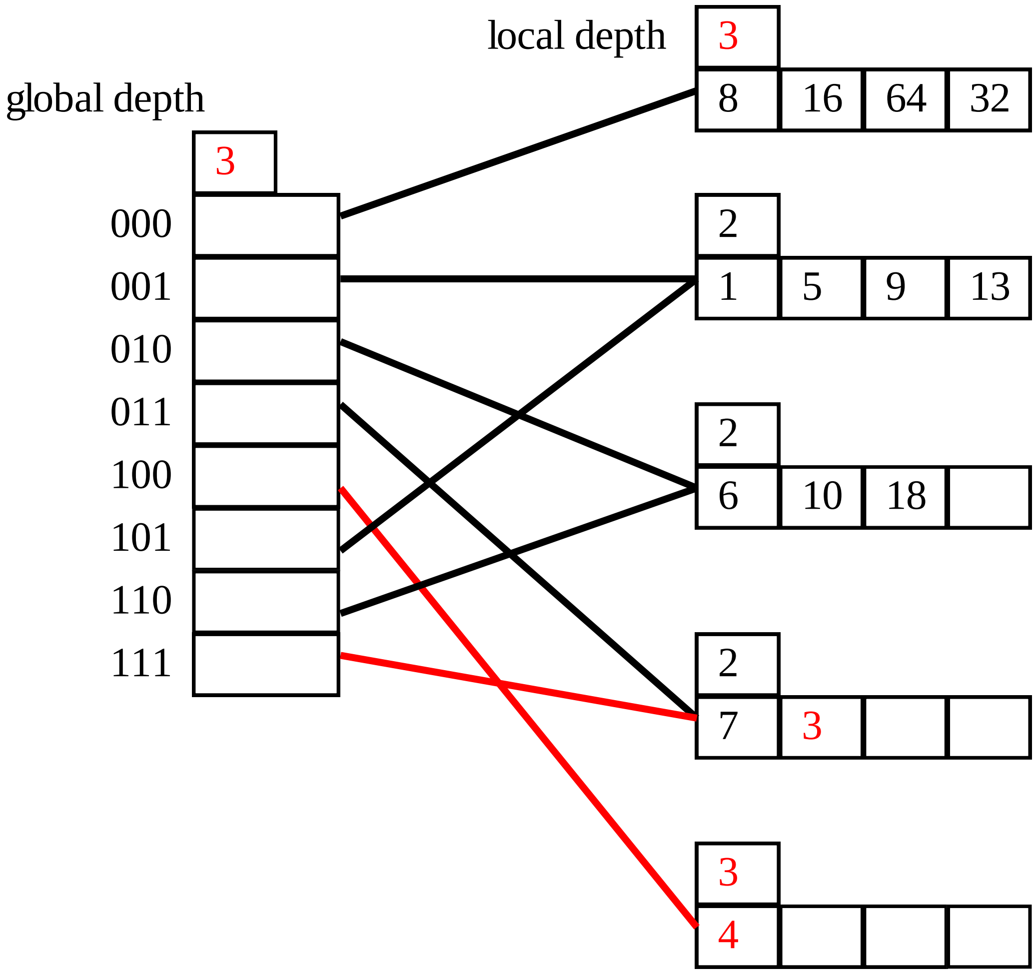
After the insertion of search keys: 18 (010010), 32 (100000).

- Insert: 3 (011), 4 (100)



