3. INDEXED FILE ORGANIZATION

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" and by using a "hash function" the values are determined.

In an ordered index, index entries are stored sorted on the search key value.

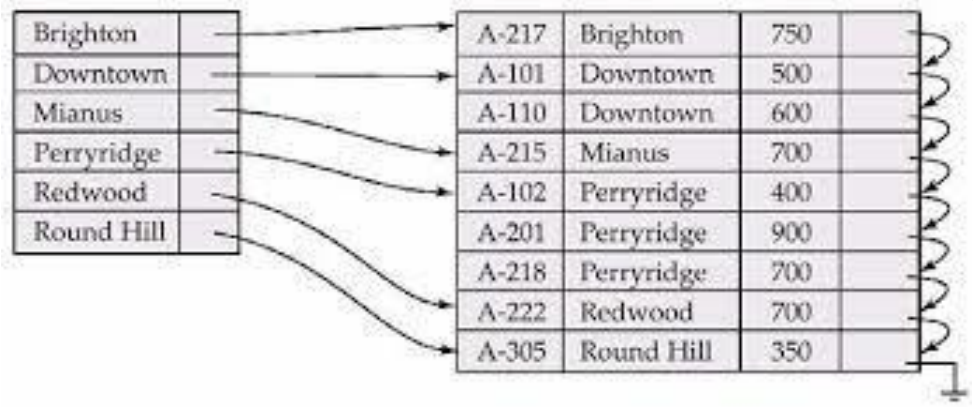
Primary index: in a sequentially ordered file, the index whose search key specifies the sequential order of the file.

Secondary index: an index whose search key specifies an order different from the sequential order of the file.

- Dense index
- Sparse index

(I) Dense Index Files

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



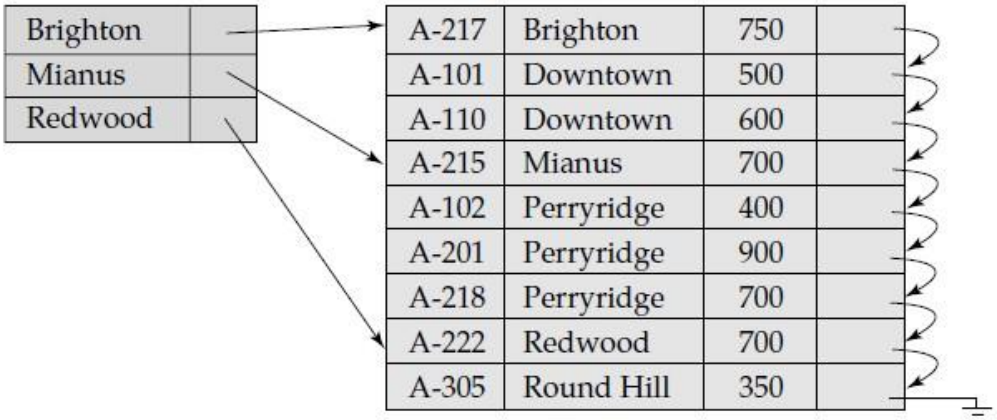
(II) Sparse Index Files

Sparse Index

- Contains index records for only some search-key values.

To locate a record with search-key value K we:

Find index record with largest search-key value that is less than or equal to Search file sequentially starting at the record to which the index record points.



