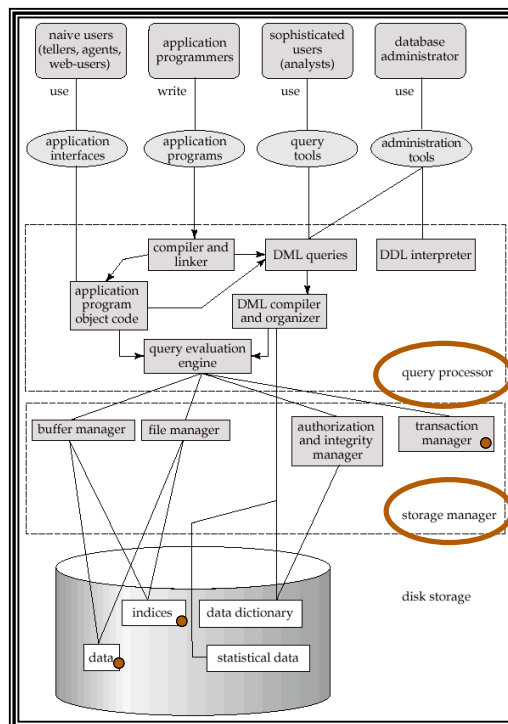


DBMS INDEXING

October 17th, 2007

Overall DBMS Structure



Basic Concepts

- Indexing mechanisms are used to speed up access to data.
A Search Key is an attribute to set of attributes used to look up records in a file.
- An index file consists of records usually referred to as *index entries*:

search-key	pointer
------------	---------

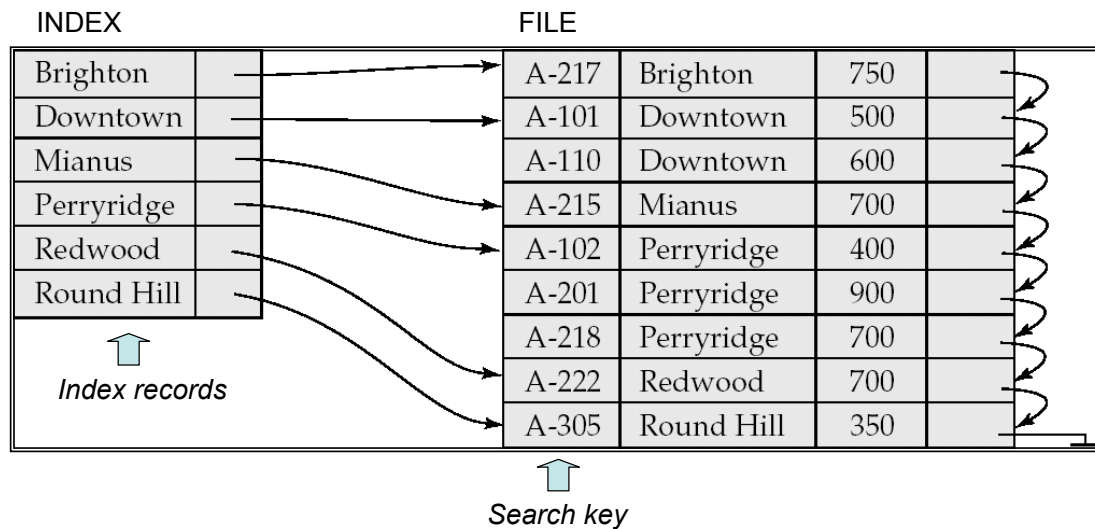
- Types of indices used:
 - *Ordered indices*: search keys are stored in sorted order
 - *Tree-indices*: search keys are organized in tree-structures to speed-up search
 - *Hash indices*: search keys are distributed uniformly across *buckets* using a *hash function*.

Ordered Indices

- In an ordered index, index entries are sorted on the search key value.
E.g., owner catalog in public car register.
 - *Primary index*: in a sequentially ordered file, the index whose search key specifies the sequential order of the file. The search key of a primary index is usually the primary key.
 - *Secondary index*: an index whose search key specifies an order different from the sequential order of the file.

