Intro to DB

CHAPTER 12
INDEXING & HASHING

Chapter 12: Indexing and Hashing

- Basic Concepts
- Ordered Indices
- B+-Tree Index Files
- B-Tree Index Files
- Static Hashing
- Dynamic Hashing
- Comparison of Ordered Indexing and Hashing
- Index Definition in SQL
- Multiple-Key Access

Basic Concepts

- to speed up access to desired data
- **Search Key**
  - attribute (or set of attributes) used to look up records in a file
- **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

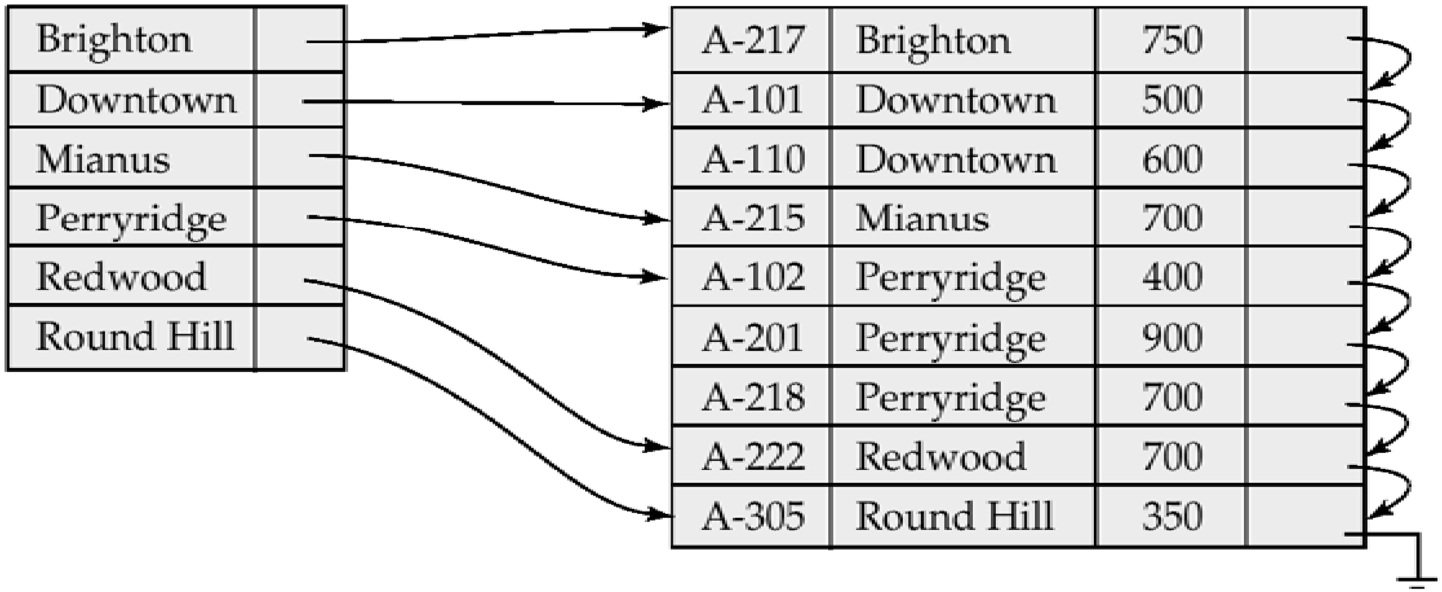
- Access types supported
  - Point queries: specific value for search key
  - Range queries: search key value falling in a specified range
- Time
  - Access time
  - Insertion time
  - Deletion time
- Space overhead

Ordered Indices

- **Primary index**
  - 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

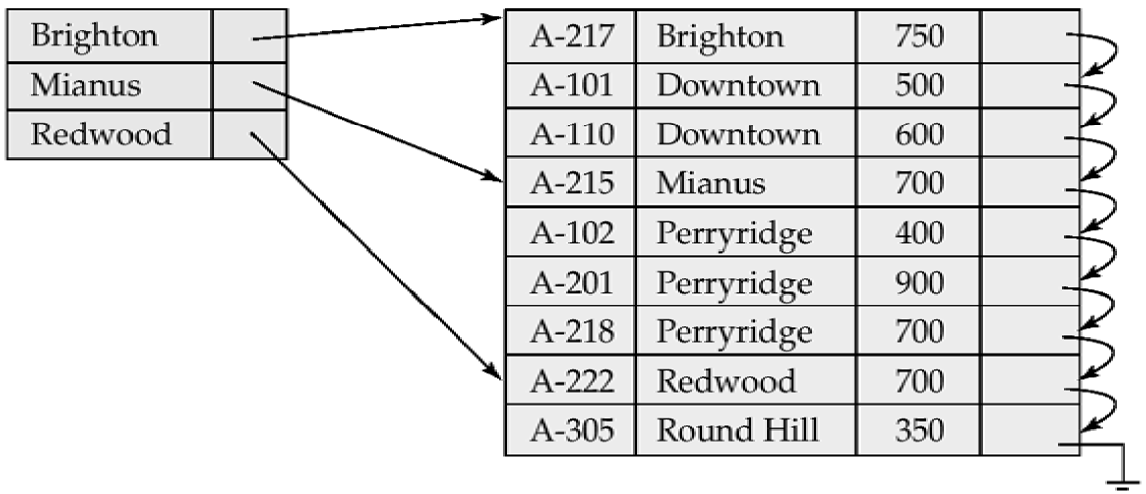
- Dense index — Index record appears for every search-key value in the file.



