COP 4710: Database Systems
Spring 2006

CHAPTER 10 – Indexing

Instructor : Mark Llewellyn
markl@cs.ucf.edu
CSB 242, 823-2790
http://www.cs.ucf.edu/courses/cop4710/spr2006

School of Electrical Engineering and Computer Science
University of Central Florida

Basic Concepts Behind Indexing

- Indexing mechanisms used to speed up access to desired data.
  - E.g., author catalog in library
- **Search Key** - attribute to set of attributes used to look up records in a file.
- An **index file** consists of records (called **index entries**) of the form

search-key	pointer

- Index files are typically much smaller than the original file.
- Two basic kinds of indices:
  - **Ordered indices:** search keys are stored in sorted order
  - **Hash indices:** search keys are distributed uniformly across "buckets" using a "hash function".

Index Evaluation Metrics

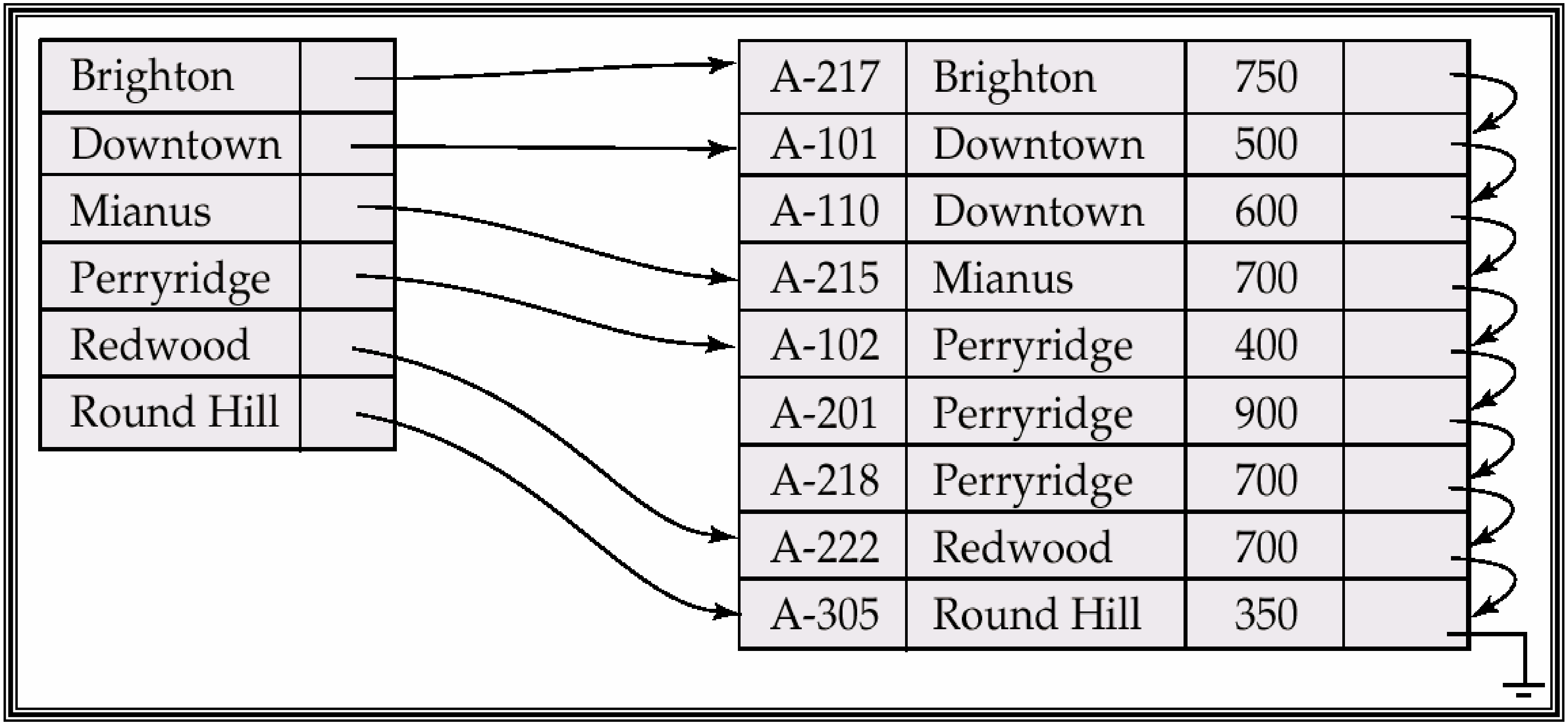
- Access types: The types of access that are efficiently supported.
  - Finding records with a specified attribute value.
  - Finding records with an attribute value falling in a specified range of values.
- Access time: The time required to find a particular data item, or set of items.
- Insertion time: The time it takes to insert a new data item. This value includes the time required to find the correct place to insert, as well as the time required to update the index structure.
- Deletion time: The time it takes to delete a data item. This value includes the time required to find the item, as well as the time required to update the index structure.
- Space overhead: The additional space required by an index structure. Provided that the amount of additional space is moderate, it is usually worthwhile to sacrifice the space to achieve improved performance.

Ordered Indices

- In an **ordered index,** index entries are stored sorted on the search key value. E.g., author catalog in library.
- **Primary index:** in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
  - Also called clustering index
  - The search key of a primary index is usually but not necessarily the primary key.
- **Secondary index**: an index whose search key specifies an order different from the sequential order of the file. Also called non-clustering index**.**
- Index-sequential file**:** ordered sequential file with a primary index.
- One of the oldest index schemes used in database systems. Designed for applications that require both sequential processing of entire files as well as random access to individual records.

Dense Index Files

- Dense index — Index record appears for every search-key value in the file.