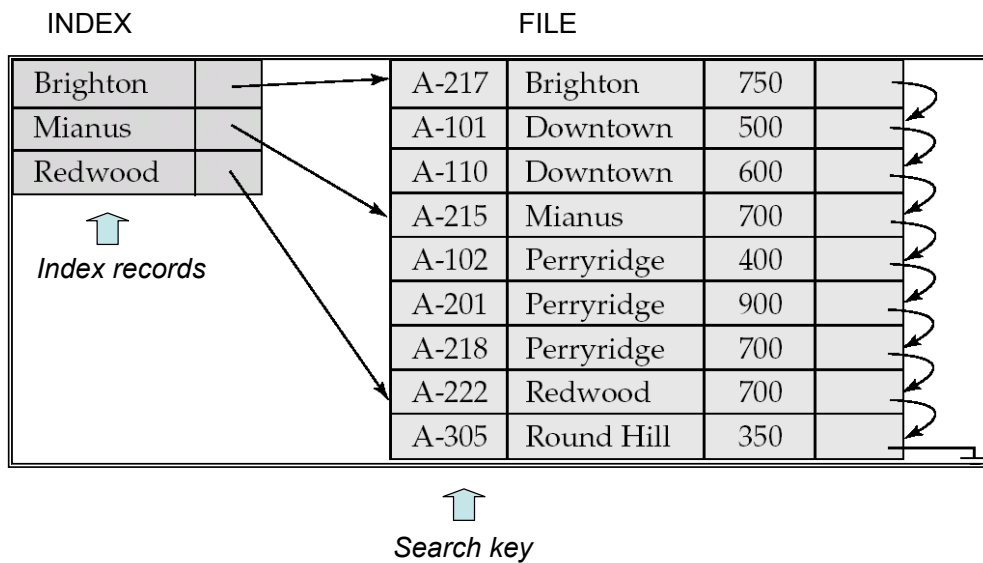
Dense Index Files

- With dense indexes, one index record appears for every search-key value in the file.



Sparse Index Files

- A sparse index contains index records for only some search-key values. This solution is only 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 $< K$
 - Perform a sequential search on the file starting at the record to which the index record points
- Advantages are less space and less maintenance overhead for insertions and deletions. Disadvantages are concerned with slower access than dense index.



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 to maintain performance.
 - Advantage of B⁺-tree index files: automatically reorganizes itself with small, local, changes, when insertions and deletions are made. Reorganization of entire file is not required to maintain performance.
 - Disadvantage of B⁺-trees: extra insertion and deletion overhead, space overhead.
- The advantages of B⁺-trees outweigh disadvantages.

- A B⁺-tree of order n is a tree satisfying the following properties:
 - All paths from root to leaf have 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
 - The root has at least 2 children.
- Special case:
 - If the root is a leaf, it has between 0 and $(n-1)$ values.

B⁺-Tree Node Structure

- Typical node

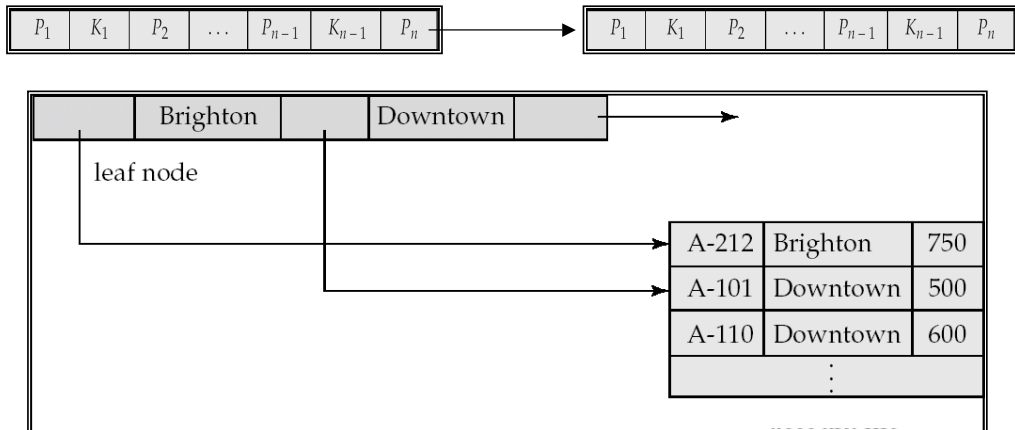


- K_i are the search-key values
 - P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).
- The search-keys in a node are ordered