Sparse Index Files

Sparse Index: contains index records for only some search-key values.

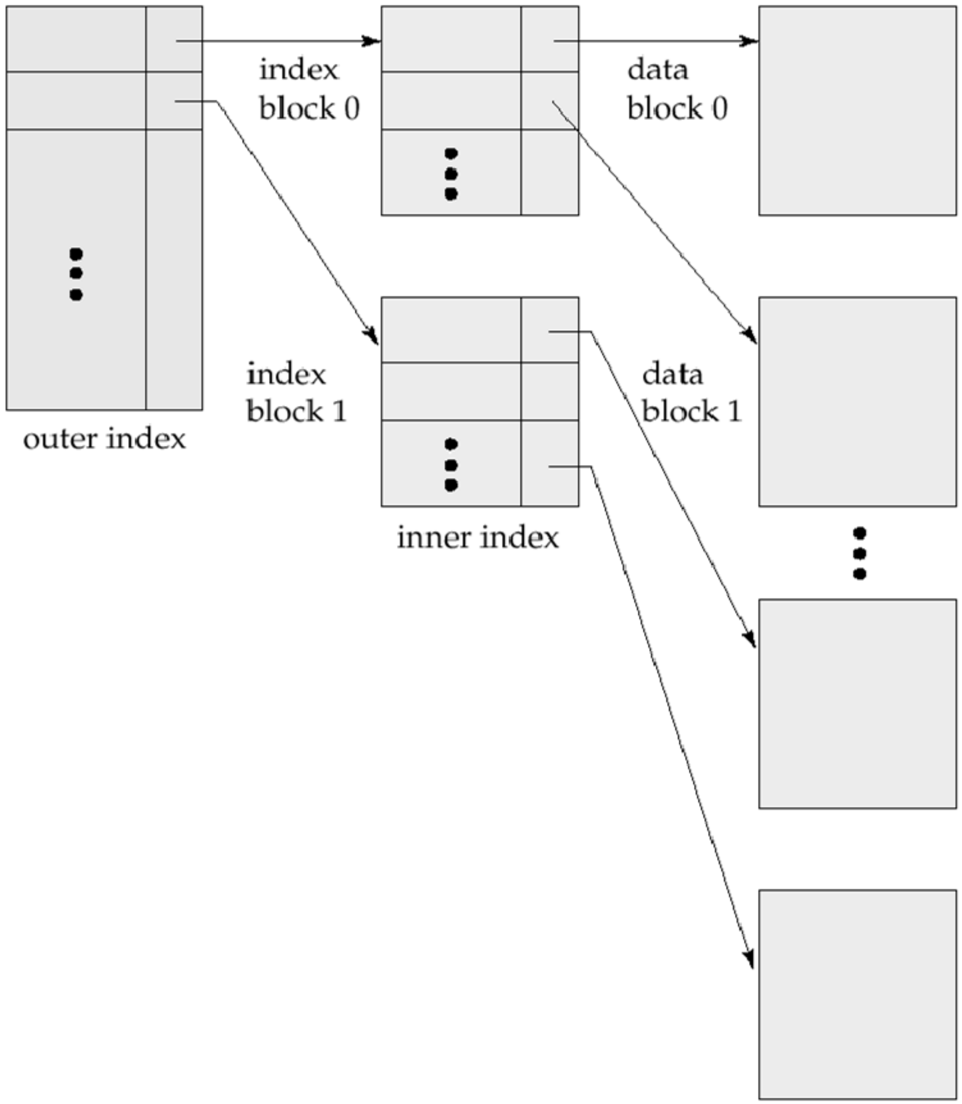
- Applicable when records are sequentially ordered on search-key
- Less space and less maintenance overhead for insertions/deletions.
- Generally slower than dense index for locating records.
- Good tradeoff: sparse index with an index entry for every block in file, corresponding to least search-key value in the block.



