Indexing

These slides are a modified version of the slides of the book "Database System Concepts" (Chapter 12), 5th Ed., McGraw-Hill, by Silberschatz, Korth and Sudarshan.
Original slides are available at www.db-book.com

Basic Concepts

- 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 supported efficiently. E.g.,
  - records with a specified value in the attribute
  - or records with an attribute value falling in a specified range of values.
- Access time
- Insertion time
- Deletion time
- Space overhead

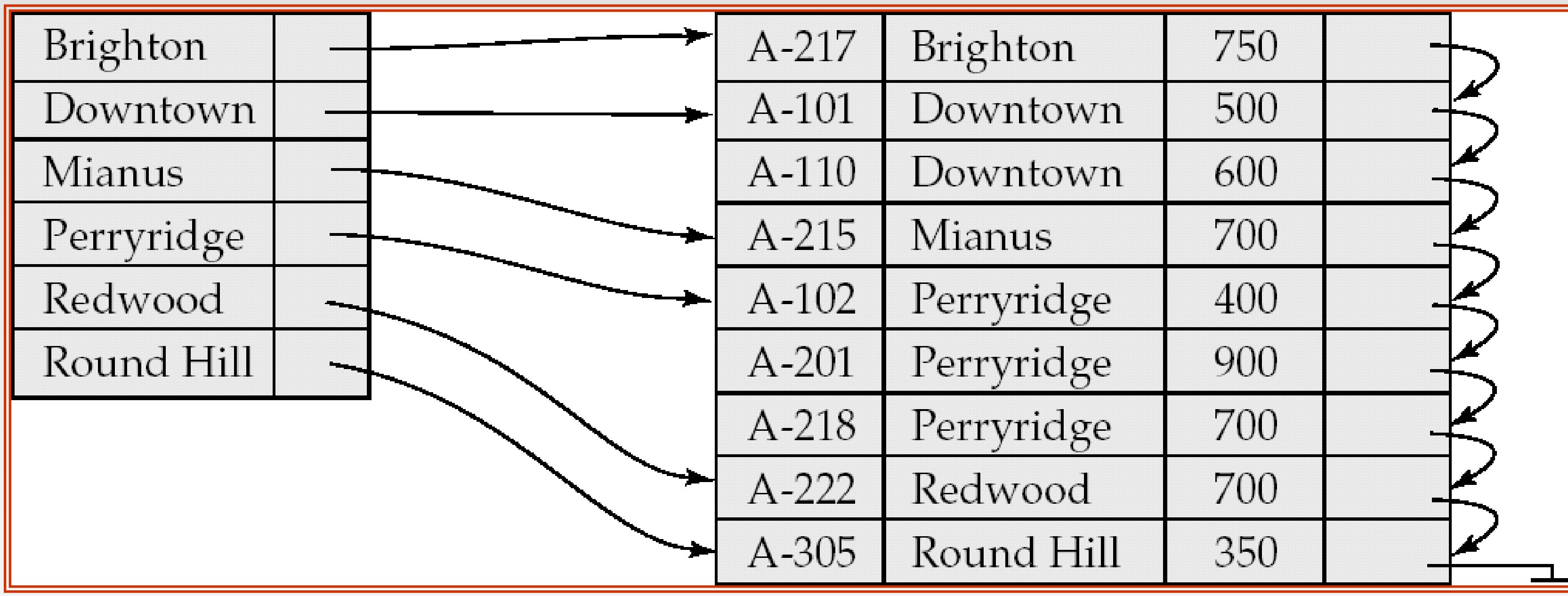
Ordered Indices

- 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.

Dense Index Files

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



Sparse Index Files

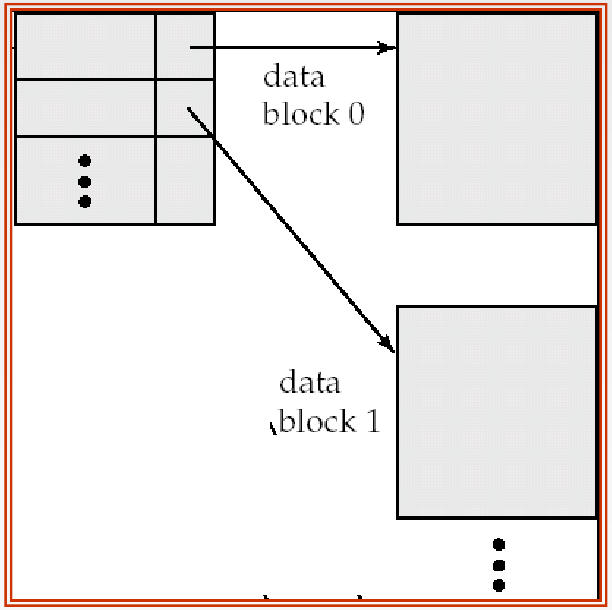
- Sparse Index: contains index records for only some search-key values.
  - Applicable when records are sequentially ordered on search-key
- To locate a record with search-key value *K* we:
  - Find index record with largest search-key value < *K*
  - Search file sequentially starting at the record to which the index record points

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	

Sparse Index Files (Cont.)

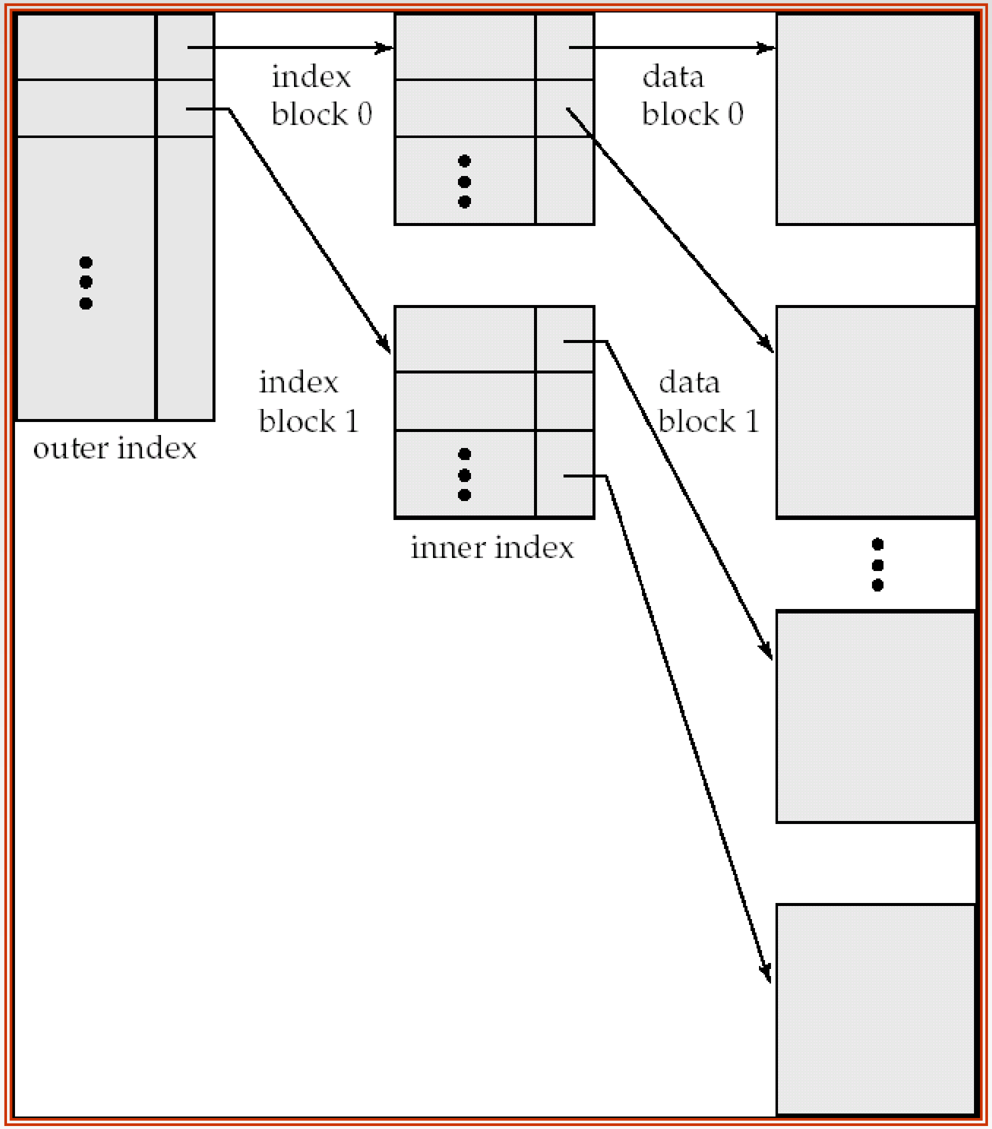
- Compared to dense indices:
  - Less space and less maintenance overhead for insertions and deletions.
  - Generally slower than dense index for locating records.
- **Good tradeoff**: sparse index with an index entry for every block in file, corresponding to least search-key value in the block.