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

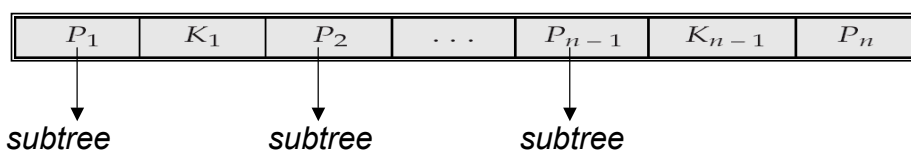
Leaf Nodes in B⁺-Trees

- Properties of a leaf node:
 - For $i = 1, 2, \dots, n-1$, 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

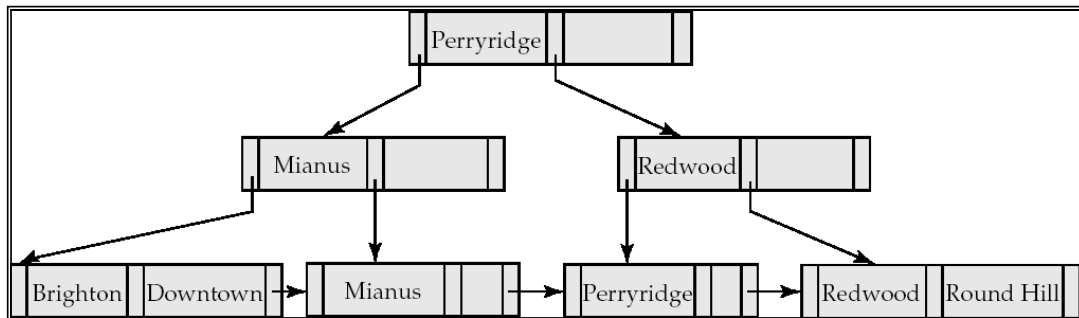


Non-Leaf Nodes in B⁺-Trees

- 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



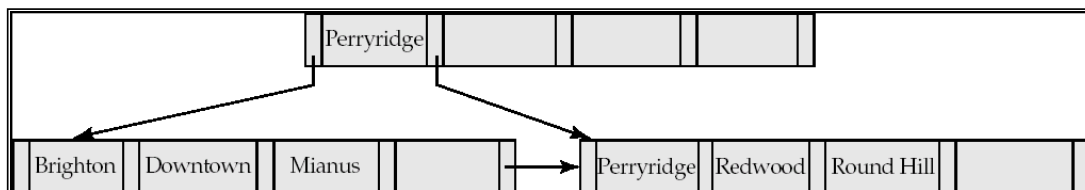
Example of a B⁺-tree



B⁺-tree for $n = 3$

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

Example of B⁺-tree



B⁺-tree for $n = 5$ (same file)

- 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 .
 - Start with the root node
 - Examine the node for the smallest search-key value $> k$.
 - If such a value exists, assume it is K_j . Then follow P_j to the child node
 - Otherwise $k \geq K_{m-1}$, where there are m pointers in the node. Then follow P_m to the child node.
 - If the node reached by following the pointer above is not a leaf node, repeat step 1 on the node
 - Else we have reached 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.

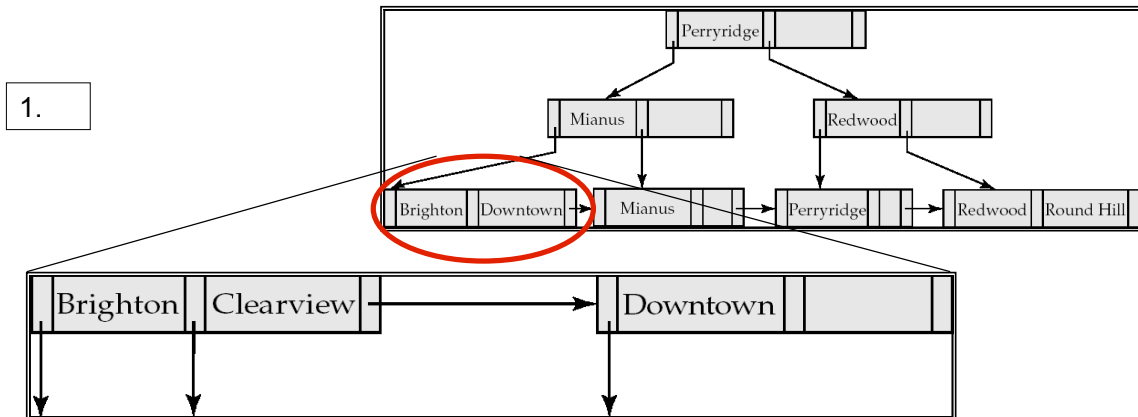
- In processing a query, a path is traversed in the tree from the root to some leaf node.
- If there are K search-key values in the file, the path is no longer than $\lceil \log_{\lfloor n/2 \rfloor}(K) \rceil$
- A node is generally the same size as a disk block, typically 4 kilobytes, and n is typically 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. (a balanced binary tree with 1 million search key values requires around 20 nodes accesses in a lookup; every node access may need a disk I/O....)