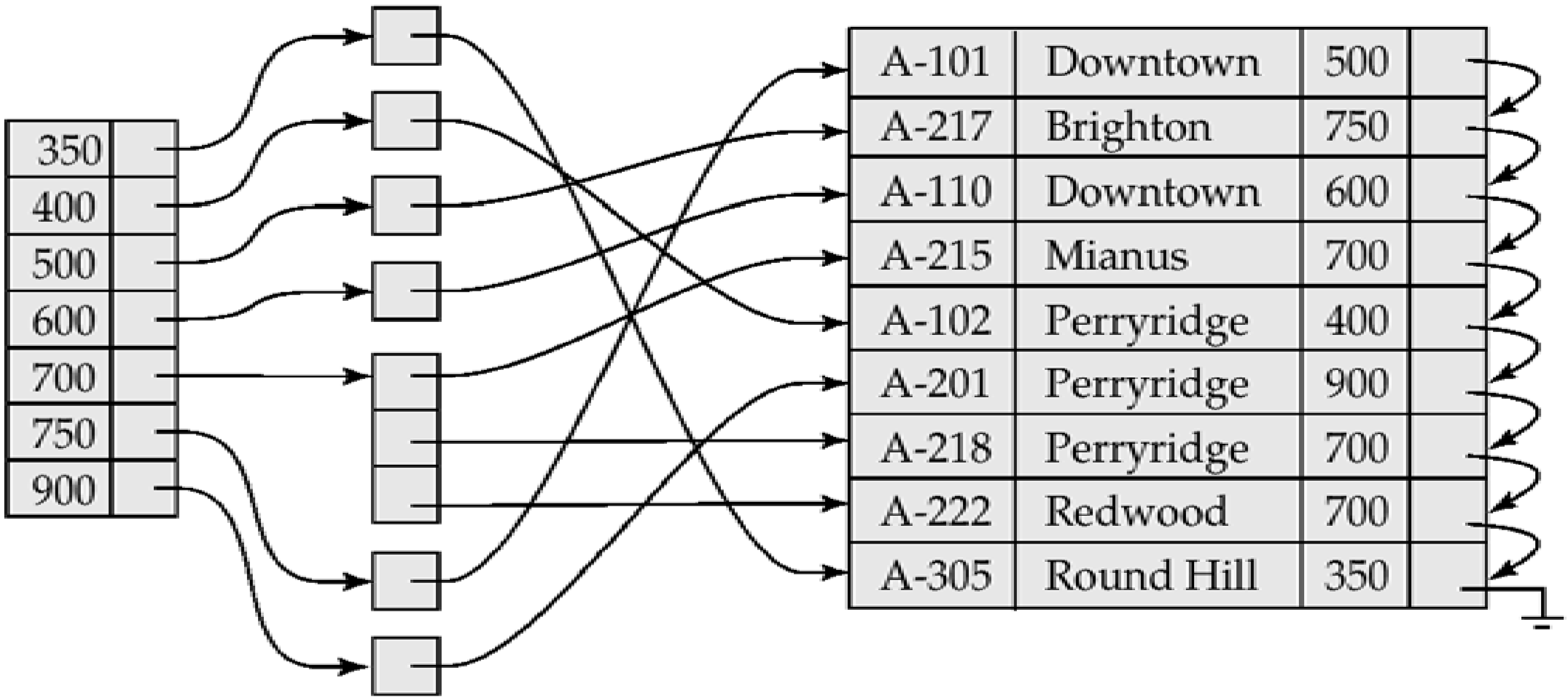
Multilevel Index

- If primary index does not fit in memory, access becomes expensive.
- 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 outer index is still too large to fit in main memory, 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.)



Secondary Index



Secondary index on *balance* **field of** *account*

Primary and Secondary Indices

- Secondary indices have to 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
- Index takes up space

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

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.

B^+-Tree Node Structure

- Typical node

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

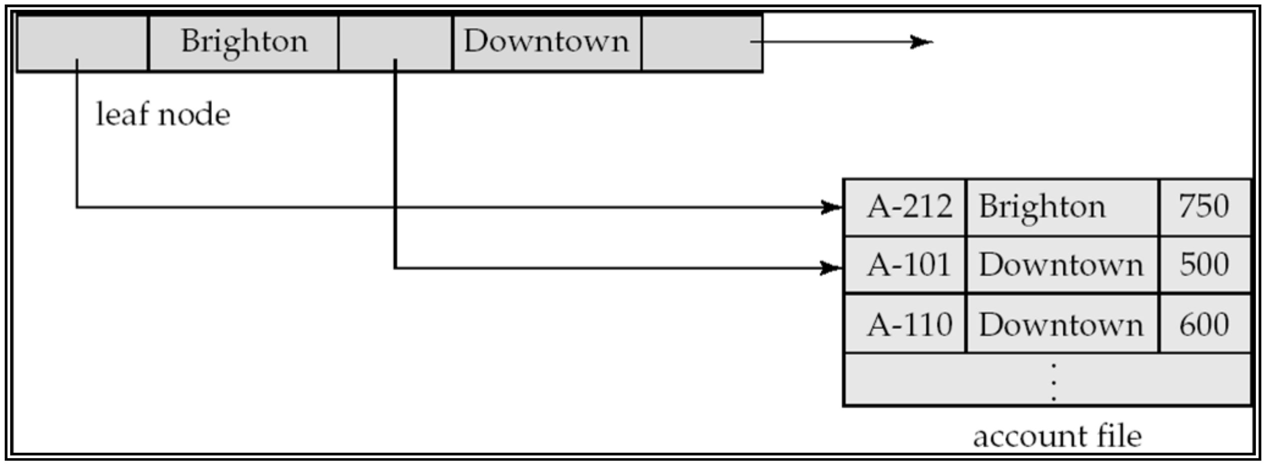
  - K_i are the search-key values
  - P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).
- The search-keys in a node are ordered