Sparse Index Files

- Sparse Index: contains index records for only some search-key values.
  - Applicable when records are sequentially ordered on search-key
- To locate a record with search-key value *K* we:
  - Find index record with largest search-key value < *K*
  - Search file sequentially starting at the record to which the index record points
- Less space and less maintenance overhead for insertions and deletions.
- Generally slower than dense index for locating records.
- Good tradeoff: sparse index with an index entry for every block in file, corresponding to least search-key value in the block.

Example of Sparse Index Files

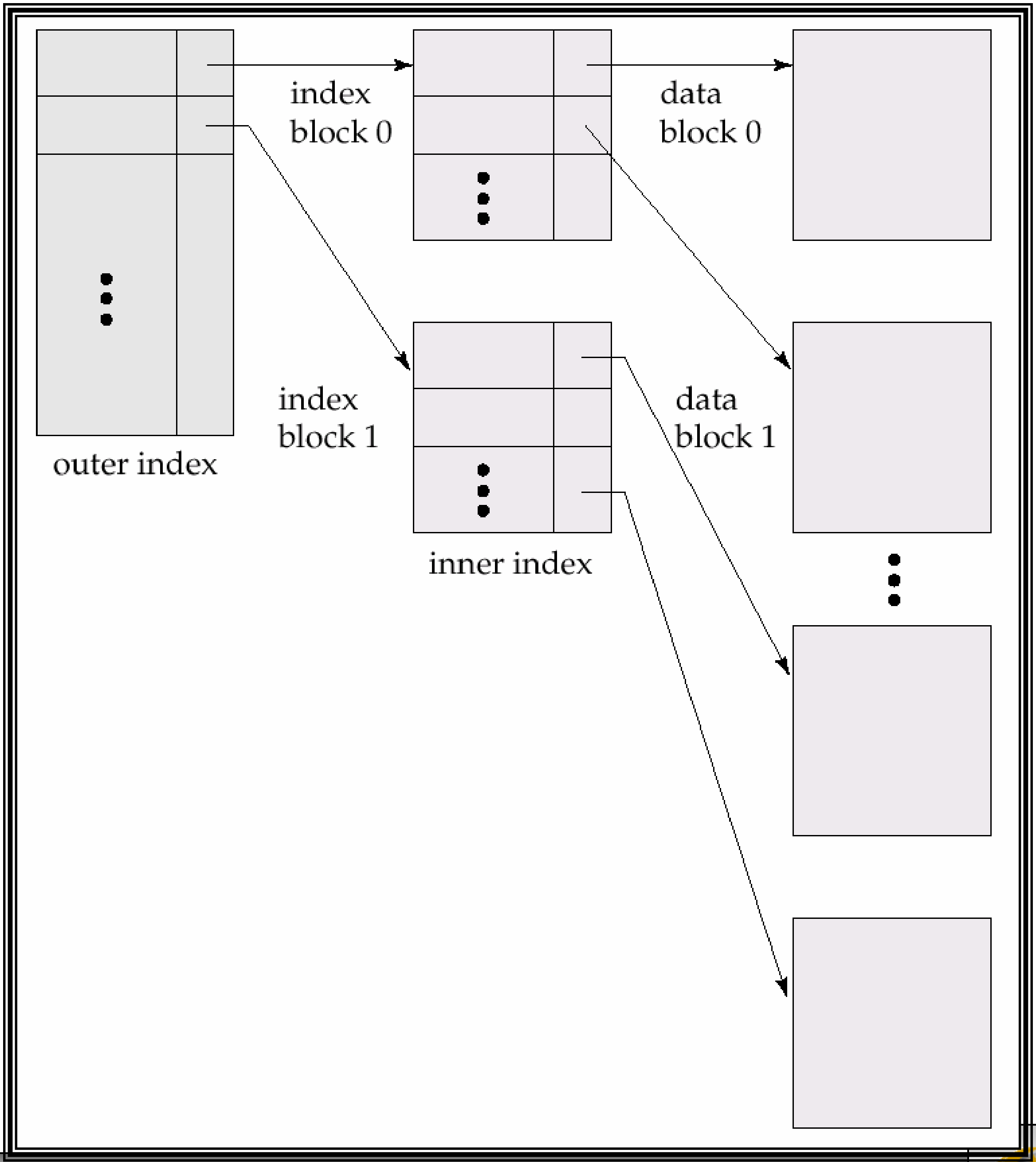
Brighton	
Mianus	
Redwood	

A-217	Brighton	750	
A-101	Downtown	500	
A-110	Downtown	600	
A-215	Mianus	700	
A-102	Perryridge	400	
A-201	Perryridge	900	
A-218	Perryridge	700	
A-222	Redwood	700	
A-305	Round Hill	350	

Multi-level Indexing

- If primary index does not fit in memory, access becomes expensive.
- To reduce number of disk accesses to index records, treat primary index kept on disk as a sequential file and construct a sparse index on it.
  - outer index – a sparse index of primary index
  - inner index – the primary index file
- If even outer index is too large to fit in main memory, yet another level of index can be created, and so on.
- Indices at all levels must be updated on insertion or deletion from the file.

Example of Multi-level Indexing



Index Update: Deletion

- If deleted record was the only record in the file with its particular search-key value, the search-key is deleted from the index also.
- Single-level index deletion:
  - Dense indices – deletion of search-key is similar to file record deletion.
  - Sparse indices – if an entry for the search key exists in the index, it is deleted by replacing the entry in the index with the next search-key value in the file (in search-key order). If the next search-key value already has an index entry, the entry is deleted instead of being replaced.