Updates on B⁺-Trees: Insertion

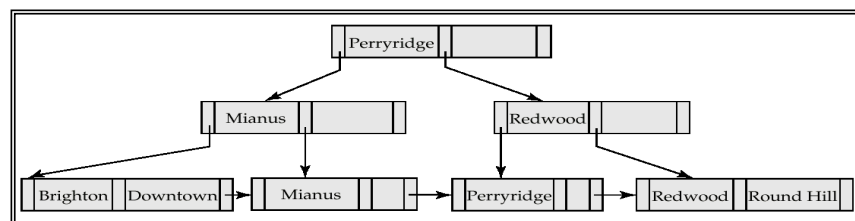
1. Find the leaf node in which the search-key value would appear
2. If the search-key value is already there in the leaf node, record is added to file and if necessary a pointer is inserted into the bucket.
Else, add the record to the main file and create a bucket if necessary. Then:
 - If there is room in the leaf node, insert a (key-value, pointer) pair in the leaf node
 - Otherwise, *split* the node along with the new (key-value, pointer) entry

Splitting of nodes proceeds upwards till splitting a node is not required:

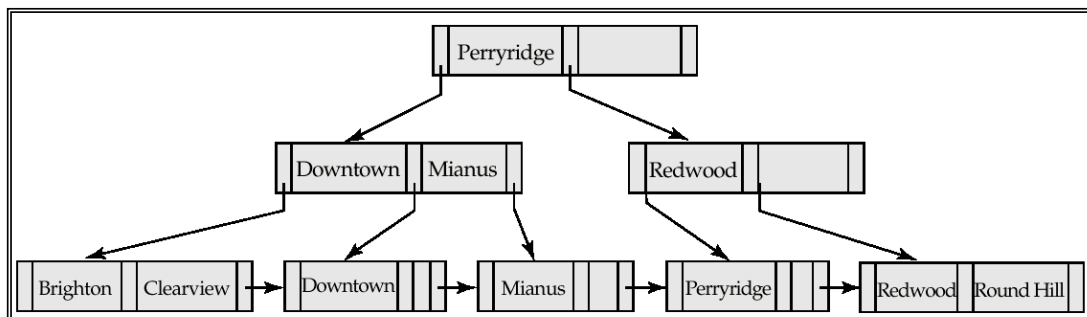
1. take the n (search-key value, pointer) pairs (including the one being inserted) in sorted order. Place the first $\lceil n/2 \rceil$ in the original node, and the rest in a new node.
2. let the new node be p , and let k be the least key value in p . Insert (k,p) in the parent of the node being split. If the parent is full, split it and propagate the split further up.
3. In the worst case the root node may be split increasing the height of the tree by 1.



Result of splitting the node containing Brighton and Downtown on inserting Clearview



2.