Multilevel Index

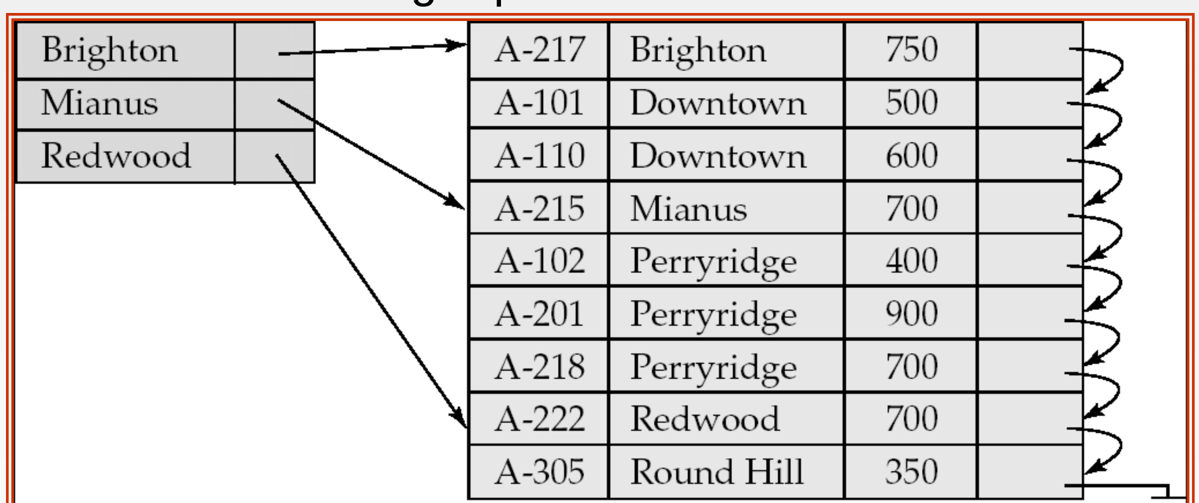
- If primary index does not fit in memory, access becomes expensive.
- Solution: treat primary index kept on disk as a sequential file and construct a sparse index on it.
  - outer index – a sparse index of primary index
  - inner index – the primary index file
- If even outer index is too large to fit 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.

Multilevel Index (Cont.)



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: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.



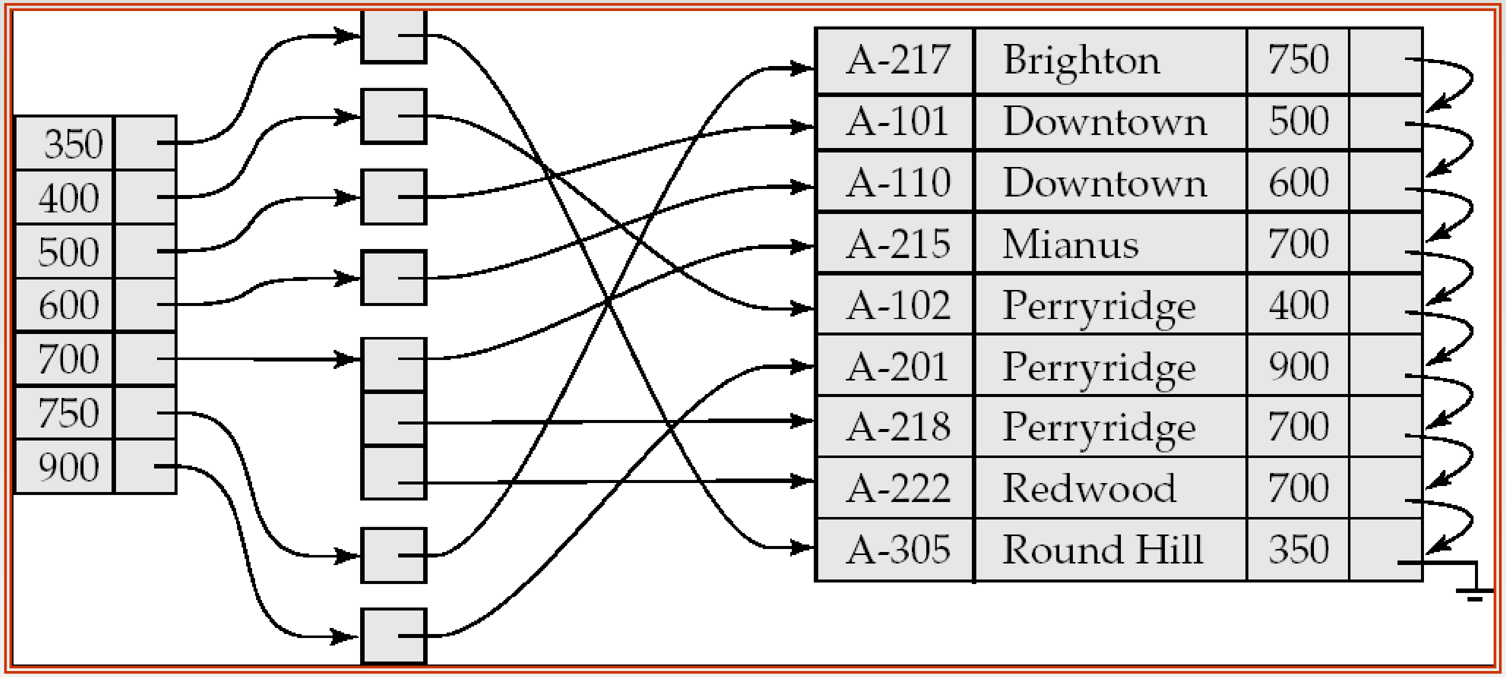
Index Update: Insertion

- 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.
    - If a new block is created, 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) satisfy some condition.
  - Example 1: In the *account* relation 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

Secondary Indices Example



Secondary index on *balance* field of *account*

- Index record points to a bucket that contains pointers to all the actual records with that particular search-key value.
- Secondary indices have to be dense