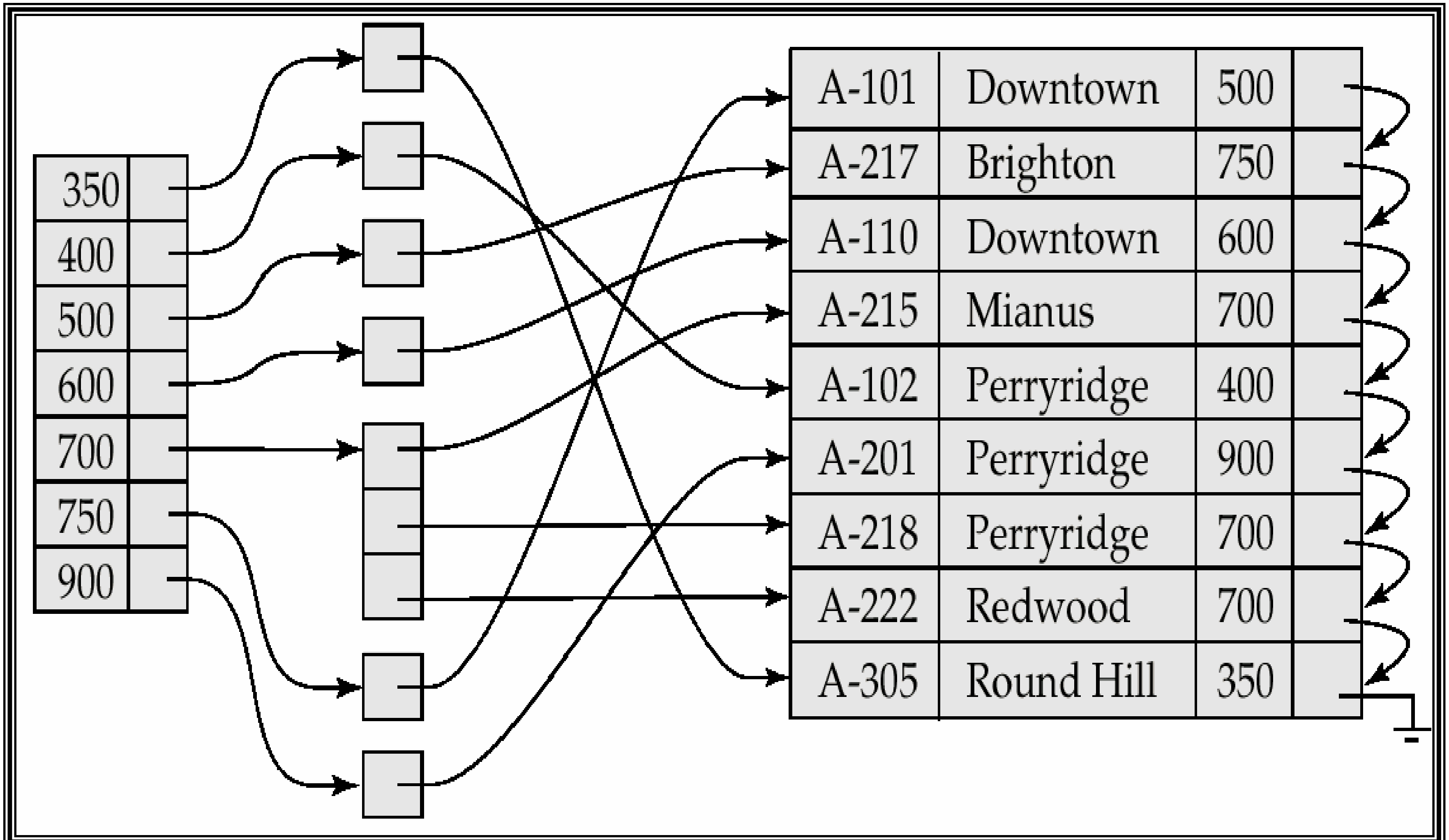
Index Update: Insertion

- Single-level index insertion:
  - Perform a lookup using the search-key value appearing in the record to be inserted.
  - Dense indices – if the search-key value does not appear in the index, insert it.
  - Sparse indices – if index stores an entry for each block of the file, no change needs to be made to the index unless a new block is created. In this case, the first search-key value appearing in the new block is inserted into the index.
- Multilevel insertion (as well as deletion) algorithms are simple extensions of the single-level algorithms

Secondary Indices

- Frequently, one wants to find all the records whose values in a certain field (which is not the search-key of the primary index) that satisfy some condition.
  - Example 1: In the *account* database stored sequentially by account number, we may want to find all accounts in a particular branch.
  - Example 2: as above, but where we want to find all accounts with a specified balance or range of balances.
- We can have a secondary index with an index record for each search-key value; index record points to a bucket that contains pointers to all the actual records with that particular search-key value.

Secondary Index on ***balance*** field of ***account***



Primary and Secondary Indices

- Secondary indices must be dense.
- Indices offer substantial benefits when searching for records.
- When a file is modified, every index on the file must be updated, Updating indices imposes overhead on database modification.
- Sequential scan using primary index is efficient, but a sequential scan using a secondary index is expensive
  - each record access may fetch a new block from disk

B^+-Tree Index Files

- B+-tree indices are an alternative to indexed-sequential 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/deletion overhead and space overhead.
- Advantages of B^+-trees outweigh disadvantages, and they are used extensively.

B^+-Tree Index Files (cont.)

- A B^+-tree is a rooted tree satisfying the following properties:
  - All paths from root to leaf are of the same length (i.e., all leaves are on the same level).
  - Each node that is not a root or a leaf holds *k-1* keys and *k* references to subtrees where $\lceil n/2 \rceil \leq k \leq n$
  - A leaf node holds *k–1* keys where $\lceil n/2 \rceil \leq k \leq n$
  - 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 (*k–1*) values.