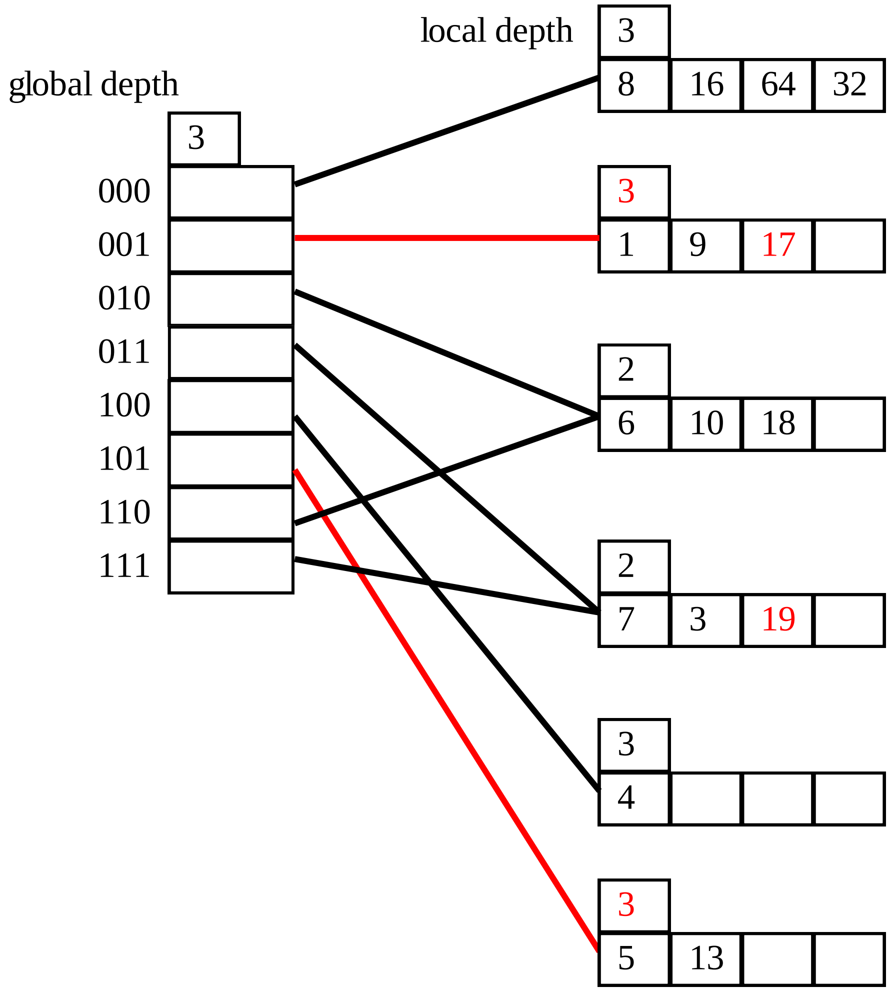
After the insertion of search keys: 4 (100), 3 (011).

- Insert: 19 (10011), 17 (10001)

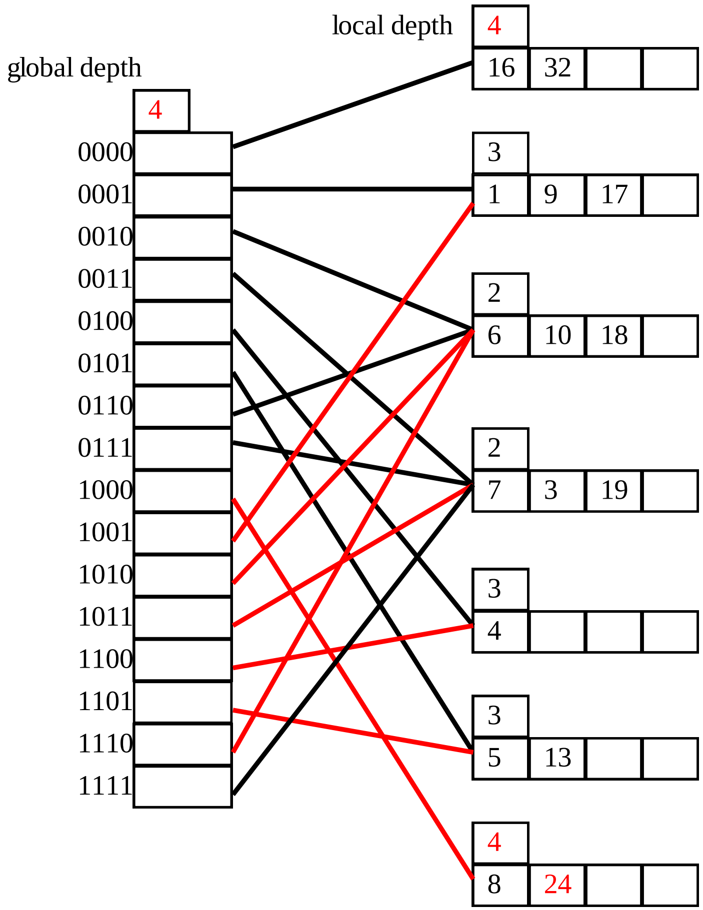


After the insertion of: 19 (10011), 17 (10001)

- Insert 24 (11000)



After the insertion of search key: 24 (11000)



Comments on Extendible Hashing

- If directory fits in memory, equality search answered with one disk access; else two.
  - 100MB file, 100 bytes/rec, 4K pages contains 1,000,000 records (as data entries) and 25,000 directory elements; chances are high that directory will fit in memory.
  - Directory grows in spurts, and, if the distribution *of hash values* is skewed, directory can grow large.
  - Multiple entries with same hash value cause problems!
- **Delete**: If removal of data entry makes bucket empty, can be merged with `split image'. If each directory element points to same bucket as its split image, can halve directory.