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

Leaf Nodes in B^+-Trees

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

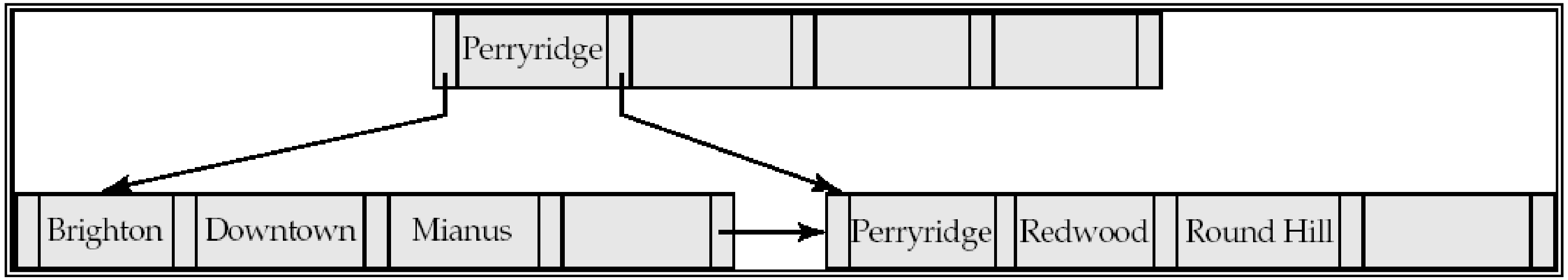


Non-Leaf Nodes in B^+-Trees

- 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	...	P_{n-1}	K_{n-1}	P_n

Example of a 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

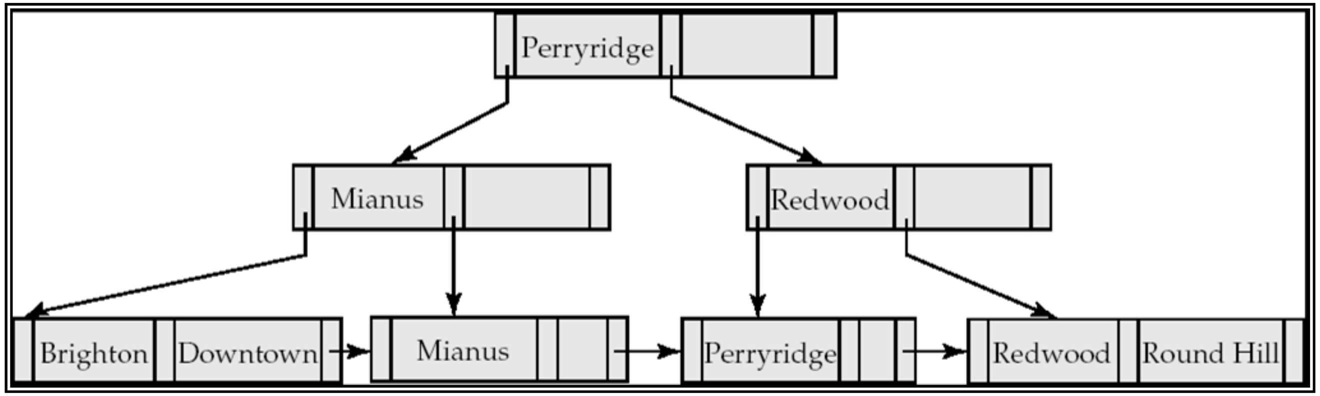
- Since the inter-node connections are done by pointers, "logically" close blocks need not be "physically" close.
- The non-leaf levels of the B^+-tree form a hierarchy of sparse indices.
- The B^+-tree contains a relatively small number of levels
  - Level below root has at least $2 * \lceil n/2 \rceil$ 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.)

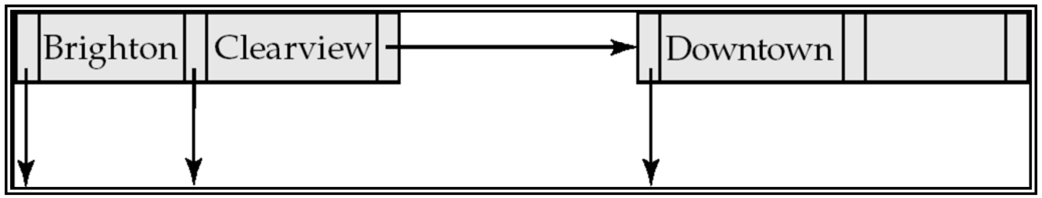
- If there are K search-key values in the file, the height of the tree is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$.
- A node is generally the same size as a disk block, typically 4 kilobytes
  - and n is typically around 100 (40 bytes per index entry).
- With 1 million search key values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

Insertion in B^+-Trees

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.