Primary and Secondary Indices

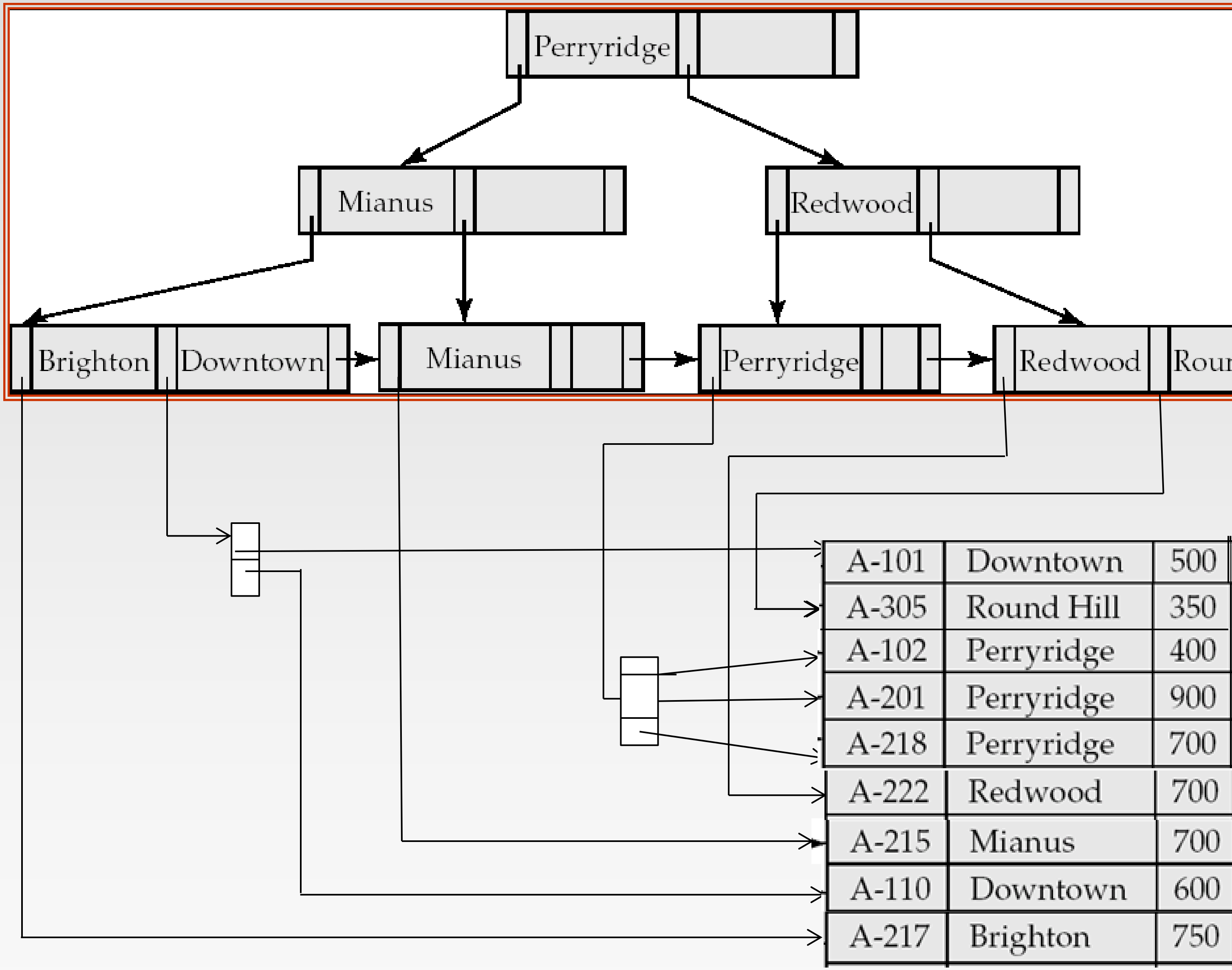
- Indices offer substantial benefits when searching for records.
- BUT: Updating indices imposes overhead on database modification -- when a file is modified, every index on the file must be updated,
- Sequential scan using primary index is efficient, but a sequential scan using a secondary index is expensive
  - Each record access may fetch a new block from disk
  - Block fetch requires about 5 to 10 milliseconds
    - versus about 100 nanoseconds for memory access

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.
- (Minor) disadvantage of B^+-trees:
  - extra insertion and deletion overhead, space overhead.
- Advantages of B^+-trees outweigh disadvantages
  - B^+-trees are used extensively

Example of a B$^+$-tree



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

account file

B^+-Tree Node Structure

- Typical node

P_1	K_1	P_2	. . .	P_{n-1}	K_{n-1}	P_n

  - a node is the same size as a disk block
  - 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)
  - n-1 maximum number of pairs (K, P) that fit in a node
  - n maximum number of pointers that fit in a node

- The search-keys in a node are ordered

$$K_1 < K_2 < K_3 < . . . < K_{n-1}$$

B^+-Tree Index Files (Cont.)

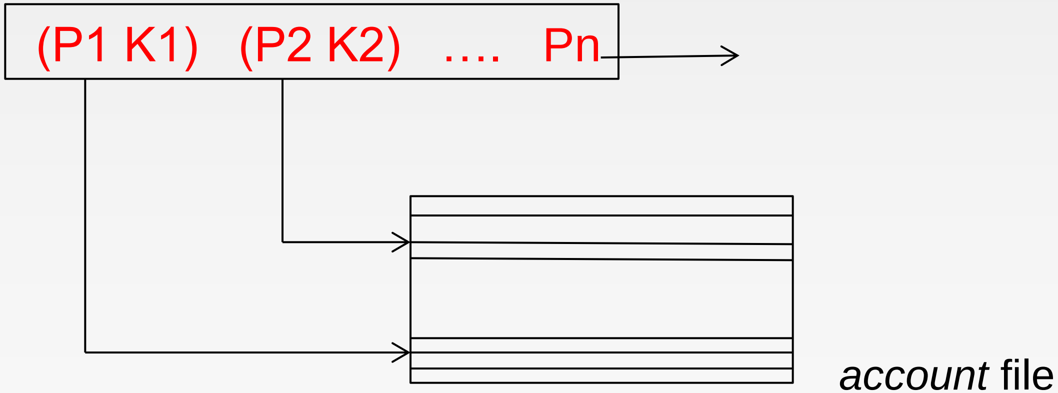
A B^+-tree is a rooted tree satisfying the following properties:

- All paths from root to leaf are of the same length
- Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children.
- A leaf node has between $\lceil (n-1)/2 \rceil$ and $n-1$ 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

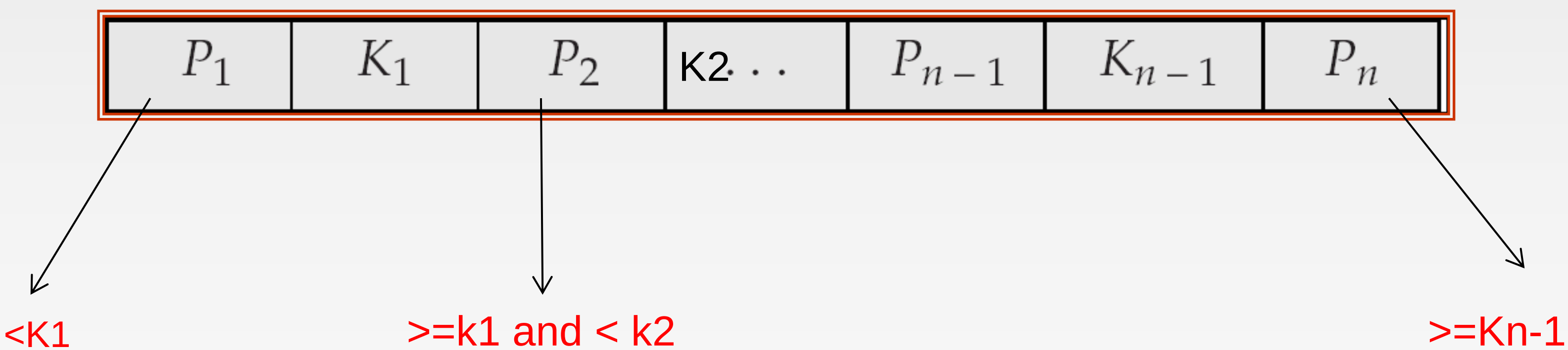
Properties of a leaf node:

- 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
- Leaves contain all the search key values

