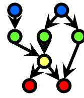


Lecture 26 of 42

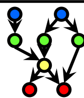
Indexing and Hashing
Discussion: B+ Trees

Wednesday, 25 October 2006

William H. Hsu
Department of Computing and Information Sciences, KSU

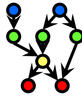
KSOL course page: http://snipurl.com/va60
Course web site: http://www.kddresearch.org/Courses/Fall-2006/CIS560
Instructor home page: http://www.cis.ksu.edu/~bhsu

Reading for Next Class:
Second half of Chapter 12, Silberschatz *et al.*, 5th edition



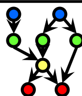
Chapter 12: Indexing and Hashing

- 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



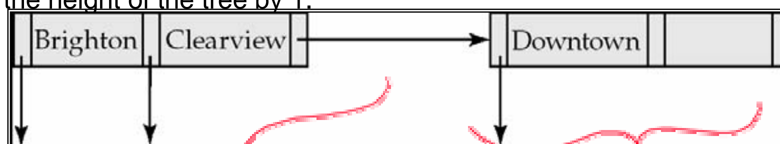
Updates on B+-Trees: Insertion

- 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.

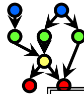


Updates on B+-Trees: Insertion (Cont.)

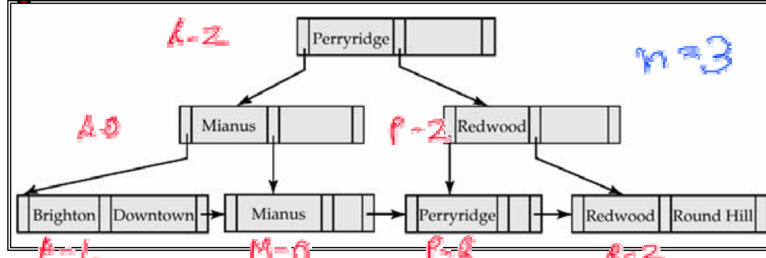
- Splitting a node:
  - take the *n*(search-key value, pointer) pairs (including the one being inserted) in sorted order. Place the first ⌈ *n*/2 ⌉ in the original node, and the rest in a new node.
  - let the new node be *p,* and let *k* be the least key value in *p.* Insert (*k,p*) in the parent of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

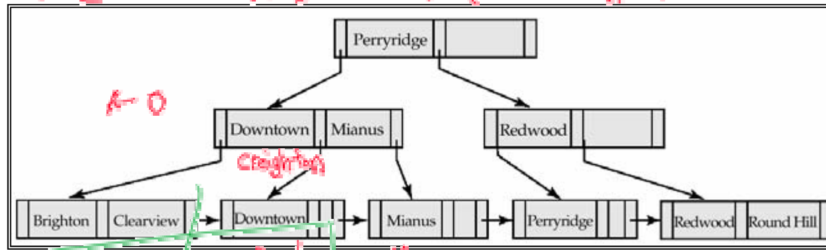


Result of splitting node containing Brighton and Downtown on inserting Clearview

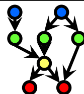


Updates on B+-Trees: Insertion (Cont.)



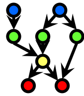


B+-Tree before and after insertion of "Clearview"



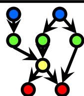
Insertion in B+-Trees (Cont.)

- **Read pseudocode in book!**



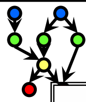
Updates on B^+-Trees: Deletion

- 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
  - 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.

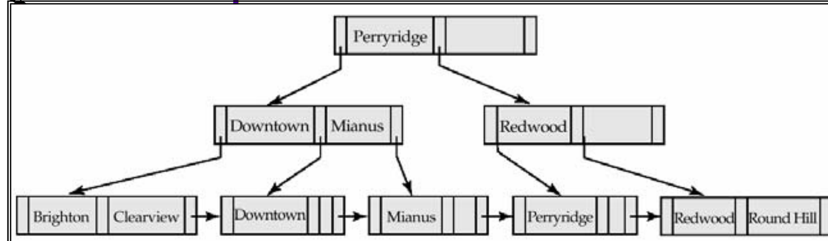


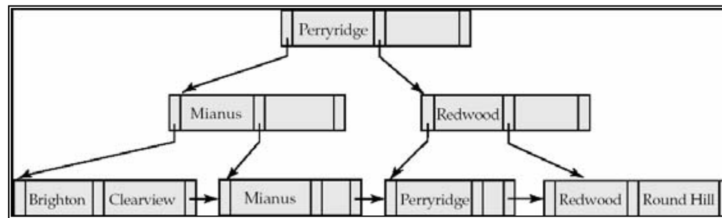
Updates on B^+-Trees: Deletion

- 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.