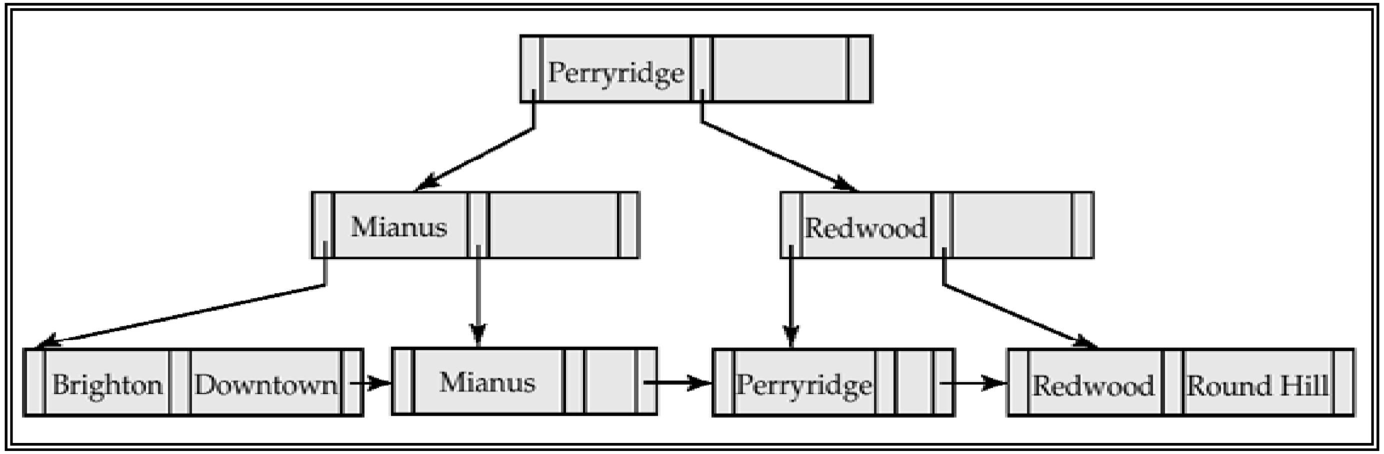
Insertion in B^+-Trees (cont.)

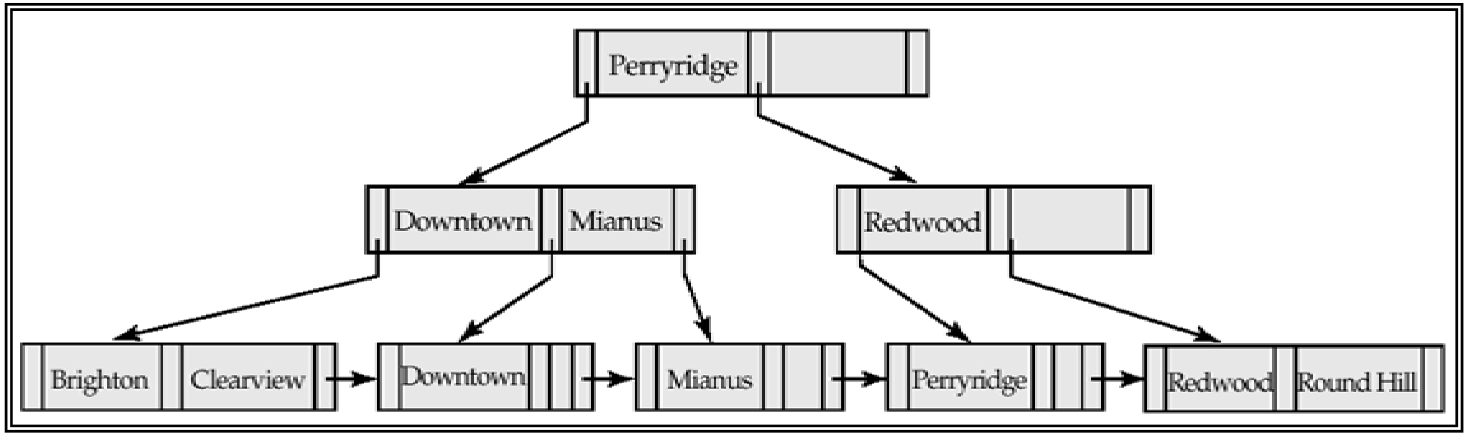
- Splitting a leaf node:
  - take the *n* (search-key value, pointer) pairs (including the one being inserted) in sorted order. Place the first $\lceil n/2 \rceil$ in the original node, and the rest in a new node.
  - let the new node be *p,* and let *k* be the least key value in *p*. Insert (*k*,*p*) 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

Insertion in B^{+}-Trees (cont.)