B^+-Tree Node Structure

- Typical node

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

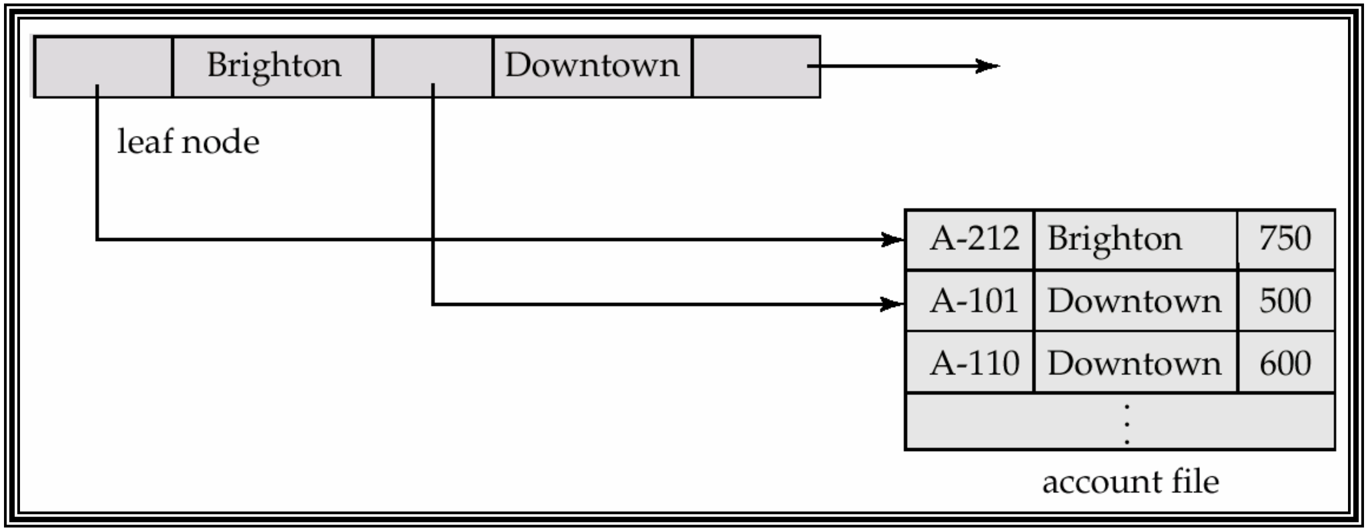
  - K_i are the search-key values
  - P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).

- The search-keys in a node are ordered

$$K_1 < K_2 < K_3 < \ldots < K_{n-1}$$

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. Only need bucket structure if search-key does not form a primary key.
- 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