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

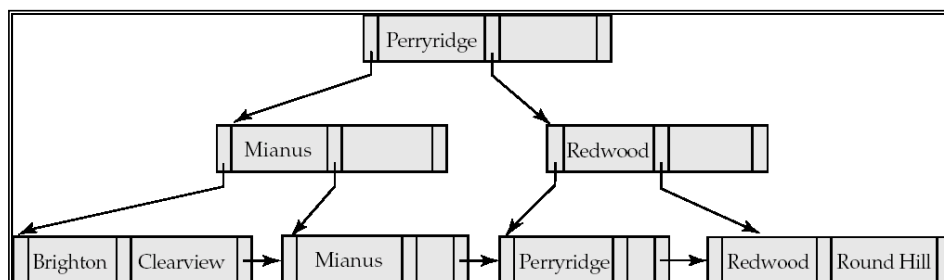
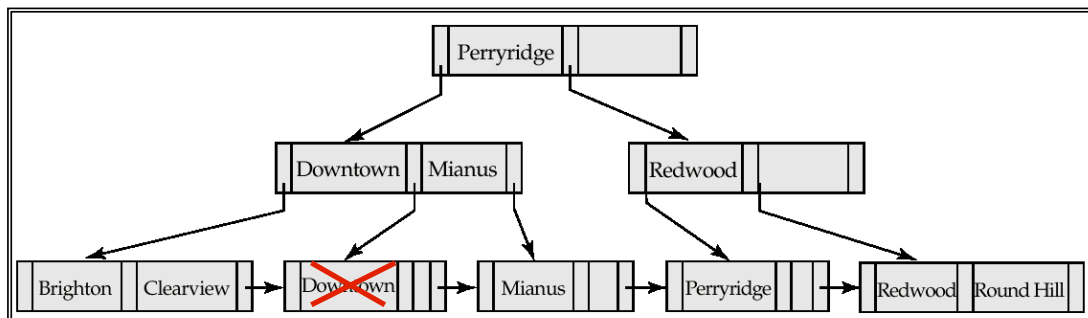
Updates on B⁺-Trees: Deletion

1. Find the record to be deleted, and remove it from the main file and from the bucket (if present)
2. Remove (search-key value, pointer) from the leaf node if there is no bucket or if the bucket has become empty
3. 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.

Otherwise, if the node has too few entries due to the removal, and the entries in the node and a sibling doesn't fit into a single node, then

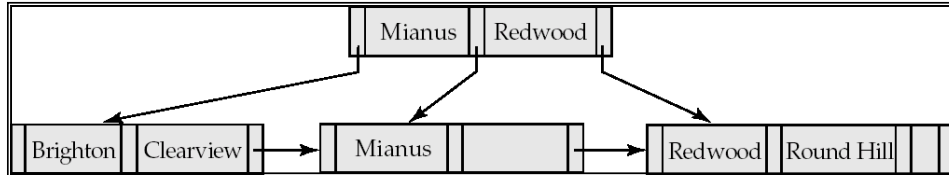
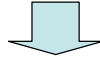
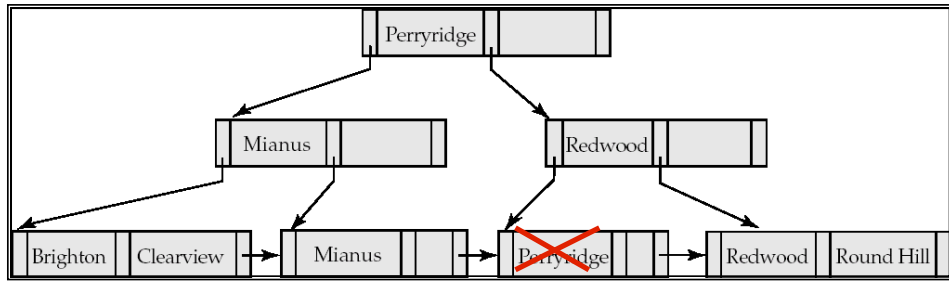
- Redistribute the pointers between the node and a sibling such that both have more than the minimum number of entries.
- Update the corresponding search-key value in the parent of the node.

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.



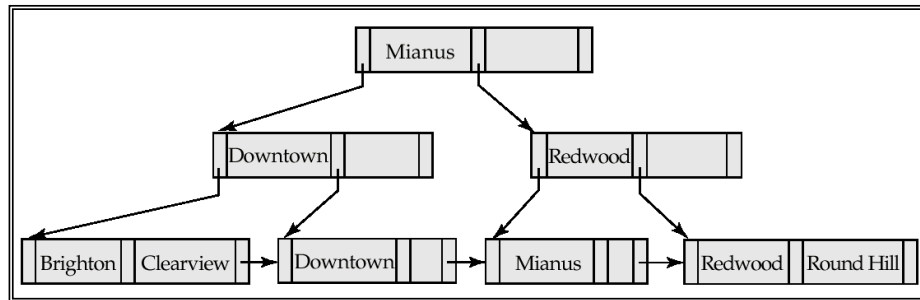
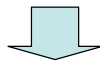
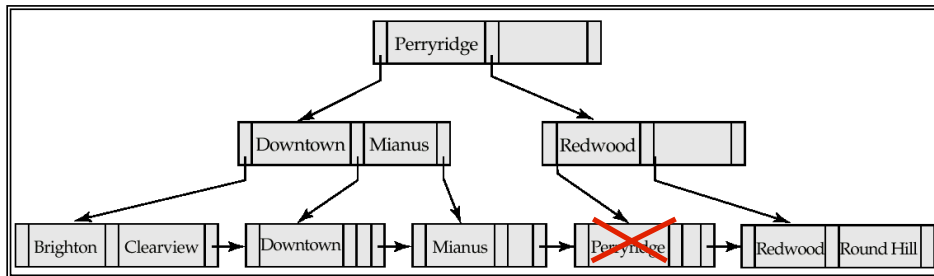
Before and after deleting "Downtown"