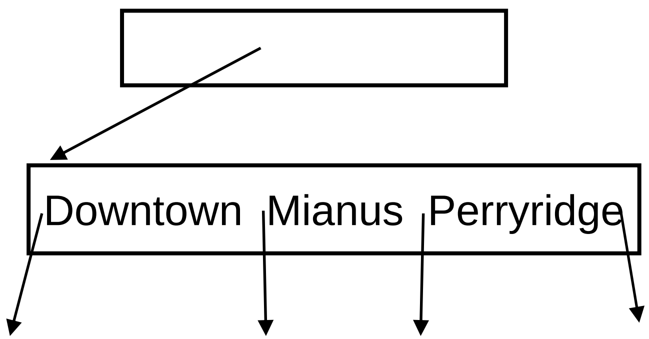


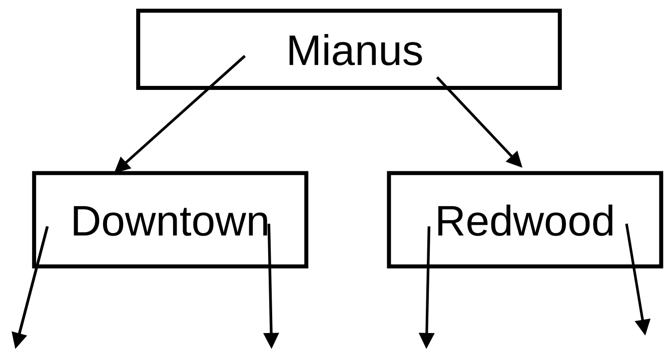


B^{+}-Tree before and after insertion of "Clearview"

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 $P_1, K_1, \ldots, K_{\lceil n/2 \rceil -1}, P_{\lceil n/2 \rceil}$ from M back into node N
  - Copy $P_{\lceil n/2 \rceil +1}, K_{\lceil n/2 \rceil +1}, \ldots, K_n, P_{n+1}$ from M into newly allocated node N'
  - Insert ($K_{\lceil n/2 \rceil}$,N') into parent N
- **Read pseudocode in book!**





Deletion on B^+-Trees

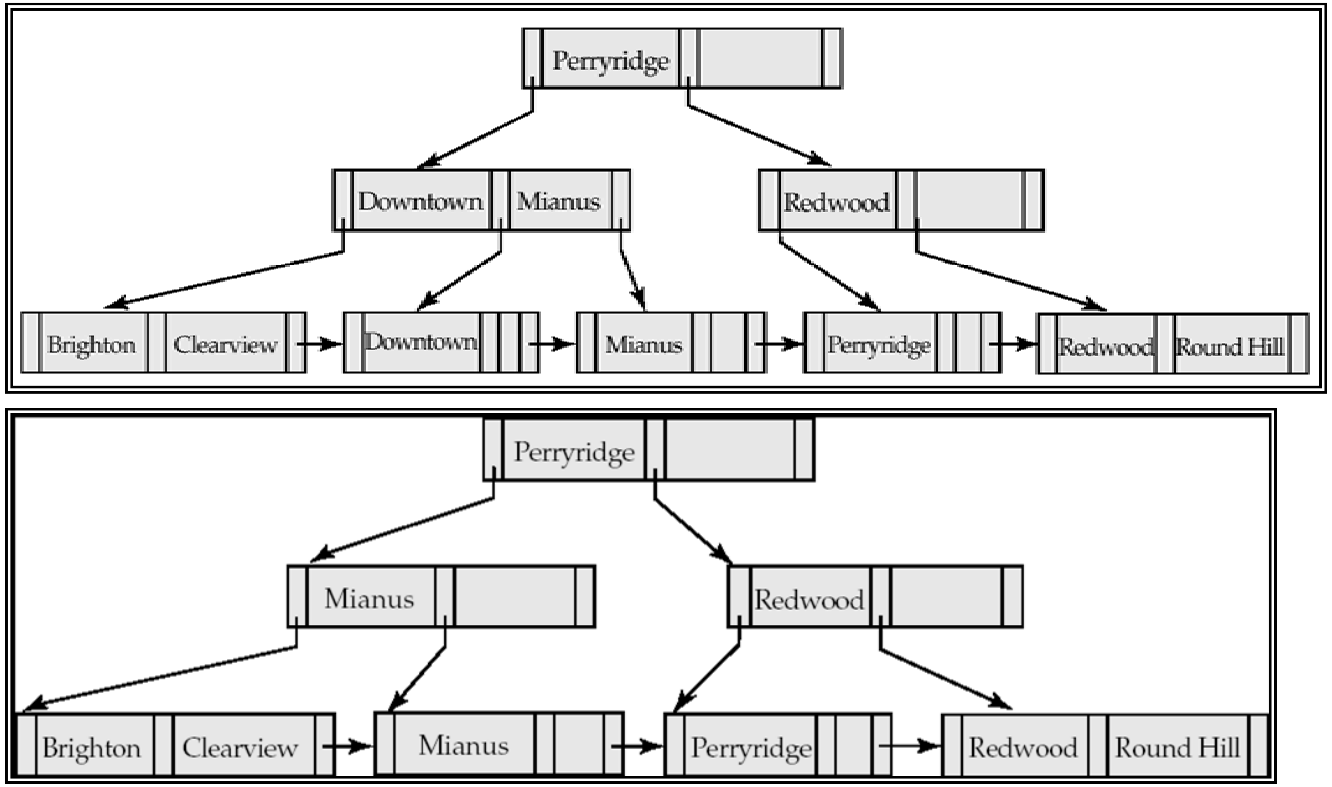
- Find the record to be deleted, and remove it from the main file and from the bucket (if present)
- Remove (search-key value, pointer) from the leaf node if there is no bucket or if the bucket has become empty
- If the node has too few entries due to the removal, and the entries in the node and a sibling 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.