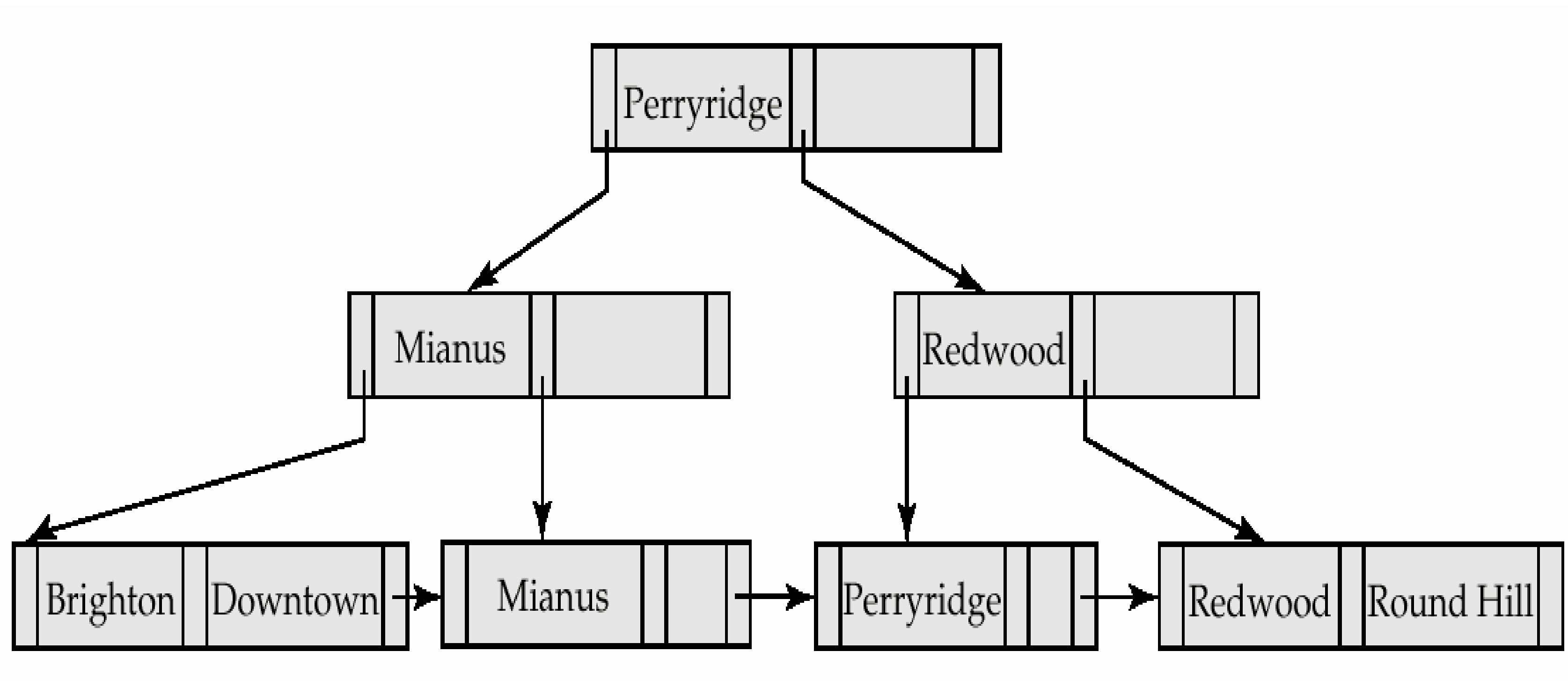


Non-Leaf Nodes in B$^+$-Trees

- 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_{m-1}

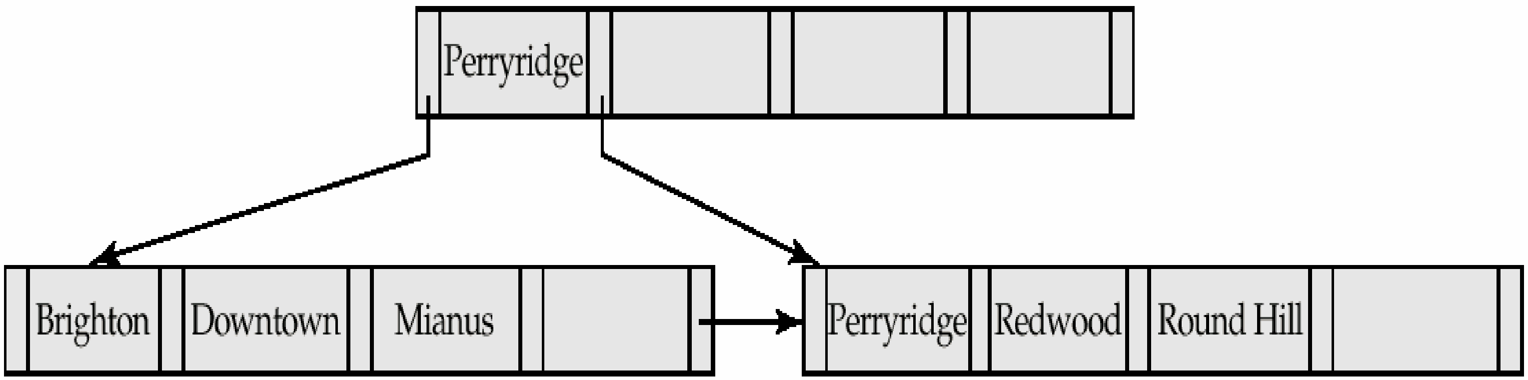
P_1	K_1	P_2	$\ldots$	P_{n-1}	K_{n-1}	P_n

Example of a B^+-tree



B^+-tree for *account* file (n = 3)

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, "logically" close blocks need not be "physically" close.
- 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 searches can be conducted efficiently.
- Insertions and deletions to the main file can be handled efficiently, as the index can be restructured in logarithmic time.

Queries on B^+-Trees

Find all records with a search-key value of *k*.

1. Start with the root node
   1. Examine the node for the smallest search-key value > *k*.
   2. If such a value exists, assume it is K_j. Then follow P_i to the child node
   3. Otherwise $k \geq K_{n-1}$, where there are *n* pointers in the node. Then follow P_n to the child node.
2. 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.
3. 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.)

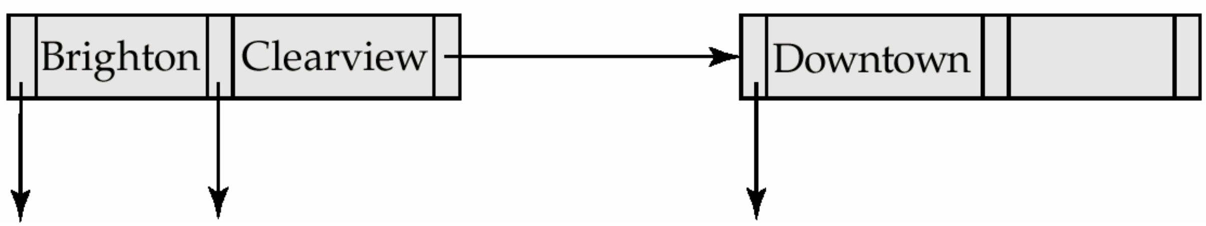
- In processing a query, a path is traversed in the tree from the root to some leaf node.
- If there are K search-key values in the file, the path is no longer than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$.
- A node is generally the same size as a disk block, typically 4 KB, and n is typically around 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.
- Contrast this with a balanced binary free with 1 million search key values — around 20 nodes are accessed in a lookup
  – above difference is significant since every node access may need a disk I/O, costing around 20 milliseconds!

Updates on B$^+$-Trees: Insertion

- Find the leaf node in which the search-key value would appear
- If the search-key value is already there in the leaf node, record is added to file and if necessary a pointer is inserted into the bucket.
- If the search-key value is not there, then add the record to the main file and create a bucket if necessary. Then:
  - If there is room in the leaf node, insert (key-value, pointer) pair in the leaf node
  - Otherwise, split the node (along with the new (key-value, pointer) entry) as discussed in the next slide.