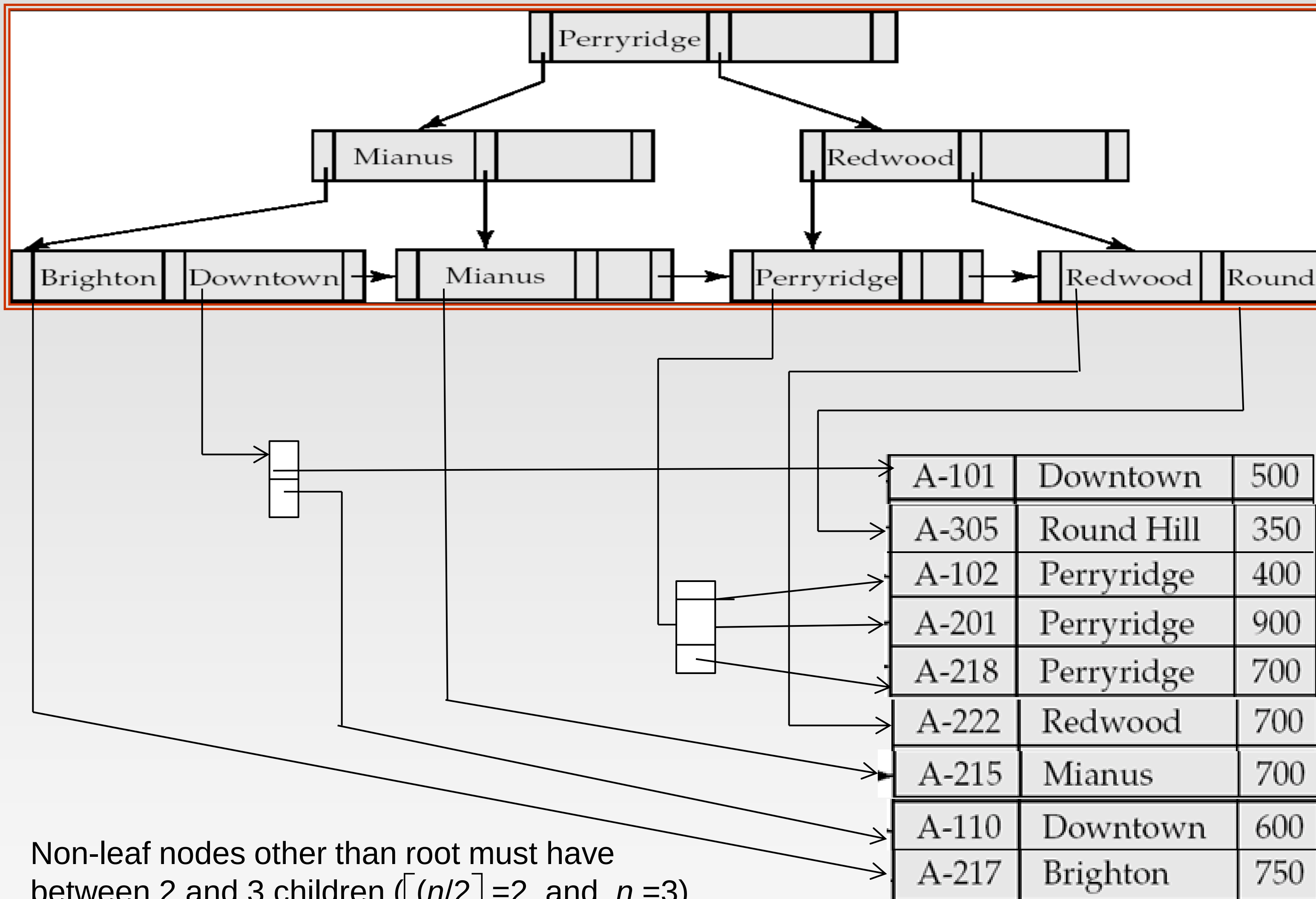


Non-Leaf Nodes in B$^+$-Trees

- 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
  - All the search-keys in the subtree to which P_n points have values greater than or equal to K_{n-1}



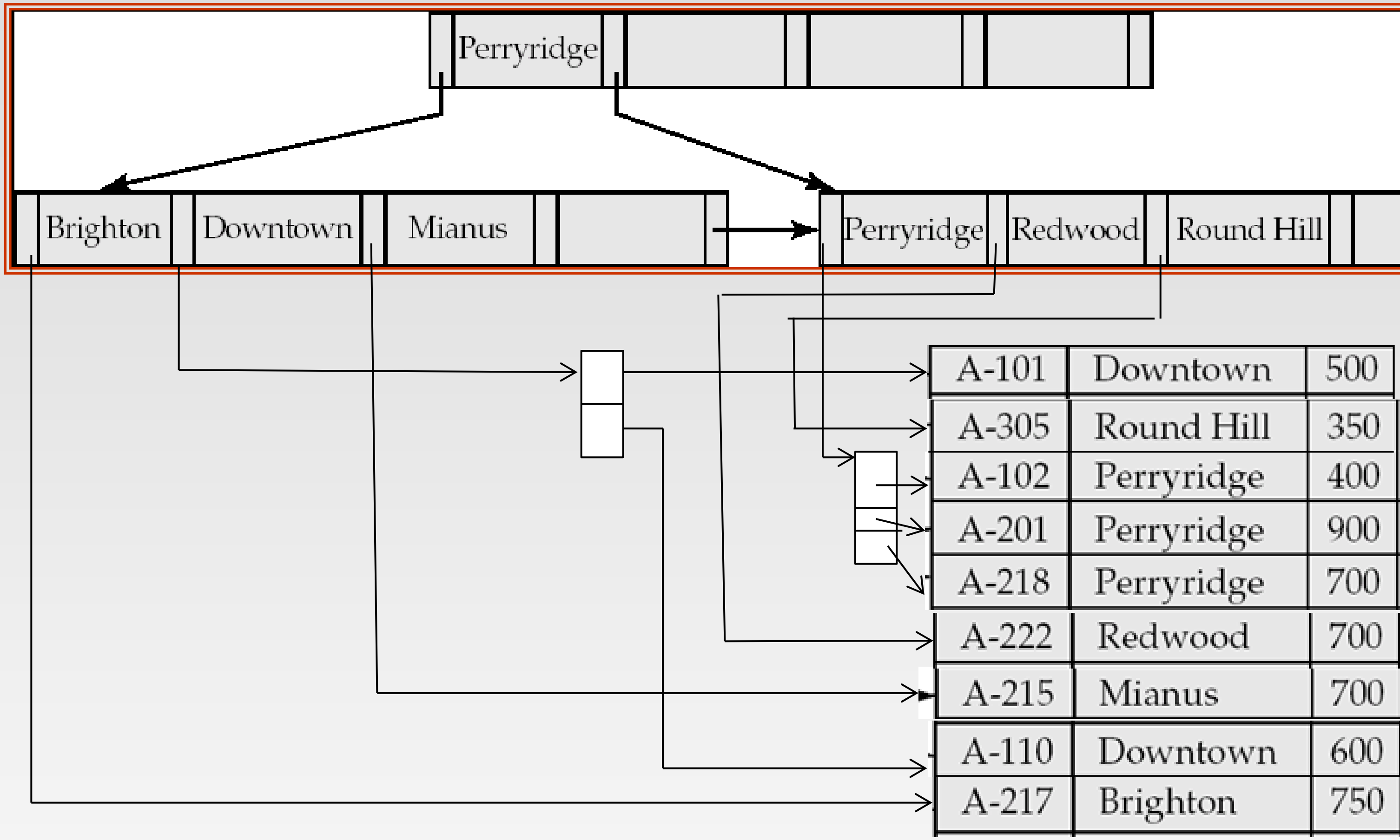
Example of a B+-tree (n=3)



account file

- Non-leaf nodes other than root must have between 2 and 3 children ($\lceil (n/2) \rceil = 2$ and $n = 3$).
- Leaf nodes must have between 1 and 2 values ($\lceil (n-1)/2 \rceil = 1$ and $n - 1 = 2$).
- Root must have at least 2 children (not leaf)
- Root must have between 0 and 2 values (leaf)

Example of B+-tree (n=5)



account file

- Non-leaf nodes other than root must have between 3 and 5 children ($\lceil n/2 \rceil = 3$ and $n = 5$).
- Leaf nodes must have between 2 and 4 values ($\lceil (n-1)/2 \rceil = 2$ and $n - 1 = 4$).
- Root must have at least 2 children (not leaf)
- Root must have between 0 and 4 values (leaf)

Observations about B^+-trees

- The non-leaf levels of the B^+-tree form a hierarchy of sparse indices.

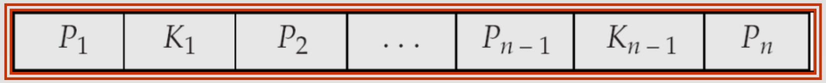
- The B^+-tree contains a relatively small number of levels
  - Level below root has at least $2*\lceil n/2 \rceil$ values
  - Next level has at least $2*\lceil n/2 \rceil * \lceil n/2 \rceil$ values
  - .. etc.
  - If there are K search-key values in the file, the tree height is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$
  - thus searches can be conducted efficiently.

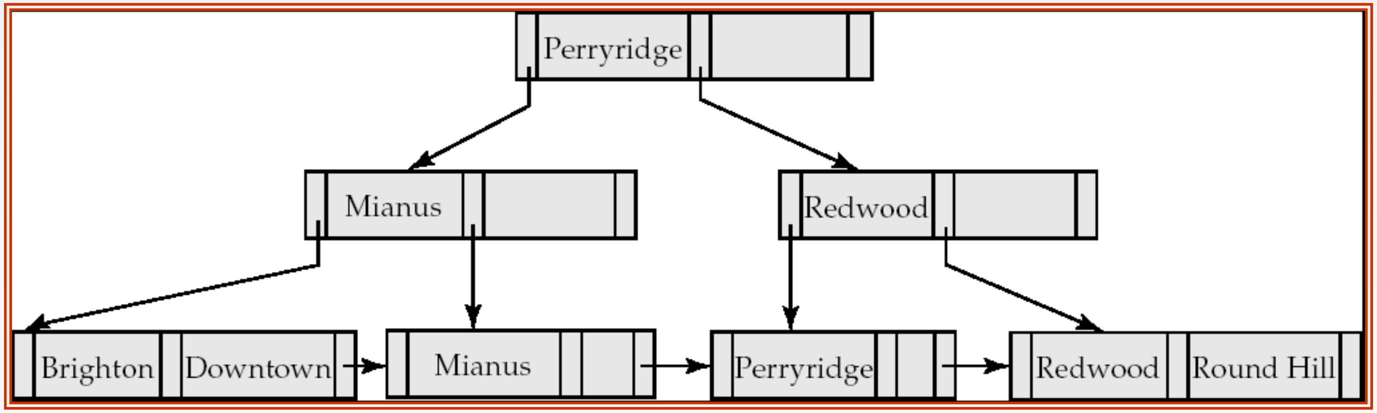
- Insertions and deletions to the main file can be handled efficiently, as the index can be restructured in logarithmic time (as we shall see).