Linear Hashing

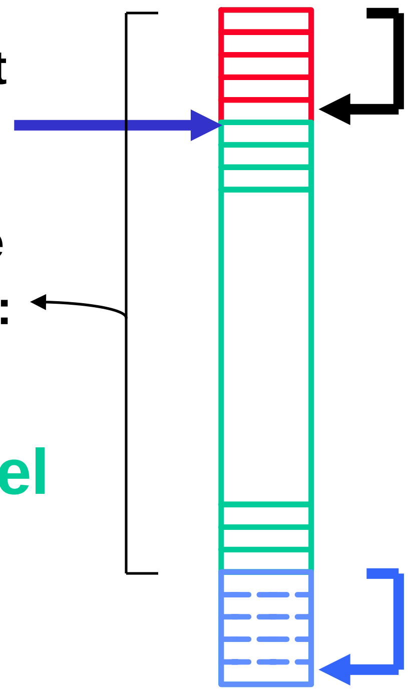
- This is another dynamic hashing scheme, an alternative to Extendible Hashing.
- LH handles the problem of long overflow chains without using a directory, and handles duplicates.
- *Idea*: Use a family of hash functions $\mathbf{h}_0$, $\mathbf{h}_1$, $\mathbf{h}_2$, ...
  - $\mathbf{h}_i(key) = \mathbf{h}(key) \bmod (2^i N)$; N = initial # buckets
  - **h** is some hash function (range is *not* 0 to N-1)
  - If $N = 2^{d0}$, for some *d0*, $\mathbf{h}_i$ consists of applying **h** and looking at the last *di* bits, where $di = d0 + i$.
  - $\mathbf{h}_{i+1}$ doubles the range of $\mathbf{h}_i$ (similar to directory doubling)

Linear Hashing (Contd.)

- Directory avoided in LH by using overflow pages, and choosing bucket to split round-robin.
  - Splitting proceeds in `rounds'. Round ends when all N_R initial (for round *R*) buckets are split. Buckets 0 to *Next-1* have been split; *Next* to N_R yet to be split.
  - Current round number is *Level*.
  - **Search:** To find bucket for data entry *r*, find $\mathbf{h}_{Level}(r)$:
    - If $\mathbf{h}_{Level}(r)$ in range `*Next* to N_R', *r* belongs here.
    - Else, r could belong to bucket $\mathbf{h}_{Level}(r)$ or bucket $\mathbf{h}_{Level}(r) + N_R$; must apply $\mathbf{h}_{Level+1}(r)$ to find out.