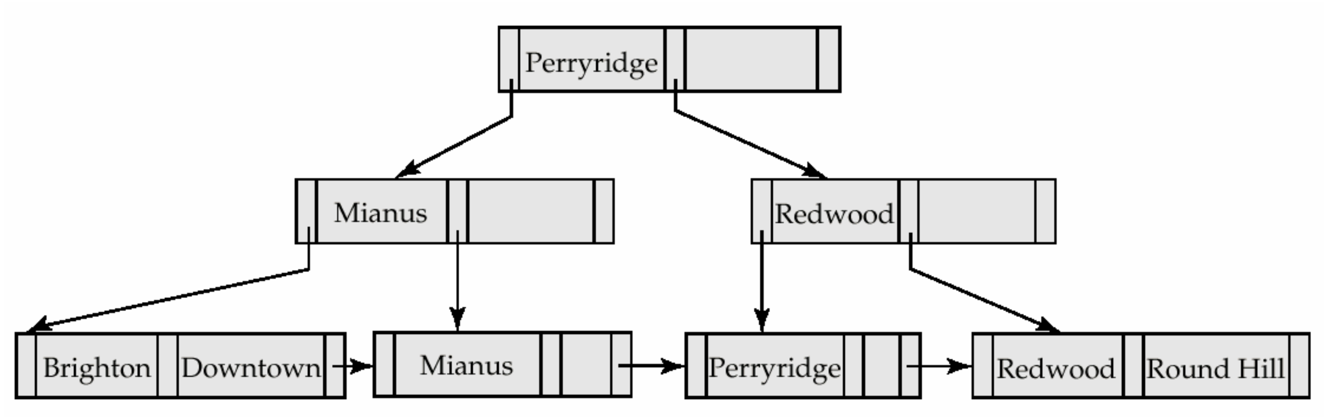
Updates on B$^+$-Trees: Insertion (cont.)

- Splitting a node:
  - take the *n*(search-key value, pointer) pairs (including the one being inserted) in sorted order. Place the first $\lceil n/2 \rceil$ in the original node, and the rest in a new node.
  - let the new node be *p*, and let *k* be the least key value in *p*. Insert (*k*,*p*) in the parent of the node being split. If the parent is full, split it and propagate the split further up.
- The splitting of nodes proceeds upwards till a node that is not full is found. In the worst case the root node may be split increasing the height of the tree by 1.

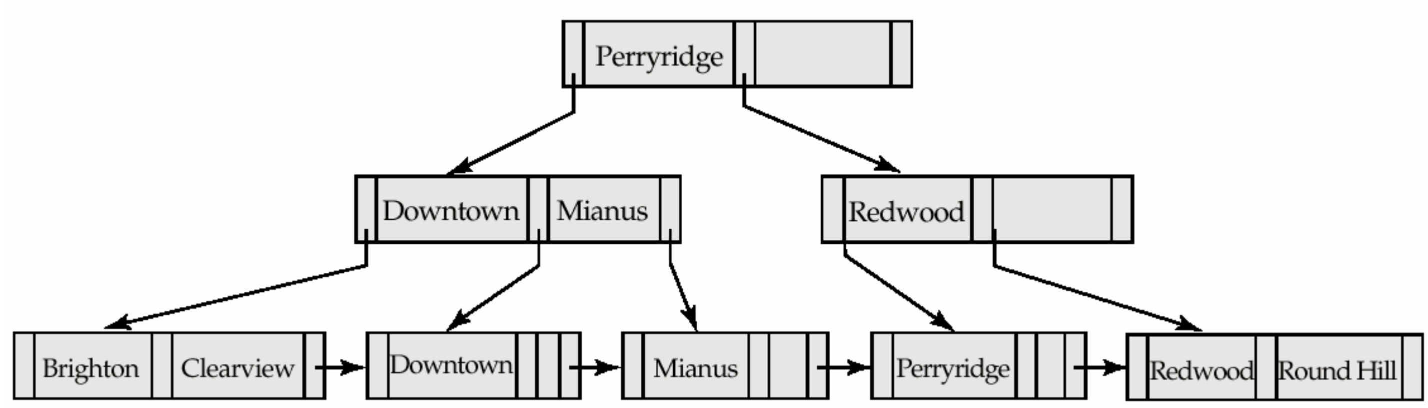


Result of splitting node containing Brighton and Downtown on inserting Clearview into the B$^+$-tree shown on page 20 (n = 3)

Updates on B+-Trees: Insertion (cont.)



before



after

Result of splitting node containing Brighton and Downtown on inserting Clearview

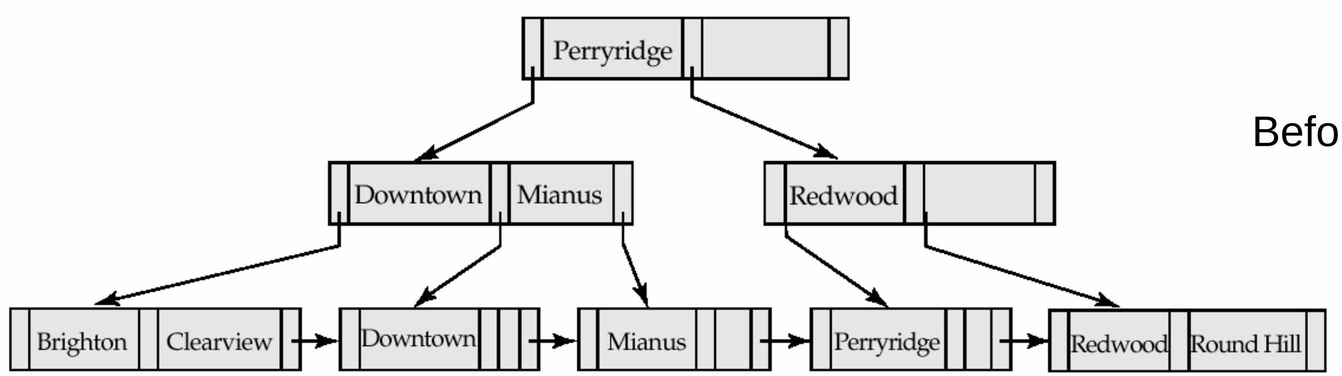
Updates on B^+-Trees: Deletion

- Find the record to be deleted, and remove it from the main file and from the bucket (if present)
- Remove (search-key value, pointer) from the leaf node if there is no bucket or if the bucket has become empty
- If the node has too few entries due to the removal, and the entries in the node and a sibling fit into a single node, then
  - Insert all the search-key values in the two nodes into a single node (the one on the left), and delete the other node.
  - Delete the pair (K_{i-1}, P_i), where P_i is the pointer to the deleted node, from its parent, recursively using the above procedure.