(III) Multilevel Index

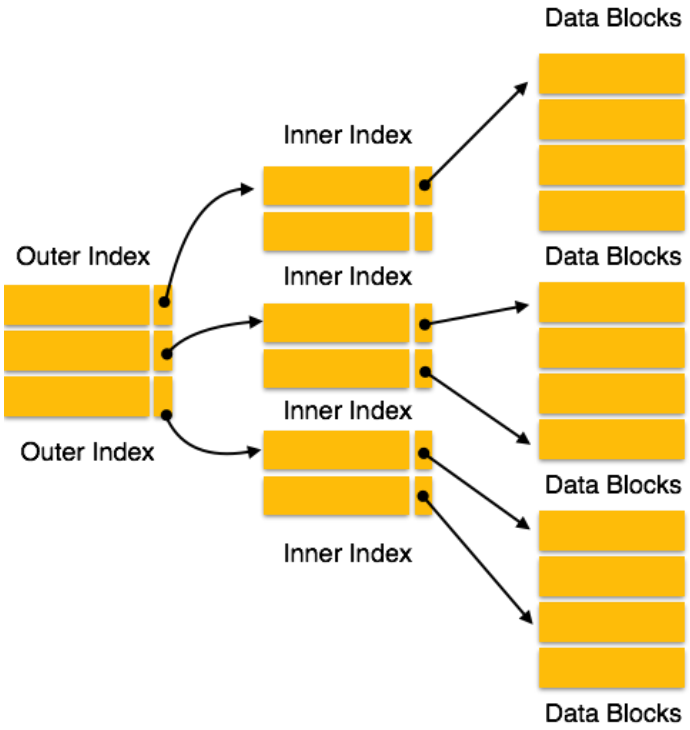
If primary index does not fit in memory, access becomes expensive.

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.



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.

(i) 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.

(ii) Single-level index insertion:

– Perform a lookup using the search-key value appearing in the record to be inserted.

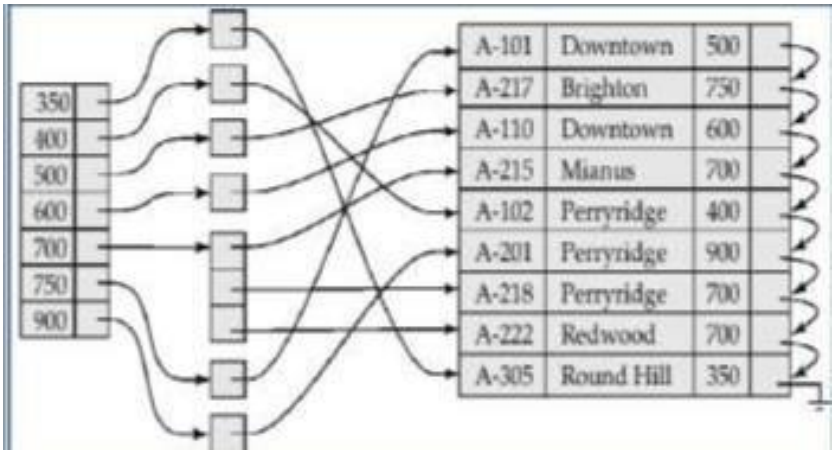
– **Dense indices**

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.