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



Deletion of "Perryridge" from result of previous example

- Node with "Perryridge" becomes empty and is merged with its sibling.
- As a result "Perryridge" node's parent **becomes underfull**, and **is merged** with its sibling (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

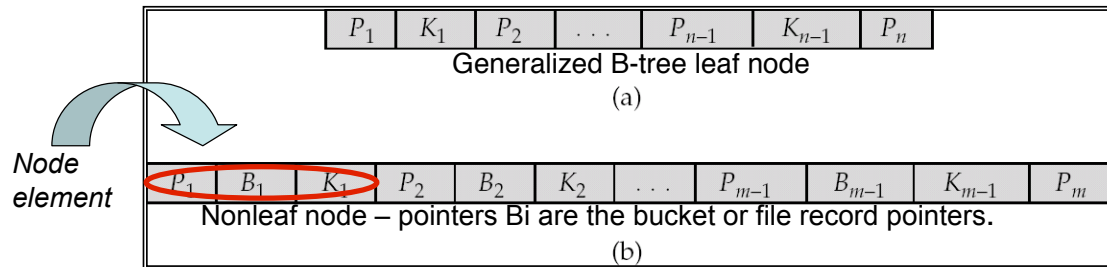


Before and after deletion of "Perryridge" from earlier example

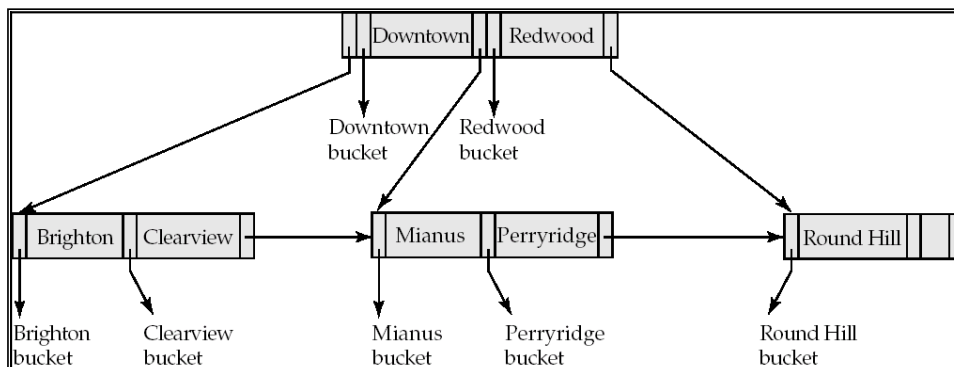
- Parent of leaf containing Perryridge became underfull, and borrowed a pointer from its left sibling
- Search-key value in the parent's parent changes as a result

B-Tree Index Files

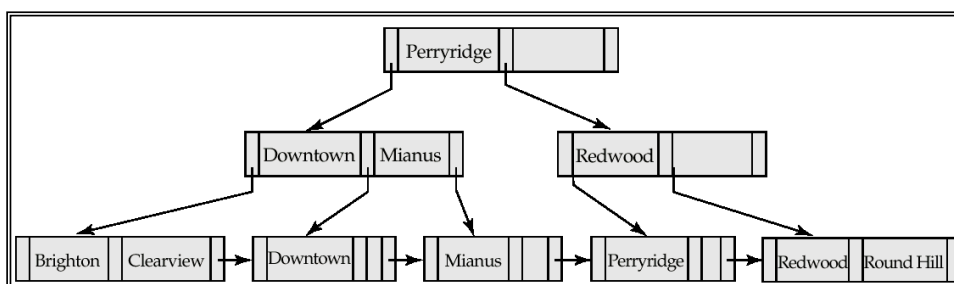
- Similar to B+-tree, but B-tree allows search-key values to appear only once; eliminates redundant storage of search keys.
- Search keys in nonleaf nodes appear nowhere else in the B-tree; an additional pointer field for each search key in a nonleaf node must be included.