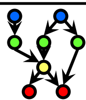
Examples of B+-Tree Deletion



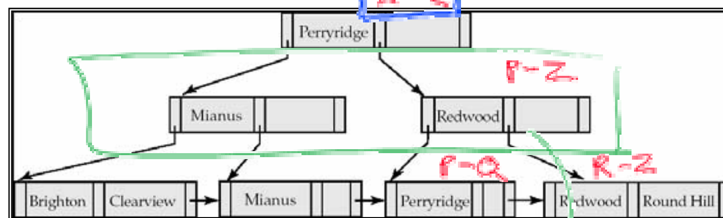


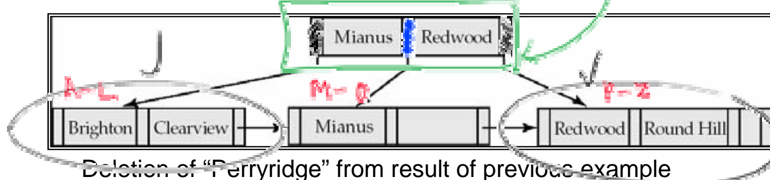
Before and after deleting "Downtown"

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



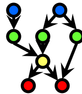
Examples of B+-Tree Deletion (Cont.)



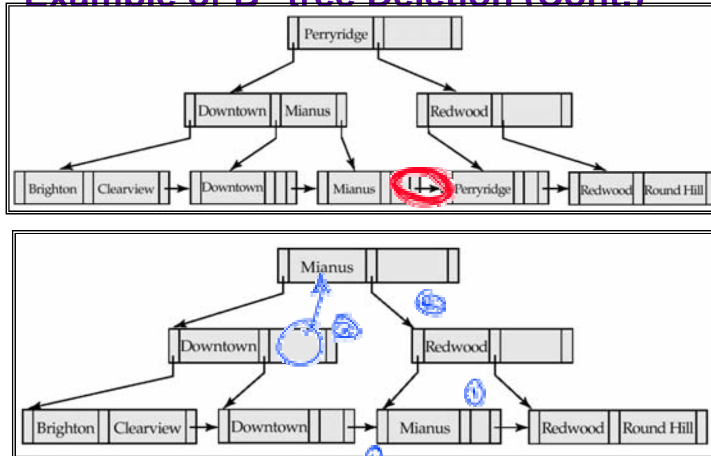


Deletion of "Perryridge" from result of previous example

- 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

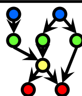


Example of B+-tree Deletion (Cont.)



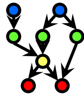
Before and after deletion of “Perryridge” from earlier example

- 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

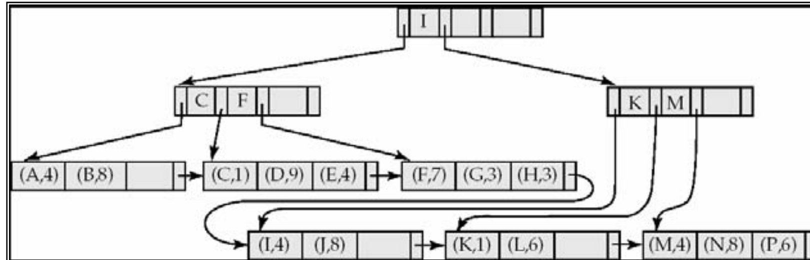


B^+-Tree File Organization

- 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.

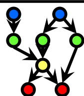


B+-Tree File Organization (Cont.)



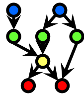
Example of B+-tree File Organization

- 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



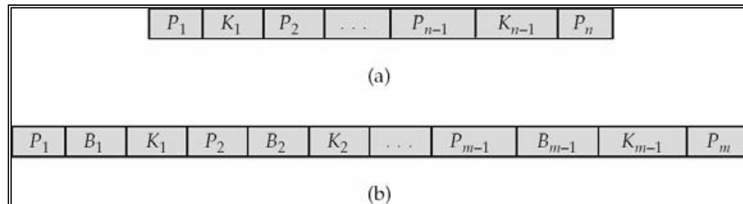
Indexing Strings

- Variable length strings as keys
  - Variable fanout
  - Use space utilization as criterion for splitting, not number of pointers
- Prefix compression
  - Key values at internal nodes can be prefixes of full key
    - Keep enough characters to distinguish entries in the subtrees separated by the key value
      - E.g. “Silas” and “Silberschatz” can be separated by “Silb”