Secondary Index on balance field of account



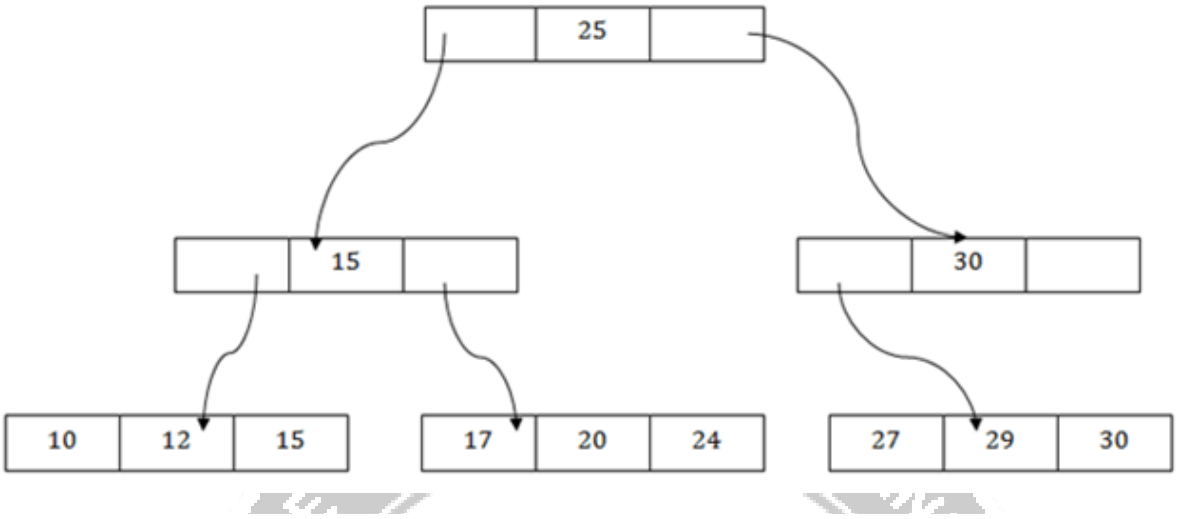
Primary and Secondary Indices

- 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

1. B+-TREE INDEX FILES

B+ File Organization

- B+ tree file organization is the advanced method of an indexed sequential access method. It uses a tree-like structure to store records in File.
- It uses the same concept of key-index where the primary key is used to sort the records. For each primary key, the value of the index is generated and mapped with the record.
- The B+ tree is similar to a binary search tree (BST), but it can have more than two children. In this method, all the records are stored only at the leaf node. Intermediate nodes act as a pointer to the leaf nodes. They do not contain any records.

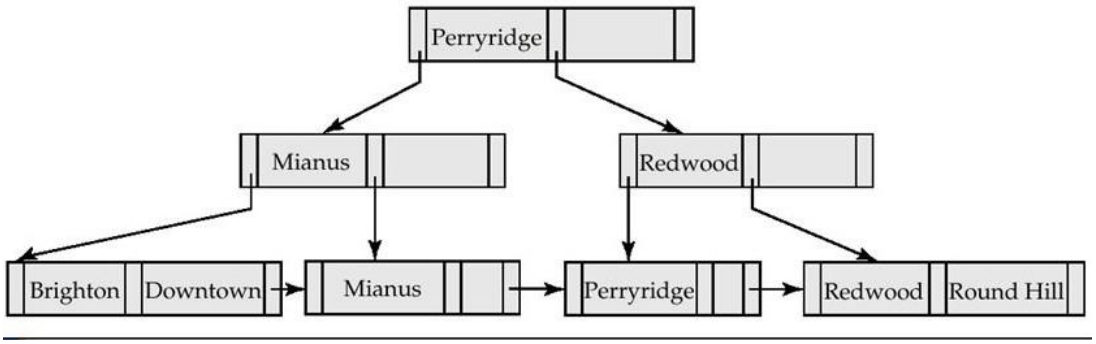


The above B+ tree shows that:

- There is one root node of the tree, i.e., 25.
- There is an intermediary layer with nodes. They do not store the actual record. They have only pointers to the leaf node.
- The nodes to the left of the root node contain the prior value of the root and nodes to the right contain next value of the root, i.e., 15 and 30 respectively.
- There is only one leaf node which has only values, i.e., 10, 12, 17, 20, 24, 27 and 29.
- Searching for any record is easier as all the leaf nodes are balanced.
- In this method, searching any record can be traversed through the single path and accessed easily

Example for B+-TREE INDEX FILES

Example of a B+-tree : B+-tree for account file (n = 3)



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.

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 [n/2] and n children.
- Special cases:
  - If the root is not a leaf, it has at least 2 children.
  - If the root is a leaf, it can have between 0 and (n–1) values.

B+-Tree Node Structure

Typical node

– Ki are the search-key values

– Pi are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).