Deletion on B^+-Trees (cont.)

- 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

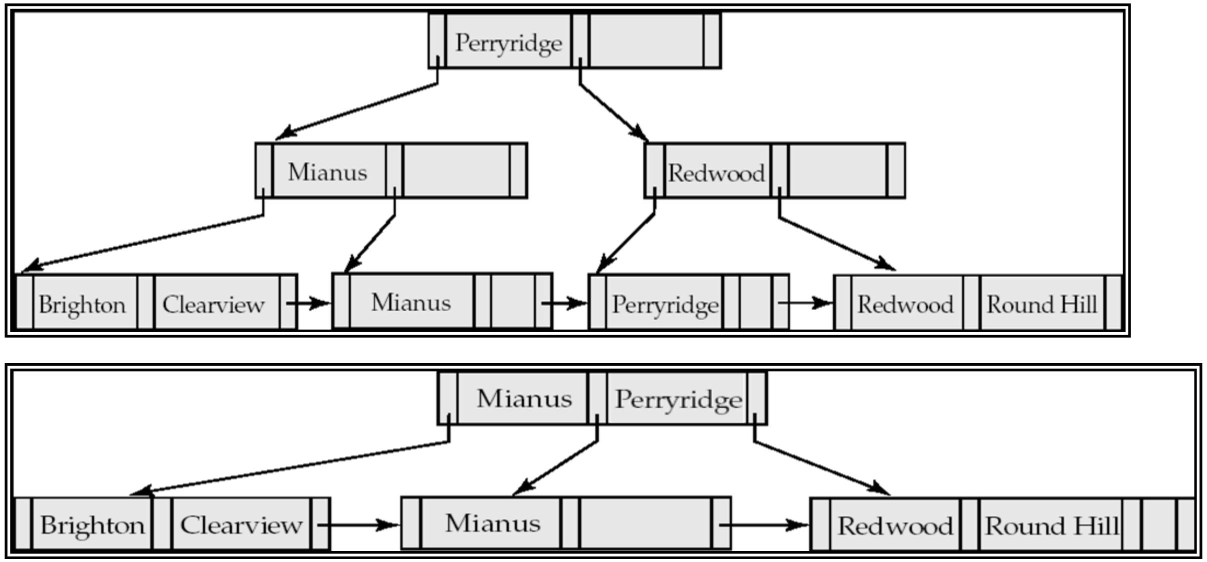
- Deleting "Downtown" causes merging of under-full leaves
  - leaf node can become empty only for n=3!



Before and after deleting "Downtown"

Examples of B^+-Tree Deletion (Cont.)