Queries on B+-Trees

- Find all records with a search-key value of *k.*
  1. *N=root*
  2. Repeat
     1. Examine *N* for the smallest search-key value > *k.*
     2. If such a value exists, assume it is K_i. Then set $N = P_i$
     3. Otherwise $k \geq K_{n-1}$. Set $N = P_n$

     Until *N* is a leaf node
  3. If for some *i*, key $K_i = k$ follow pointer P_i to the desired record or bucket.
  4. Else no record with search-key value *k* exists.





Queries on B+-Trees (Cont.)

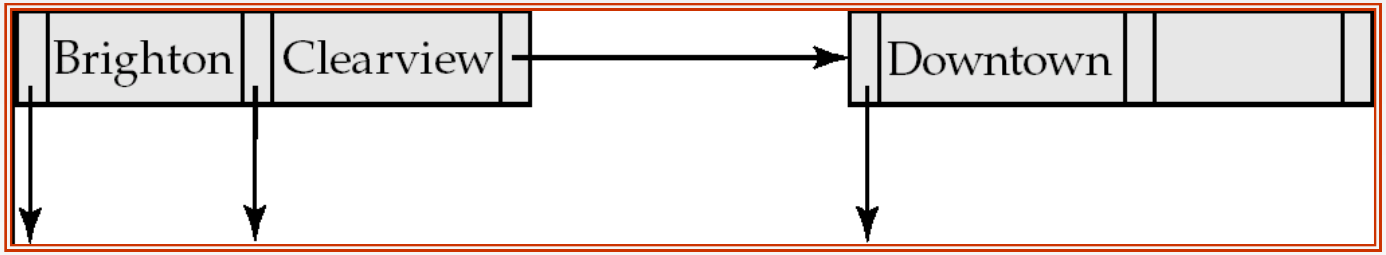
- If there are K search-key values in the file, the height of the tree is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$.
- A node is generally the same size as a disk block, typically 4 kilobytes
  - and n is typically around 100 (40 bytes per index entry).
- With 1 million search key values and $n = 100$
  - at most $log_{50}(1{,}000{,}000) = 4$ nodes are accessed in a lookup.
- Contrast this with a balanced binary tree 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

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

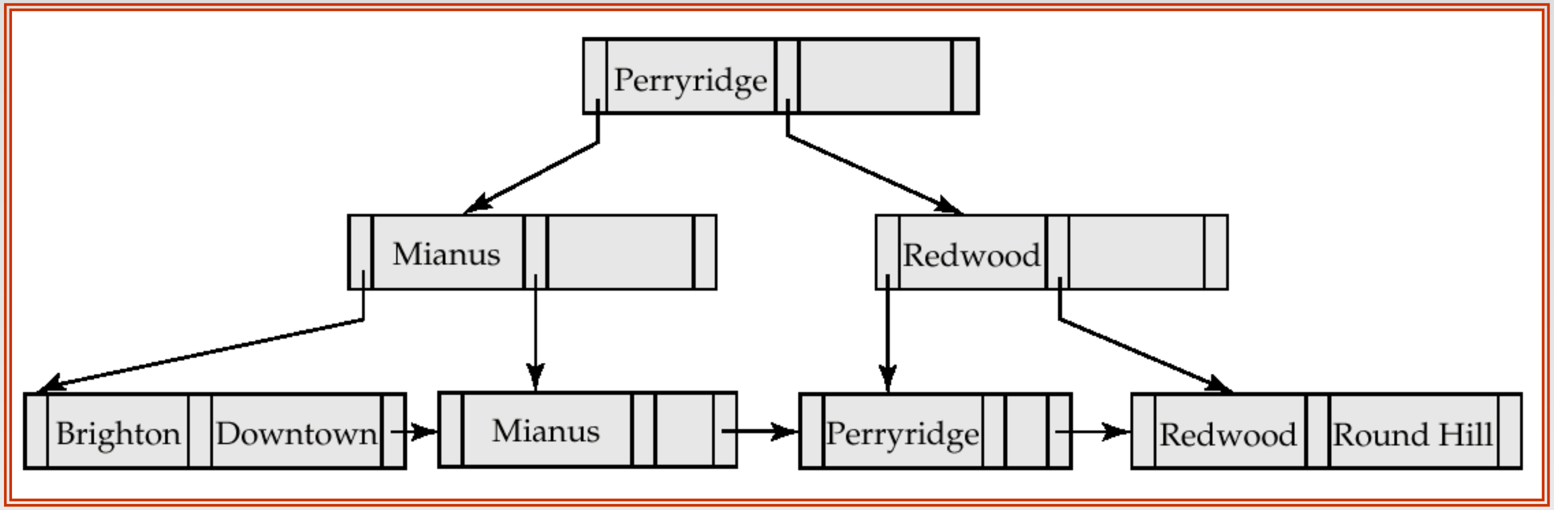
Updates on B$^+$-Trees: Insertion (Cont.)

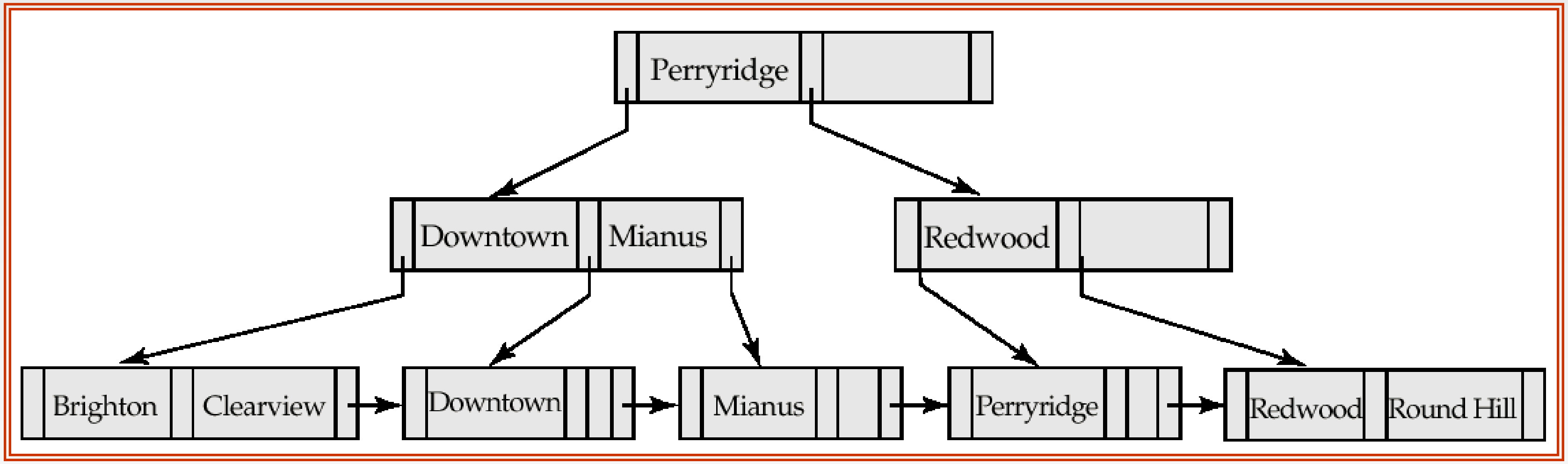
- Splitting a leaf node:
  - take the *n* (search-key value, pointer) pairs (including the one being inserted) in sorted order. Place the first $\lceil n/2 \rceil$ in the original node, and the rest in a new node.
  - let the new node be *p,* and let *k* be the least key value in *p.* Insert (*k*,*p*) in the parent of the node being split.
  - If the parent is full, split it and **propagate** the split further up.
- 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
Next step: insert entry with (Downtown,pointer-to-new-node) into parent