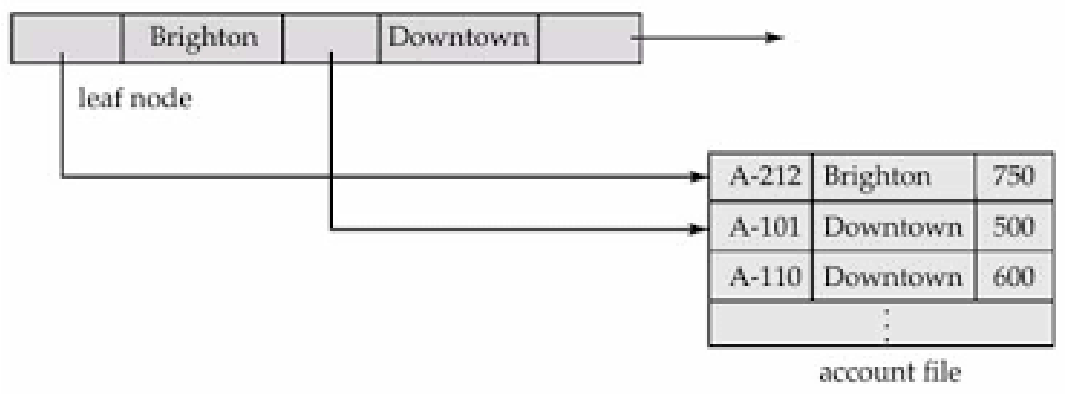
- The search-keys in a node are ordered

K1 < K2 < K3 < . . . < Kn–1

Properties of Leaf Nodes

For i = 1, 2, . . ., n–1, pointer Pi either points to a file record with search-key value Ki, or to a bucket of pointers to file records, each record having search-key value Ki.

Pn points to next leaf node in search-key order

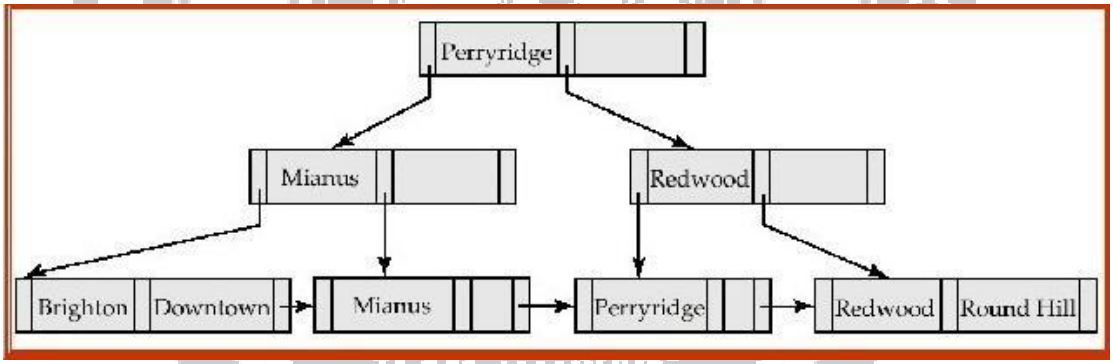


Non-Leaf Nodes in B+-Trees

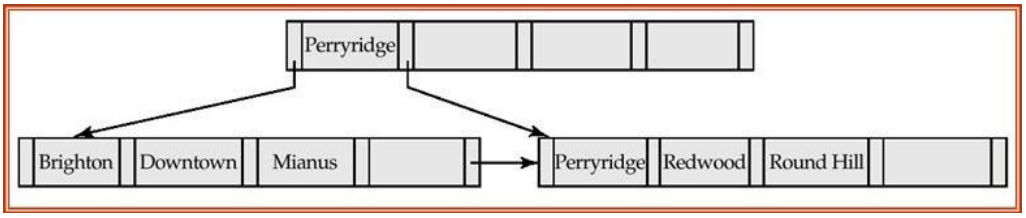
Non leaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with m pointers:

– All the search-keys in the subtree to which P1 points are less than K1.

Example of a B+-tree: B+-tree for account file (n = 3)



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



Non-leaf nodes other than root must have between 3 and 5 children ((n/2⌉ 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.