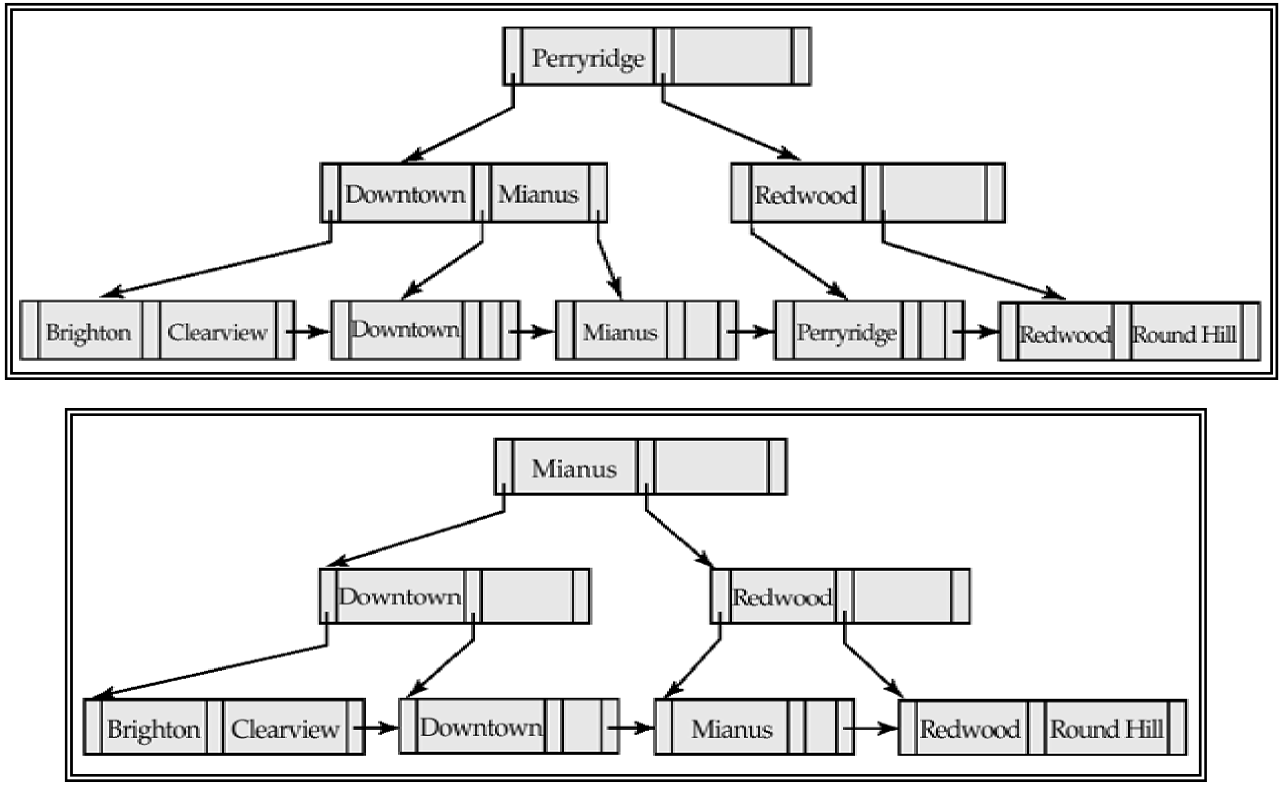
- Leaf with "Perryridge" becomes underfull (actually empty, in this special case) and merged with its sibling.
- As a result "Perryridge" node's parent became underfull, and was merged with its sibling
  - Value separating two nodes (at parent) moves into merged node
  - Entry deleted from parent
- Root node then has only one child, and is deleted



Deletion of "Perryridge" from result of previous example

Examples of B$^+$-Tree Deletion (Cont.)

- 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



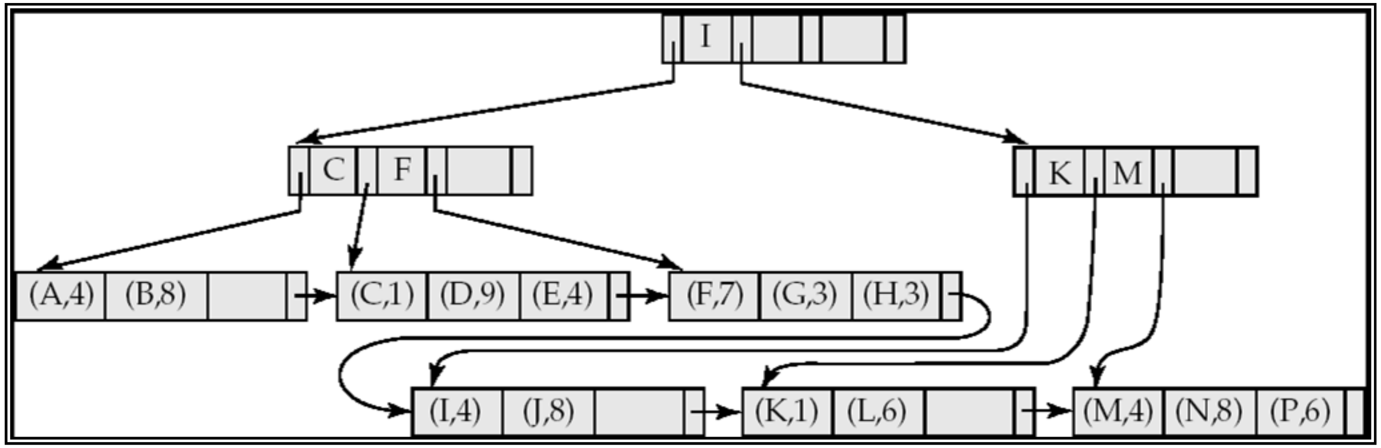
Before and after deletion of "Perryridge" from earlier example

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.)

- 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



Ordered Indexing vs Hashing

- Cost of periodic re-organization
  - Static hashing is worst => dynamic hashing
- Relative frequency of insertions and deletions
- Is it desirable to optimize average access time at the expense of worst-case access time?
- Expected type of queries:
  - Hashing is generally better at retrieving records having a specified value of the key.
  - If range queries are common, ordered indices are to be preferred

Index Definition in SQL

- Create an index

  create index <index-name> **or** <relation-name>
  <attribute-list>)

  E.g.: **create index** *b-index* **on** *branch(branch-name)*

- Use **create unique index** to indirectly specify and enforce the condition that the search key is a candidate key.
  - Not really required if SQL **unique** integrity constraint is supported
- To drop an index

  drop index <index-name>

END OF CHAPTER 12