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

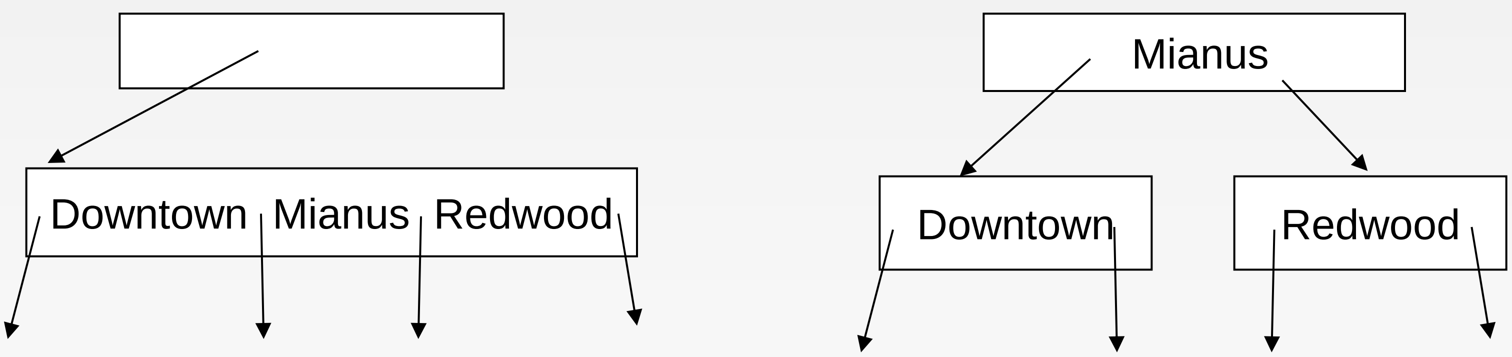




B+-Tree before and after insertion of “Clearview”

Insertion in B^+-Trees (Cont.)

- Splitting a non-leaf node: when inserting (k,p) into an already full internal node N
  - Copy N to an in-memory area M with space for n+1 pointers and n keys
  - Insert (k,p) into M
  - Copy $P_1, K_1, \ldots, \mathbf{K}_{\lceil n/2 \rceil - 1}, P_{\lceil n/2 \rceil}$ from M back into node N
  - Copy $P_{\lceil n/2 \rceil + 1}, \mathbf{K}_{\lceil n/2 \rceil + 1}, \ldots, K_n, P_{n+1}$ from M into newly allocated node N'
  - Insert ($\mathbf{K}_{\lceil n/2 \rceil}$, N') into parent of N
- **Pseudocode in book!**



Updates on B$^+$-Trees: Deletion

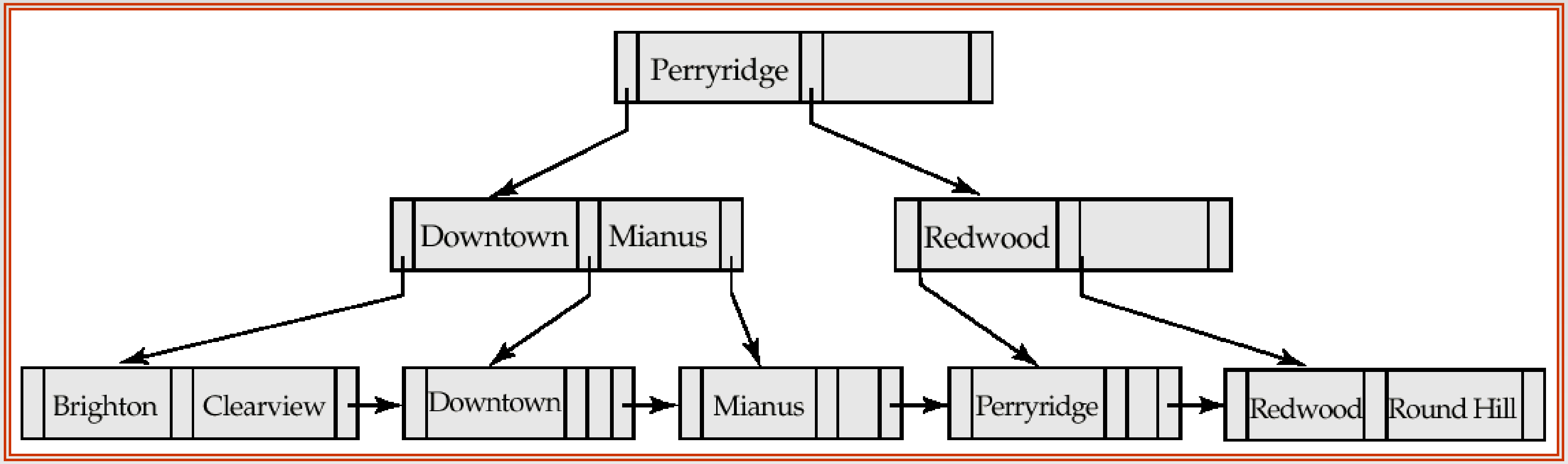
- Find the record to be deleted, and remove it from the main file and from the bucket (if present)
- Remove (search-key value, pointer) from the leaf node if there is no bucket or if the bucket has become empty

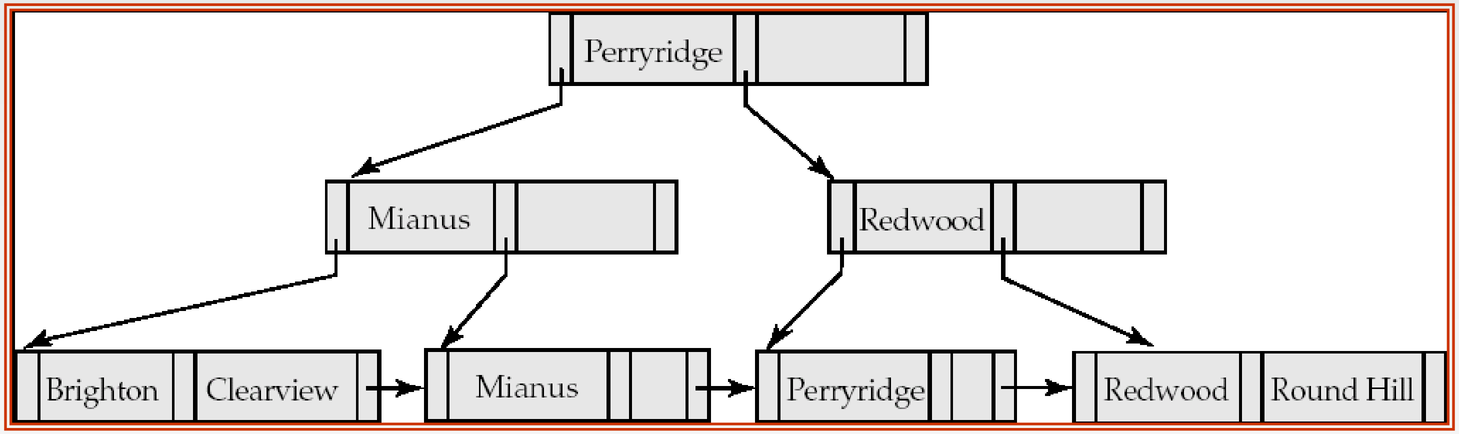
- If the node has too few entries due to the removal, and the entries in the node and a sibling fit into a single node, then ***merge siblings:***
  - Insert all the search-key values in the two nodes into a single node (the one on the left), and delete the other node.
  - Delete the pair (K_{i-1}, P_i), where P_i is the pointer to the deleted node, from its parent, recursively using the above procedure.
  - Merge of intermediate node: value separating the two nodes (at parent) moves into merged node

Updates on B^+-Trees: Deletion

- Otherwise, if the node has too few entries due to the removal, but the entries in the node and a sibling do not fit into a single node, then **redistribute pointers**:
  - 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