B-Tree Index File Example



B-tree (above) and B+-tree (below) on same data



- 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. 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.

- Typically, advantages of B-Trees do not outweigh disadvantages.

Static Hashing

- A bucket is a unit of storage containing one or more records (a bucket is typically a disk block).
- In a hash file organization we obtain the bucket of a record directly from its search-key value using a hash function.

- Hash function h is a function from the set of all search-key values K to the set of all bucket addresses B .
- Hash function is used to locate records for access, insertion as well as deletion.

- Records with different search-key values may be mapped to the same bucket; thus the entire bucket has to be searched sequentially to locate a record.

Example of Hash File Organization

Hash file organization of the *account* file, using *branch_name* as key

<p>bucket 0</p> <table border="1" style="width: 100%; height: 20px;"> <tr><td style="width: 33%;"></td><td style="width: 33%;"></td><td style="width: 33%;"></td></tr> </table> <p>bucket 1</p> <table border="1" style="width: 100%; height: 20px;"> <tr><td style="width: 33%;"></td><td style="width: 33%;"></td><td style="width: 33%;"></td></tr> </table> <p>bucket 2</p> <table border="1" style="width: 100%; height: 20px;"> <tr><td style="width: 33%;"></td><td style="width: 33%;"></td><td style="width: 33%;"></td></tr> </table> <p>bucket 3</p> <table border="1" style="width: 100%;"> <tr><td style="width: 33%;">A-217</td><td style="width: 33%;">Brighton</td><td style="width: 33%;">750</td></tr> <tr><td>A-305</td><td>Round Hill</td><td>350</td></tr> <tr><td style="height: 20px;"></td><td></td><td></td></tr> </table> <p>bucket 4</p> <table border="1" style="width: 100%;"> <tr><td style="width: 33%;">A-222</td><td style="width: 33%;">Redwood</td><td style="width: 33%;">700</td></tr> <tr><td style="height: 20px;"></td><td></td><td></td></tr> </table>										A-217	Brighton	750	A-305	Round Hill	350				A-222	Redwood	700				<p>bucket 5</p> <table border="1" style="width: 100%;"> <tr><td style="width: 33%;">A-102</td><td style="width: 33%;">Perryridge</td><td style="width: 33%;">400</td></tr> <tr><td>A-201</td><td>Perryridge</td><td>900</td></tr> <tr><td>A-218</td><td>Perryridge</td><td>700</td></tr> <tr><td style="height: 20px;"></td><td></td><td></td></tr> </table> <p>bucket 6</p> <table border="1" style="width: 100%; height: 20px;"> <tr><td style="width: 33%;"></td><td style="width: 33%;"></td><td style="width: 33%;"></td></tr> </table> <p>bucket 7</p> <table border="1" style="width: 100%;"> <tr><td style="width: 33%;">A-215</td><td style="width: 33%;">Mianus</td><td style="width: 33%;">700</td></tr> <tr><td style="height: 20px;"></td><td></td><td></td></tr> </table> <p>bucket 8</p> <table border="1" style="width: 100%;"> <tr><td style="width: 33%;">A-101</td><td style="width: 33%;">Downtown</td><td style="width: 33%;">500</td></tr> <tr><td>A-110</td><td>Downtown</td><td>600</td></tr> <tr><td style="height: 20px;"></td><td></td><td></td></tr> </table> <p>bucket 9</p> <table border="1" style="width: 100%; height: 20px;"> <tr><td style="width: 33%;"></td><td style="width: 33%;"></td><td style="width: 33%;"></td></tr> </table>	A-102	Perryridge	400	A-201	Perryridge	900	A-218	Perryridge	700							A-215	Mianus	700				A-101	Downtown	500	A-110	Downtown	600						
A-217	Brighton	750																																																								
A-305	Round Hill	350																																																								
A-222	Redwood	700																																																								
A-102	Perryridge	400																																																								
A-201	Perryridge	900																																																								
A-218	Perryridge	700																																																								
A-215	Mianus	700																																																								
A-101	Downtown	500																																																								
A-110	Downtown	600																																																								