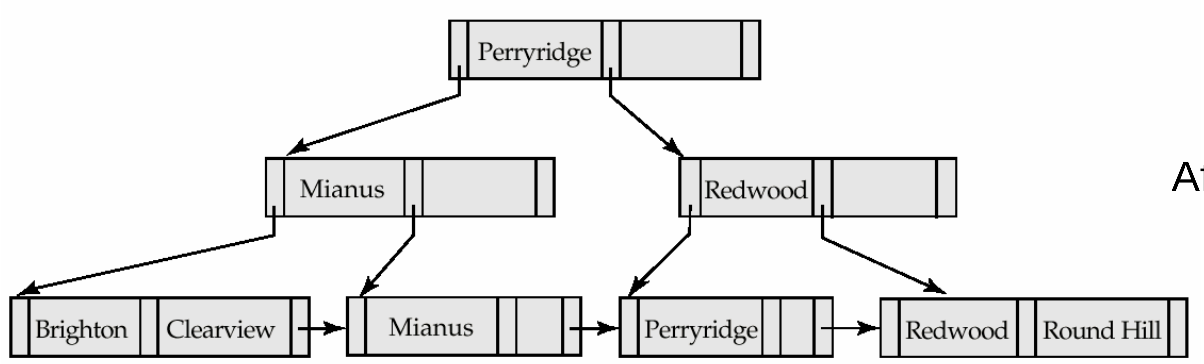
Updates on B^+-Trees: Deletion (cont.)

- Otherwise, if the node has too few entries due to the removal, and the entries in the node and a sibling fit into a single node, then
  - Redistribute the pointers between the node and a sibling such that both have more than the minimum number of entries.
  - Update the corresponding search-key value in the parent of the node.
- The node deletions may cascade upwards till a node which has $\lceil n/2 \rceil$ or more pointers is found. If the root node has only one pointer after deletion, it is deleted and the sole child becomes the root.

Examples of B^+-Tree Deletion



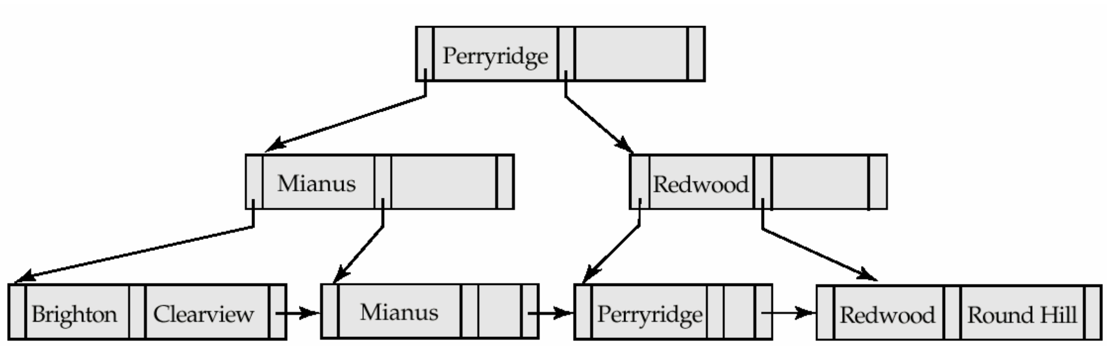
Before



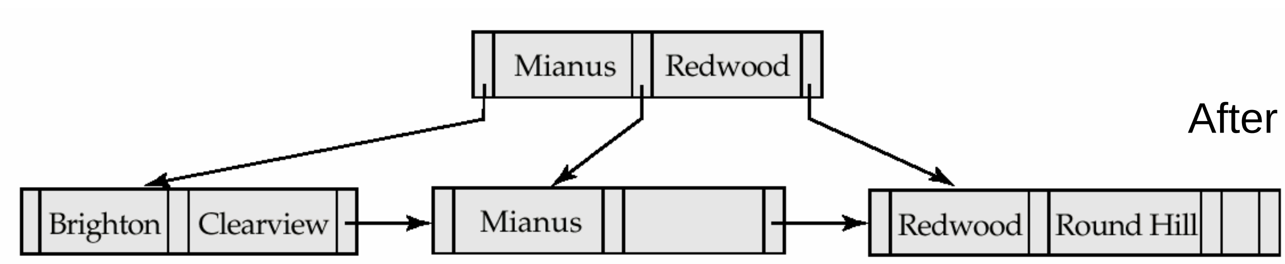
After deleting "Downtown"

The removal of the leaf node containing "Downtown" did not result in its parent having too few pointers. So the cascaded deletions stopped with the deleted leaf node's parent.

Examples of B+-Tree Deletion (cont.)