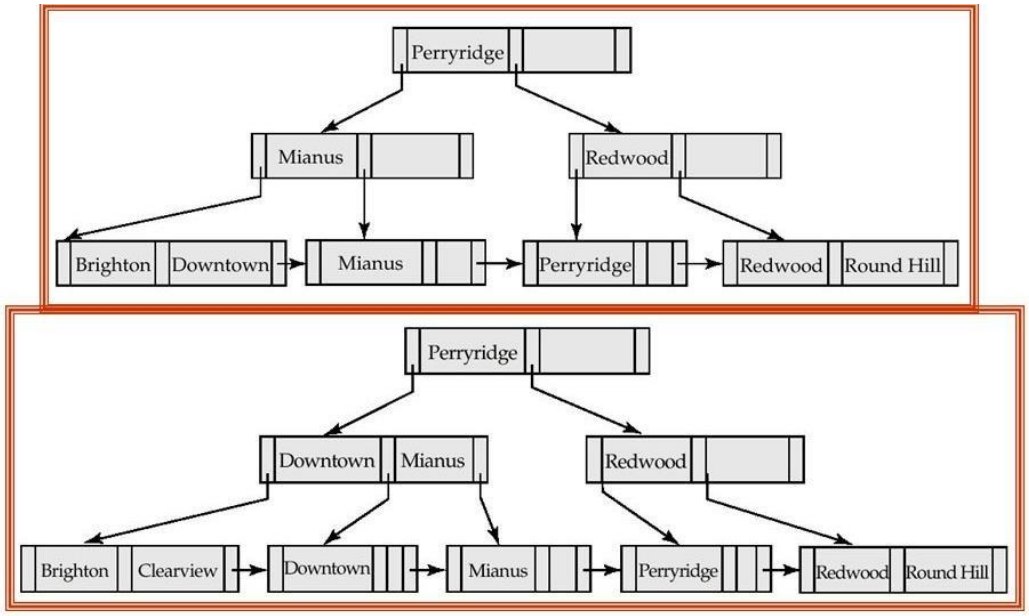
- The B+-tree contains a relatively small number of levels thus searches can be conducted efficiently.
- Insertions and deletions to the main file can be handled efficiently.

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.

Example: B+-Tree before and after insertion of "Clearview"

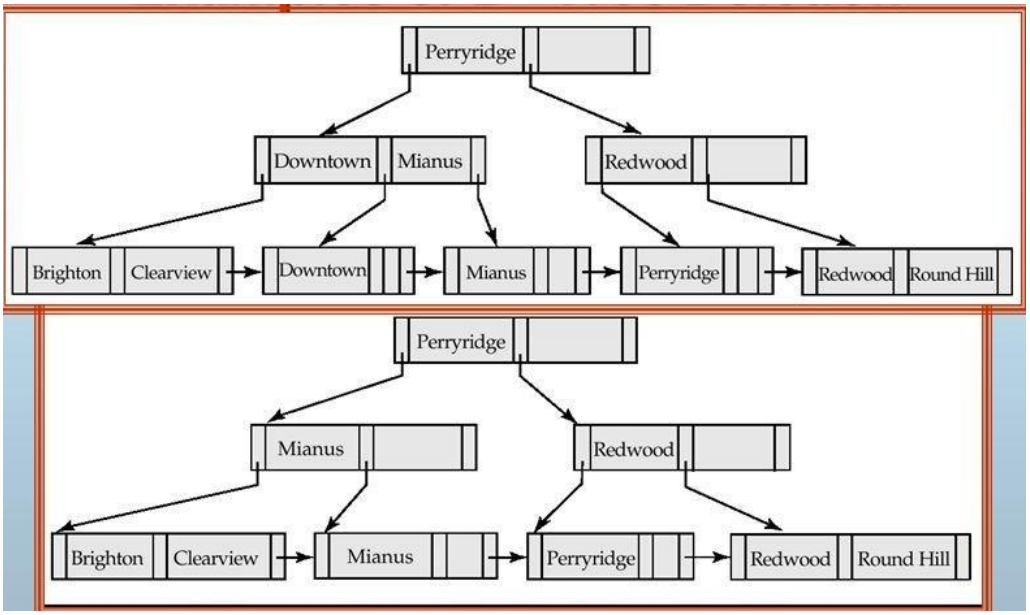


B+-Tree before and after insertion of "Clearview"