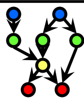


B-Tree Index Files

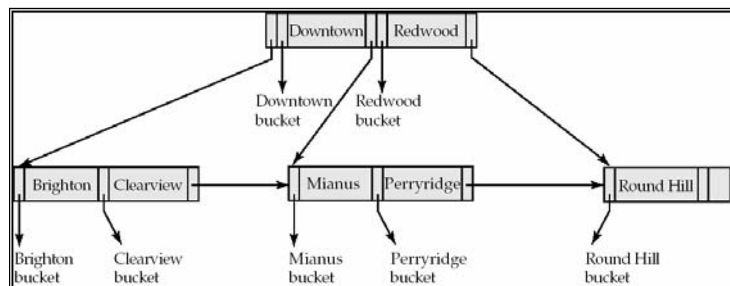
- Similar to B+-tree, but B-tree allows search-key values to appear only once; eliminates redundant storage of search keys.
- Search keys in nonleaf nodes appear nowhere else in the B-tree; an additional pointer field for each search key in a nonleaf node must be included.
- Generalized B-tree leaf node



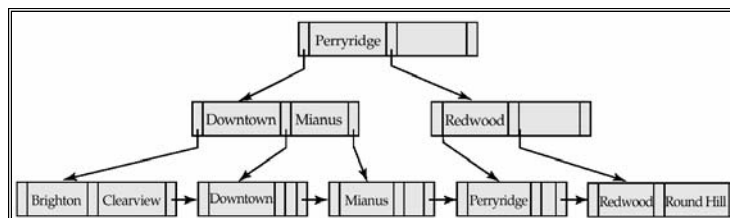
- Nonleaf node – pointers Bi are the bucket or file record pointers.

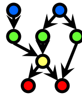


B-Tree Index File Example



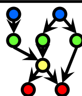
B-tree (above) and B+-tree (below) on same data





B-Tree Index Files (Cont.)

- 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 out weigh disadvantages.

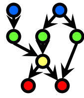


Multiple-Key Access

- Use multiple indices for certain types of queries.
- Example:

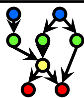
  select *account_number*
  from *account*
  where *branch_name* = "Perryridge" **and** *balance* = 1000
- Possible strategies for processing query using indices on single attributes:
  1. Use index on *branch_name* to find accounts with balances of $1000; test *branch_name* = "Perryridge".
  2. Use index on *balance* to find accounts with balances of $1000; test *branch_name* = "Perryridge".
  3. Use *branch_name* index to find pointers to all records pertaining to the Perryridge branch. Similarly use index on *balance*. Take intersection of both sets of pointers obtained.

CIS 560: Database System Concepts Wednesday, 25 Oct 2006 Computing & Information Sciences Kansas State University



Indices on Multiple Keys

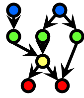
- **Composite search keys** are search keys containing more than one attribute
  - E.g. (*branch_name, balance*)
- Lexicographic ordering: $(a_1, a_2) < (b_1, b_2)$ if either
  - $a_1 < a_2$, or
  - $a_1 = a_2$ and $a_2 < b_2$



Indices on Multiple Attributes

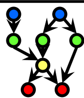
Suppose we have an index on combined search-key (*branch_name, balance*).

- With the **where** clause
  where *branch_name* = "Perryridge" **and** *balance* = 1000
  the index on (*branch_name, balance*) can be used to fetch only records that satisfy both conditions.
  - Using separate indices in less efficient — we may fetch many records (or pointers) that satisfy only one of the conditions.
- Can also efficiently handle
  where *branch_name* = "Perryridge" **and** *balance* < 1000
- But cannot efficiently handle
  where *branch_name* < "Perryridge" **and** *balance* = 1000
  - May fetch many records that satisfy the first but not the second



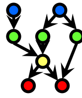
Non-Unique Search Keys

- Alternatives:
  - Buckets on separate block (bad idea)
  - List of tuple pointers with each key
    - Extra code to handle long lists
    - Deletion of a tuple can be expensive
    - Low space overhead, no extra cost for queries
  - Make search key unique by adding a record-identifier
    - Extra storage overhead for keys
    - Simpler code for insertion/deletion
    - Widely used