Overview of LH File

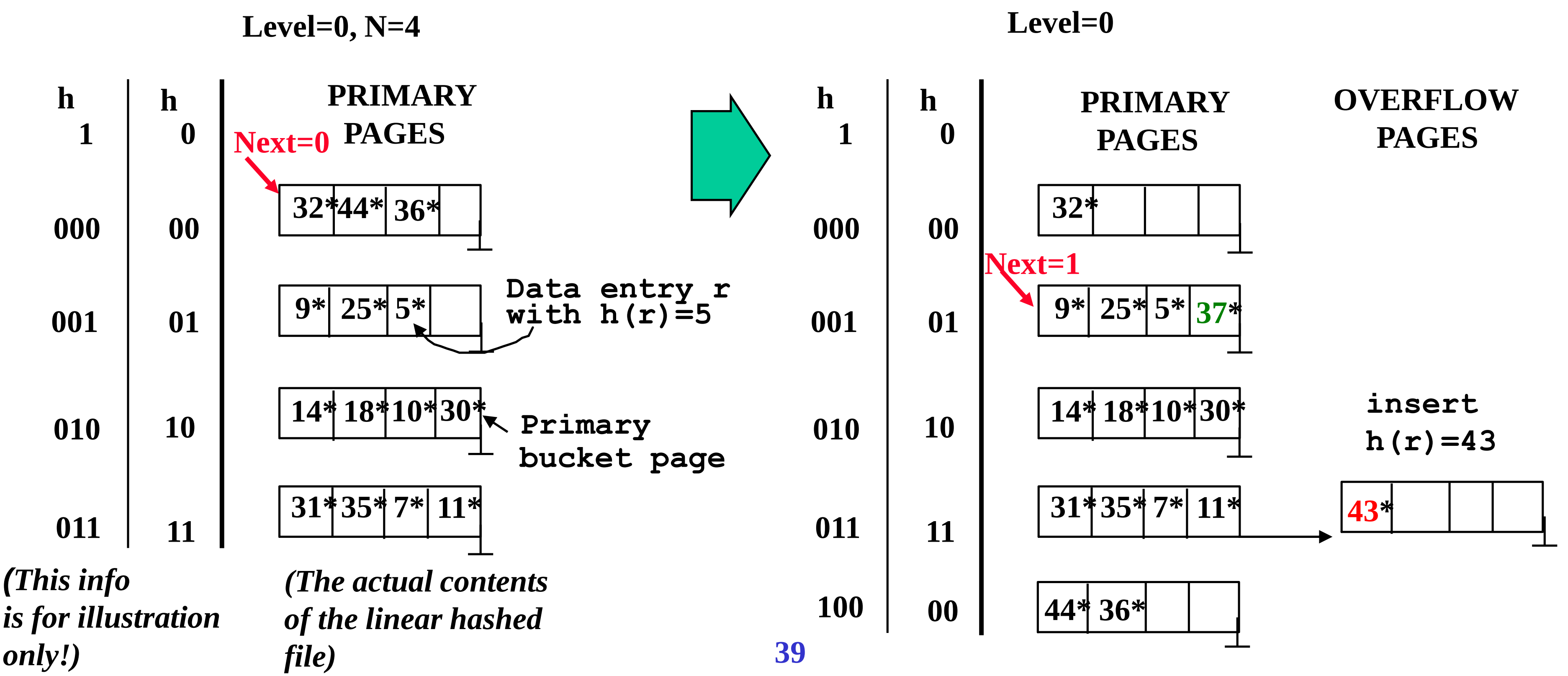
- In the middle of a round.