Updates on B+-Trees: Deletion

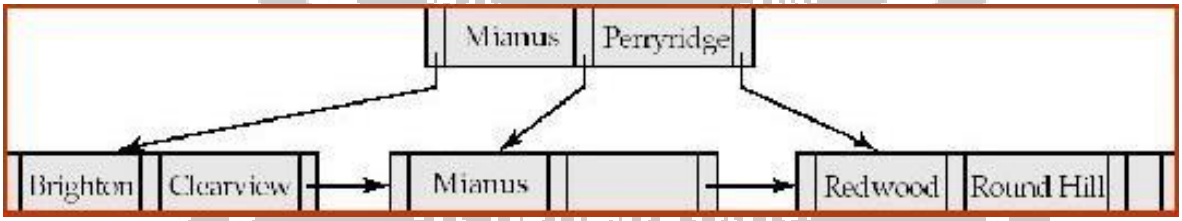
- Find the record to be deleted, and remove it from the main file and from the bucket (if present)
- Remove (search-key value, pointer) from the leaf node if there is no bucket or if the bucket has become empty
- If the node has too few entries due to the removal, and the entries in the node and a sibling 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 (Ki–1, Pi), where Pi is the pointer to the deleted node, from its parent, recursively using the above procedure.



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

Deletion of "Perryridge" from result of previous example



- Node with "Perryridge" becomes empty and merged with its sibling.
- Root node then had only one child, and was deleted and its child became the new root node

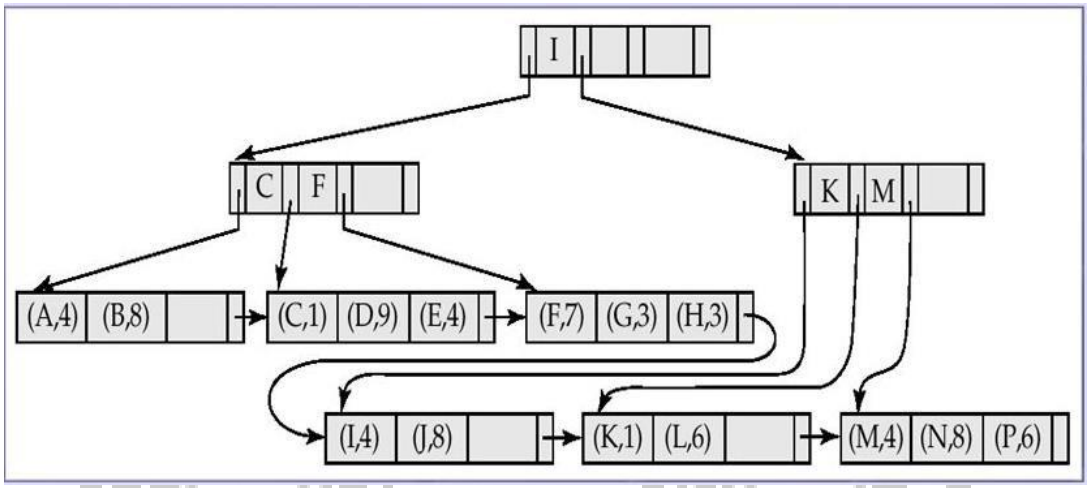
B+-Tree File Organization

• 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.