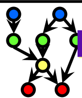
Other Issues

- Covering indices
  - Add extra attributes to index so (some) queries can avoid fetching the actual records
    - Particularly useful for secondary indices
      - Why?
  - Can store extra attributes only at leaf
- Record relocation and secondary indices
  - If a record moves, all secondary indices that store record pointers have to be updated
  - Node splits in B^+-tree file organizations become very expensive
  - Solution: use primary-index search key instead of pointer in secondary index
    - Extra traversal of primary index to locate record
      - Higher cost for queries, but node splits are cheap
    - Add record-id if primary-index search key is non-unique



Static Hashing

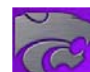
- A **bucket** is a unit of storage containing one or more records (a bucket is typically a disk block).
- In a **hash file organization** we obtain the bucket of a record directly from its search-key value using a **hash function.**
- Hash function *h* is a function from the set of all search-key values *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 entire bucket has to be searched sequentially to locate a record.

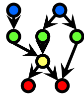


Example of Hash File Organization (Cont.)

Hash file organization of *account* file, using *branch_name* as key
(See figure in next slide.)

- 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(Perryridge) = 5 h(Round Hill) = 3 h(Brighton) = 3

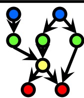




Example of Hash File Organization

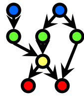
Hash file organization of *account* file, using *branch_name* as key (see previous slide for details).

Bucket	Account	Branch	Balance
bucket 0			
bucket 1			
bucket 2			
bucket 3	A-217	Brighton	750
	A-305	Round Hill	350
bucket 4	A-222	Redwood	700
bucket 5	A-102	Perryridge	400
	A-201	Perryridge	900
	A-218	Perryridge	700
bucket 6			
bucket 7	A-215	Mianus	700
bucket 8	A-101	Downtown	500
	A-110	Downtown	600
bucket 9			



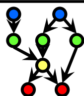
Hash Functions

- 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. .



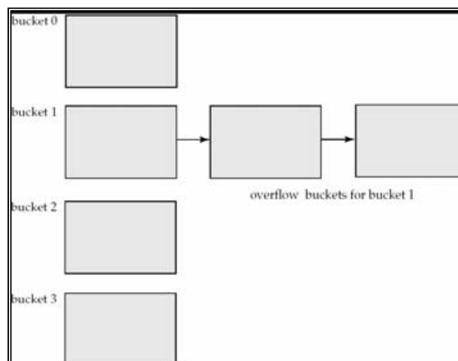
Handling of Bucket Overflows

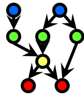
- Bucket overflow can occur because of
  - Insufficient buckets
  - Skew in distribution of records. This can occur due to two reasons:
    - multiple records have same search-key value
    - chosen hash function produces non-uniform distribution of key values
- Although the probability of bucket overflow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



Handling of Bucket Overflows (Cont.)

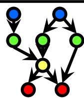
- Overflow chaining – the overflow buckets of a given bucket are chained together in a linked list.
- Above scheme is called closed hashing.
  - An alternative, called open hashing, which does not use overflow buckets, is not suitable for database applications.



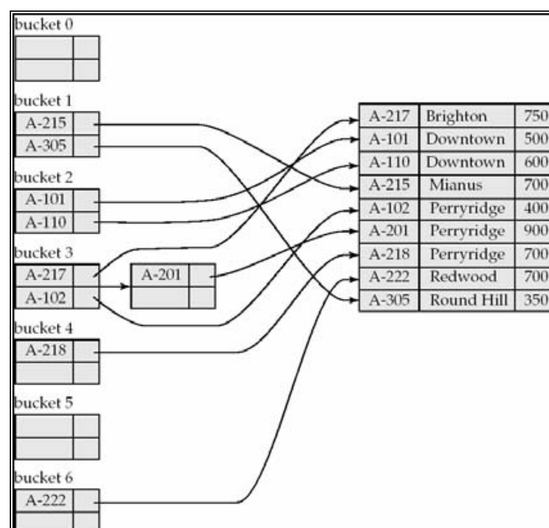


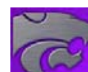
Hash Indices

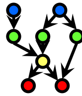
- Hashing can be used not only for file organization, but also for index-structure creation.
- A **hash index** organizes the search keys, with their associated record pointers, into a hash file structure.
- Strictly speaking, hash indices are always secondary indices
  - if the file itself is organized using hashing, a separate primary hash index on it using the same search-key is unnecessary.
  - However, we use the term hash index to refer to both secondary index structures and hash organized files.



Example of Hash Index







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, $0 \le i \le 32$.
  - 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.