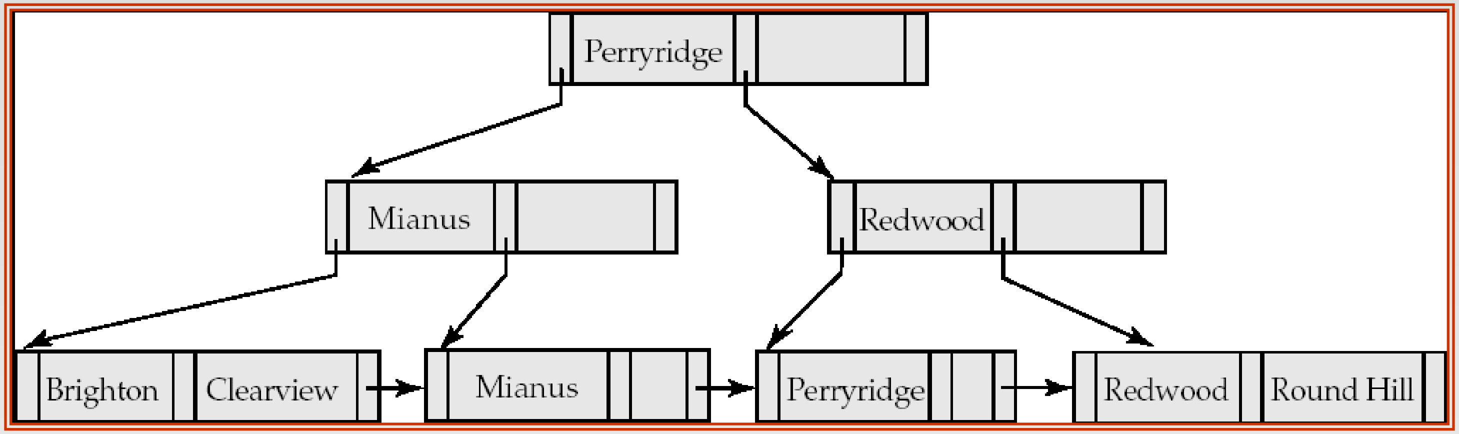


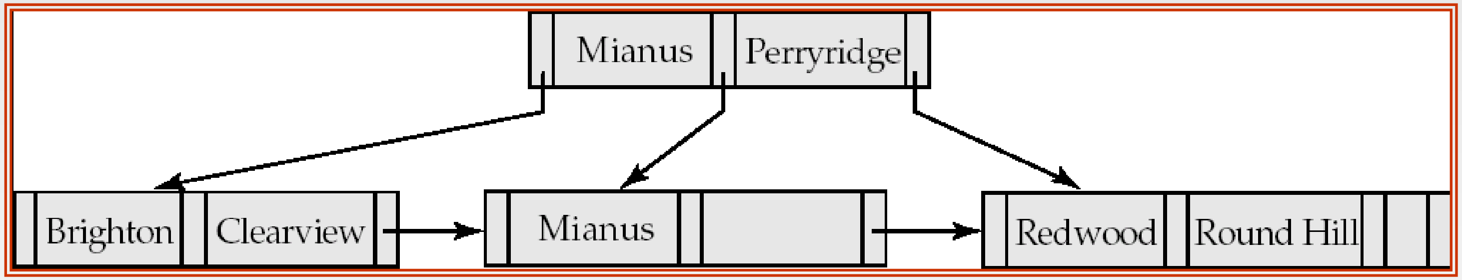


Before and after deleting "Downtown"

- Deleting "Downtown" causes merging of under-full leaves
  - leaf node can become empty only for n=3!

Examples of B+-Tree Deletion (Cont.)

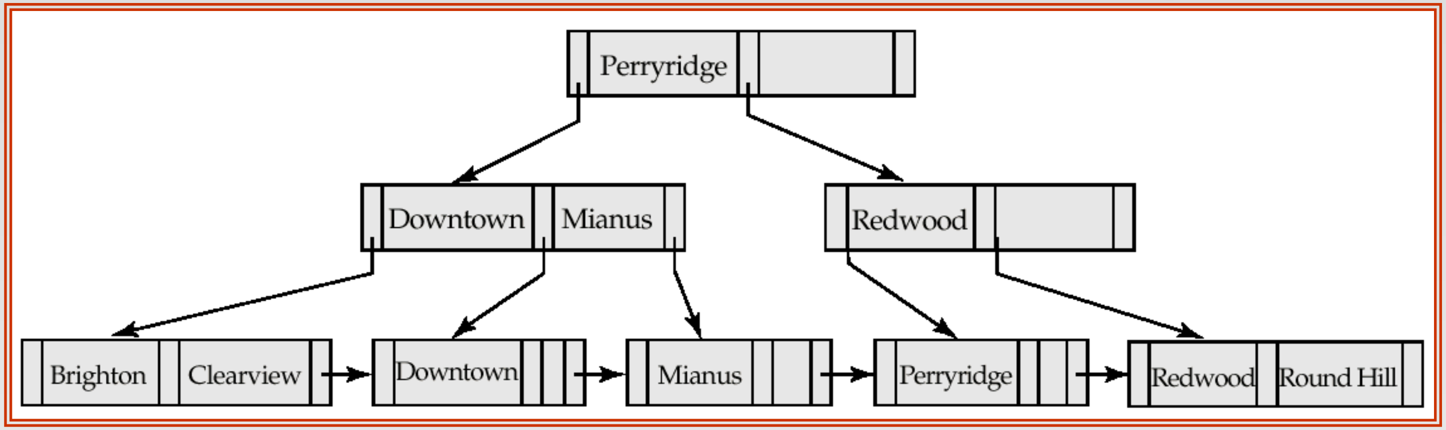


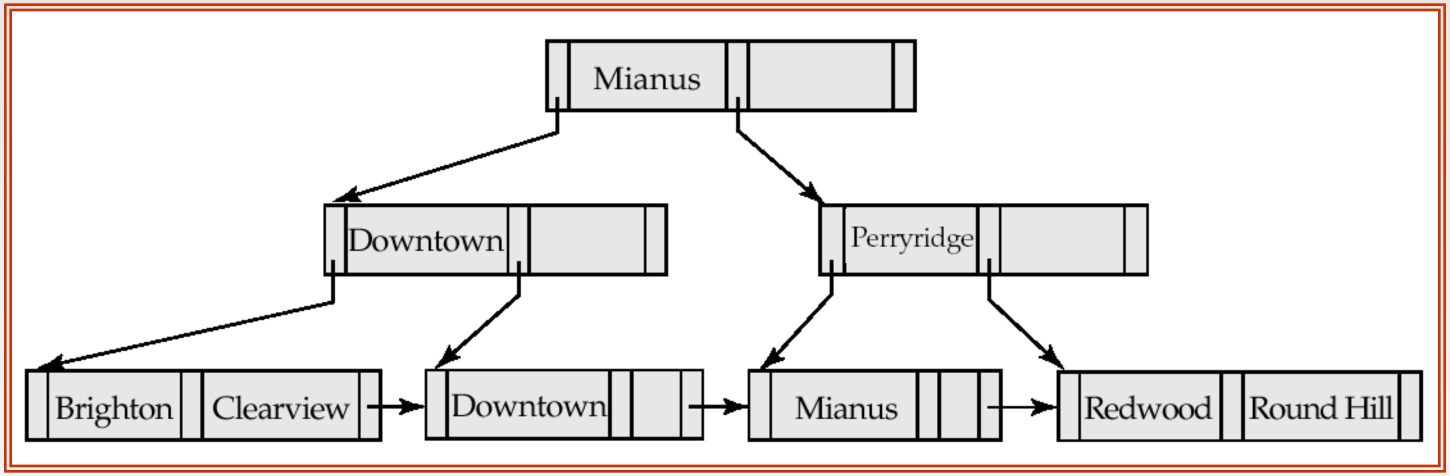


Deletion of “Perryridge” from result of previous example

- Leaf 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
  - Value separating two nodes (at parent) moves into merged node
  - Entry deleted from parent
- Root node then has only one child, and is deleted

Example of B+-tree Deletion (Cont.)