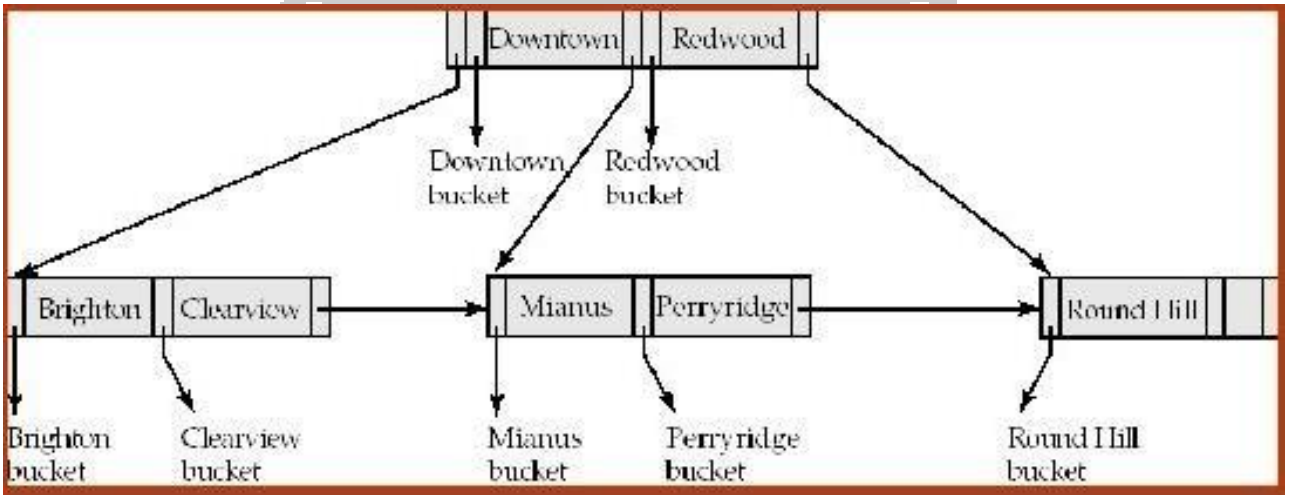
• Insertion and deletion are handled in the same way as insertion and deletion of entries in a B+-tree index.



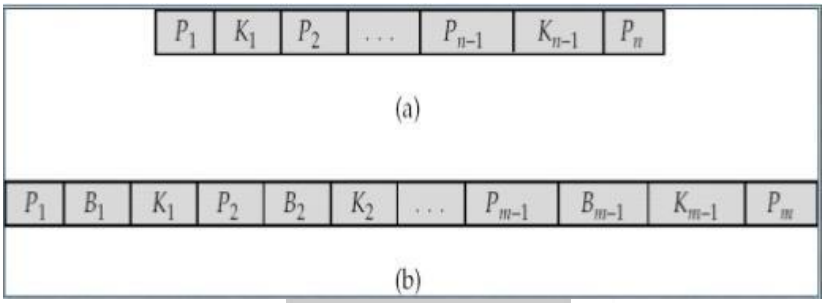
2. B -TREE INDEX FILES

• Similar to B+-tree, but B-tree allows search-key values to appear only once; eliminates redundant storage of search keys.

• Search keys in nonleaf nodes appear nowhere else in the B-tree; an additional pointer field for each search key in a nonleaf node must be included.



Generalized B-tree leaf node



Nonleaf node – pointers Bi are the bucket or file record pointers.

Advantages of B-Tree indices: –

- May use less tree nodes than a corresponding B+-Tree.
- Sometimes possible to find search-key value before reaching leaf node.

Disadvantages of B-Tree indices:

- Only small fraction of all search-key values are found early
- Non-leaf nodes are larger, so fan-out is reduced (no. of pointers). Thus, B-Trees typically have greater depth than corresponding B+-Tree
- Insertion and deletion more complicated than in B+-Trees
- Implementation is harder than B+-Trees.