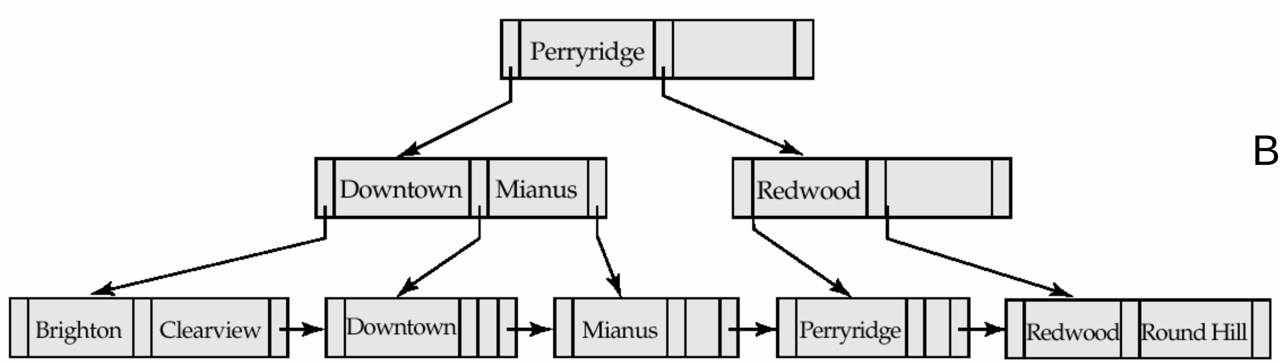
Before



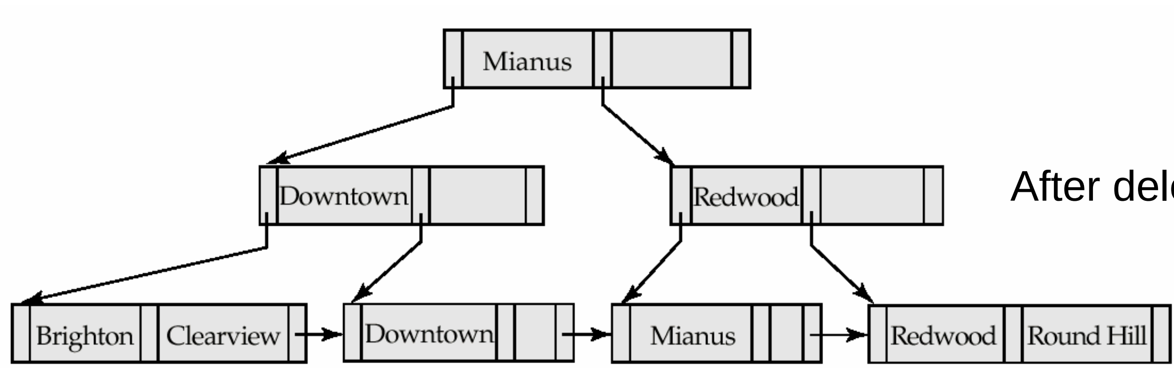
After deletion of "Perryridge"

- Node with "Perryridge" becomes underfull (actually empty, in this special case) and merged with its sibling.
- As a result "Perryridge" node's parent became underfull, and was merged with its sibling (and an entry was deleted from their parent).
- Root node then had only one child, and was deleted and its child became the new root node.

Example of B^{+}-tree Deletion (cont.)



Before



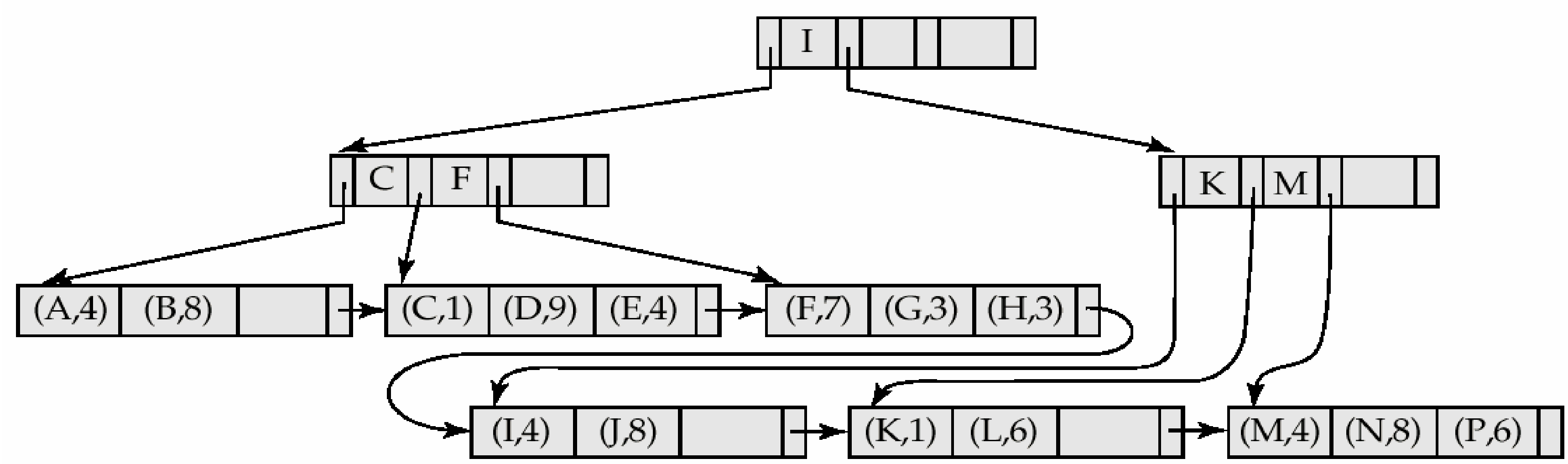
After deletion of "Perryridge"

- Parent of leaf containing Perryridge became underfull, and borrowed a pointer from its left sibling.
- Search-key value in the parent's parent changes as a result.

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.

B+-Tree File Organization (cont.)



Example of B+-tree File Organization

- Good space utilization important since records use more space than pointers.
- To improve space utilization, involve more sibling nodes in redistribution during splits and merges
  - Involving 2 siblings in redistribution (to avoid split / merge where possible) results in each node having at least $\lfloor 2n/3 \rfloor$ entries.