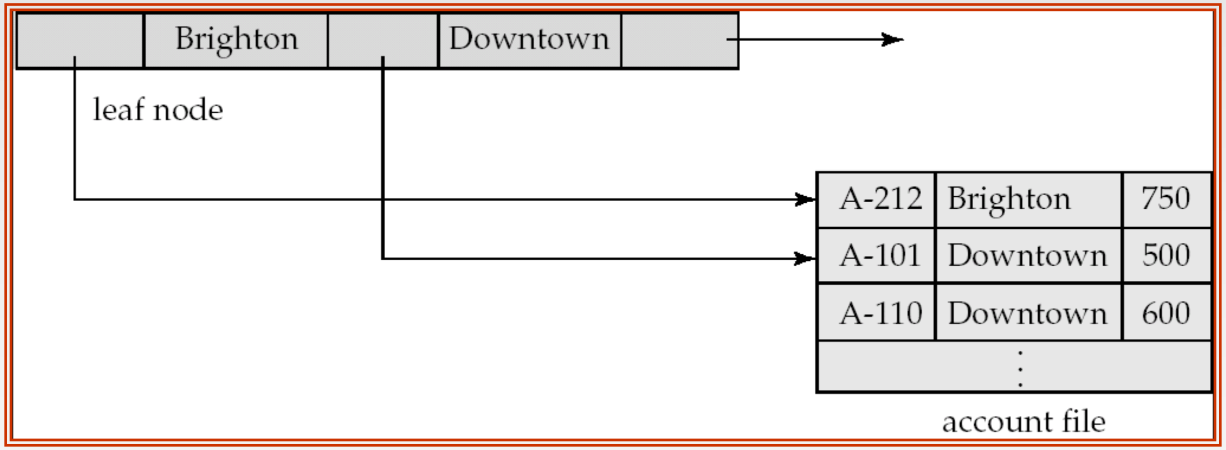
Before and after deletion of "Perryridge" from the first example

- Parent of leaf containing Perryridge became underfull, since merge is not possible, borrowed a pointer from its left sibling
- Search-key value in the parent's parent changes as a result

Example of B$^+$-tree for a sequentially ordered file

Solution 1

Leaf node: (Pi, Ki), Pi is a pointer to the first record with search key value Ki in the file

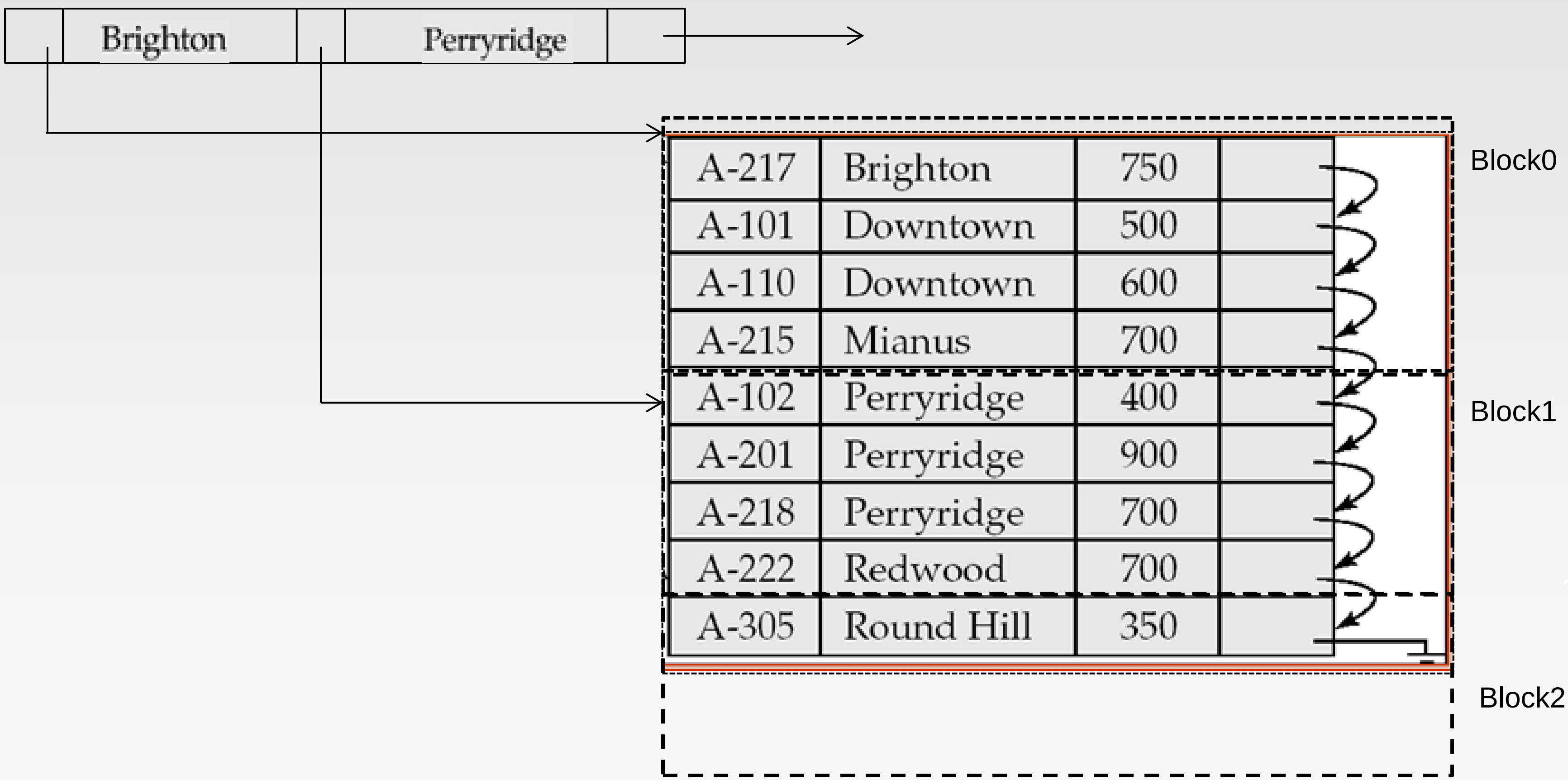


Example of B+-tree for a sequentially ordered file (cont.)

Solution 2

Leaf nodes: pointers to blocks

(Pi, Ki): Pi pointer to a block of the file, whose least search key value is Ki

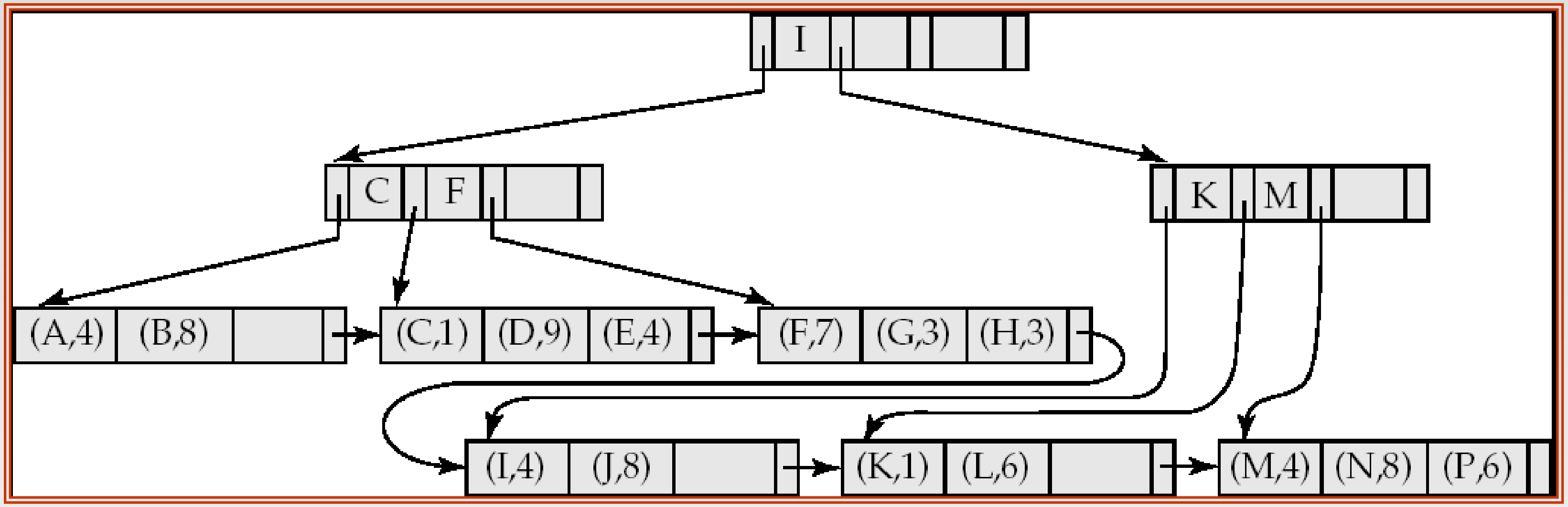


B^+-Tree File Organization

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.
- Leaf nodes are still required to be half full
  - 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.
- 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