- Hash file organization of *account* file, using *branch_name* as key
- There are 10 buckets,
- The binary representation of the *i*th character is assumed to be the integer *i*.
- The hash function returns the sum of the binary representations of the characters modulo 10

E.g.

$$h(\text{Perryridge}) = 5$$

$$h(\text{Round Hill}) = 3 \quad h(\text{Brighton}) = 3$$

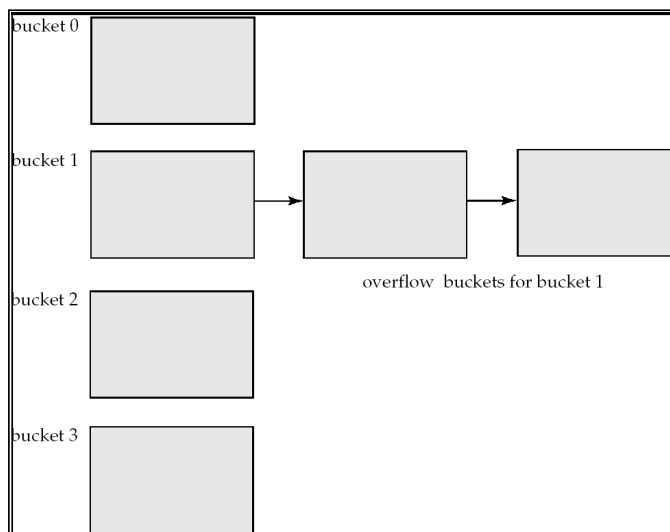
Hash Functions

- Worst 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.
- An ideal hash function is *uniform*, i.e., each bucket is assigned the same number of search-key values from the set of *all* possible values.
- 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. .

Handling of Bucket Overflows

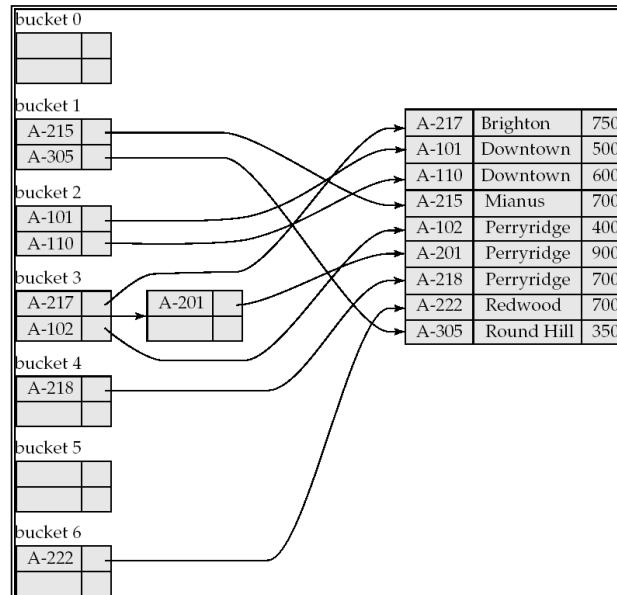
- Bucket overflow can occur because of
 - Insufficient buckets
 - Skew in distribution of records. This can occur due to two reasons:
 - multiple records have same search-key value
 - chosen hash function produces non-uniform distribution of key values
- Although the probability of bucket overflow can be reduced, it cannot be eliminated; it is handled by using *overflow buckets*.

- Overflow chaining – the overflow buckets of a given bucket are chained together in a linked list. This scheme is called *closed hashing*



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.



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.

Dynamic Hashing

- Good for database that grows and shrinks in size. Allows the hash function to be modified dynamically
- *Extendable hashing* – one form of dynamic hashing
 - Hash function generates values over a large range — typically b -bit integers, with $b = 32$.
 - At any time use only a prefix of the hash function to index into a table of bucket addresses.
 - Let the length of the prefix be i bits, $0 \leq i \leq 32$.
 - Bucket address table size = 2^i . Initially $i = 0$
 - Value of i grows and shrinks as the size of the database grows and shrinks.
 - Multiple entries in the bucket address table may point to a bucket.
 - Thus, actual number of buckets is $< 2^i$

The number of buckets also changes dynamically due to coalescing and splitting of buckets.