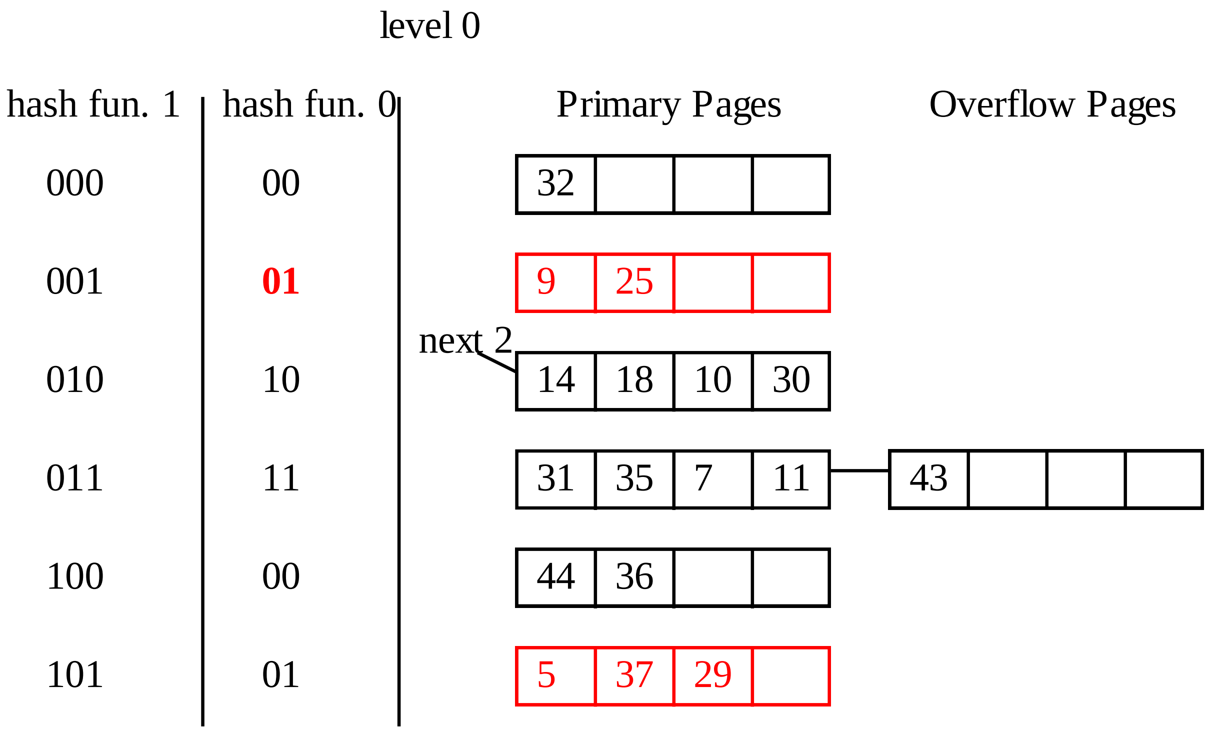
Example of Linear Hashing

- On split, $\mathbf{h}_{Level+1}$ is used to re-distribute entries.

- insert 43 (101011)
- insert 37(..101),
- insert 29 (..101)

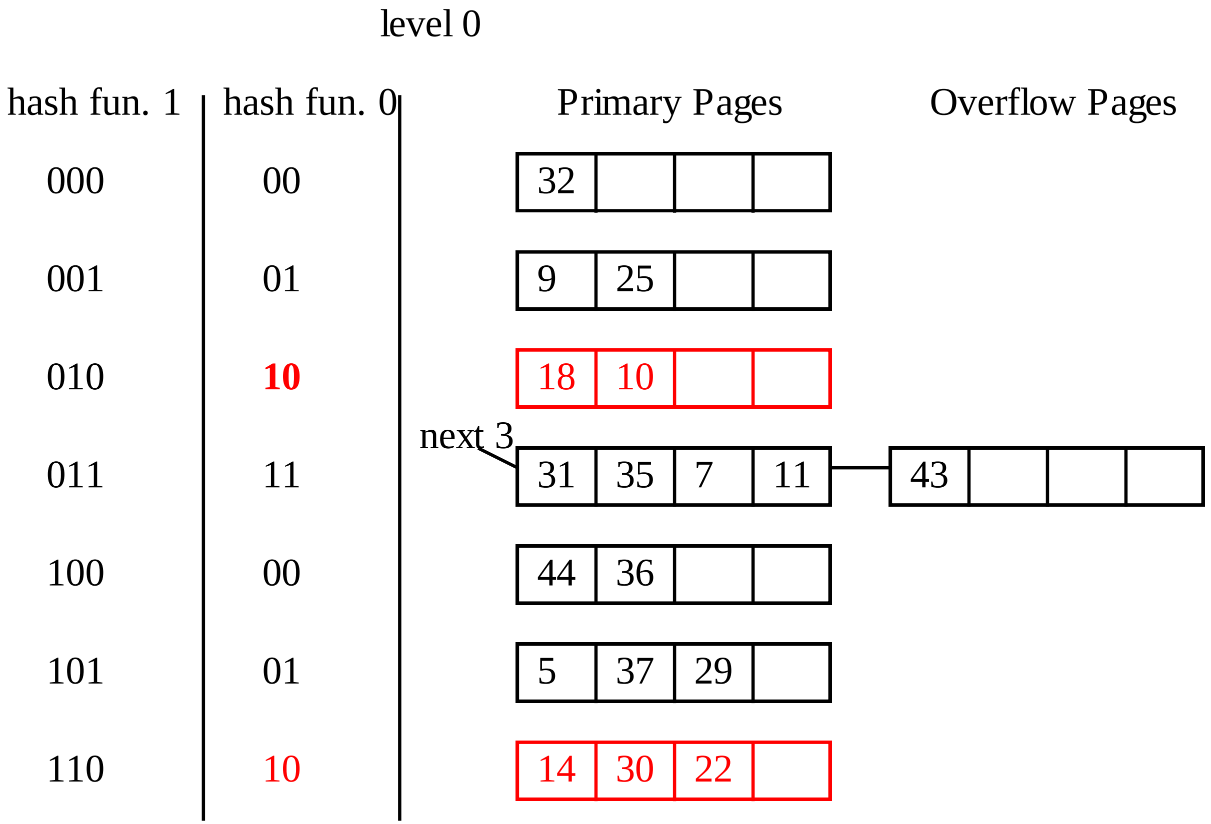


After inserting 29: 11101



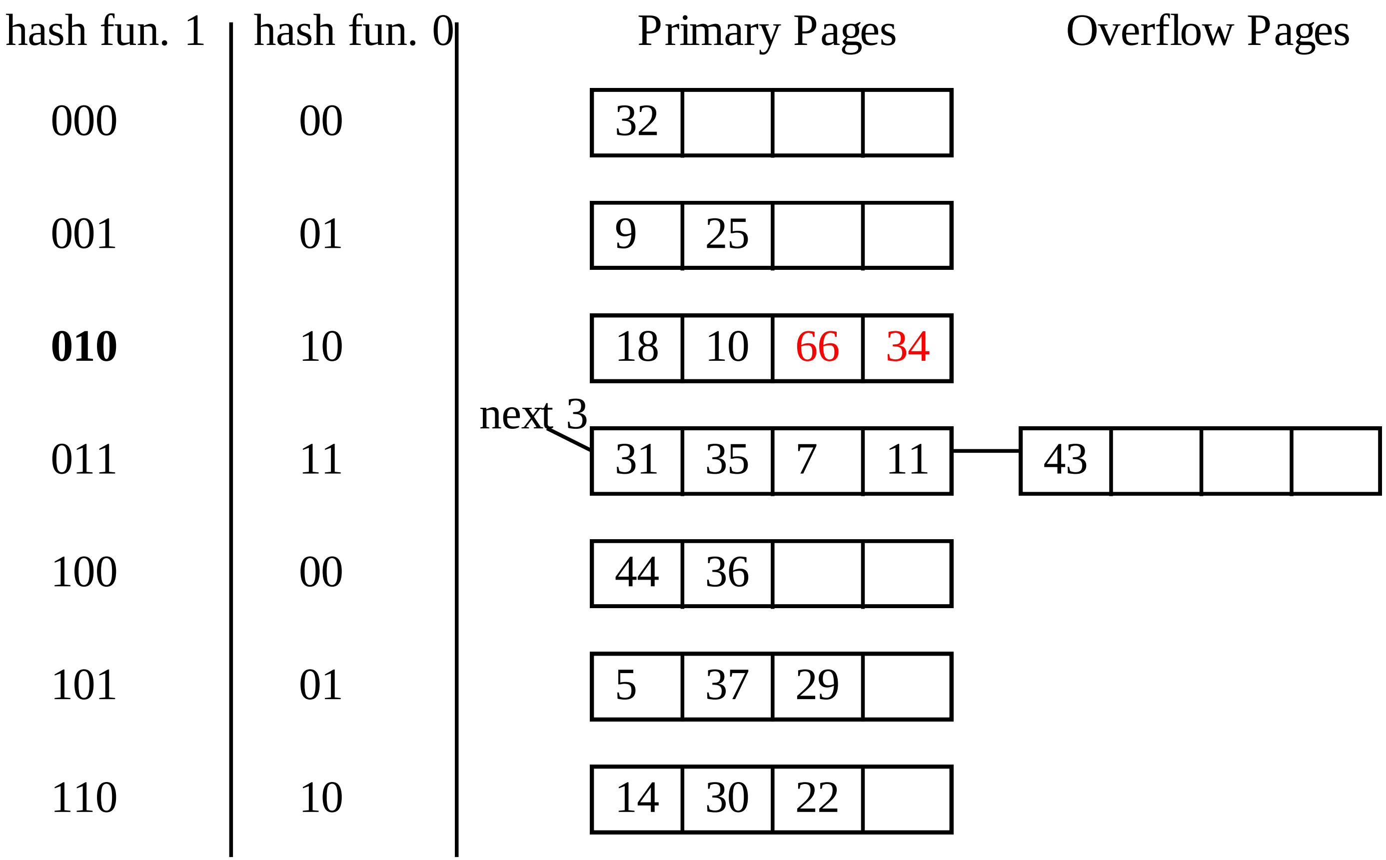
LETS INSERT 22: 10110

After inserting 22: 10110



LETS INSERT 66: 1000010 AND 34: 100010

After inserting 66: 1000010   AND 34: 100010



LETS INSERT 50: 110010

After inserting 50: 110010