Tree-Structured Indexing

Linda Wu

(CMPT 354 • 2004-2)

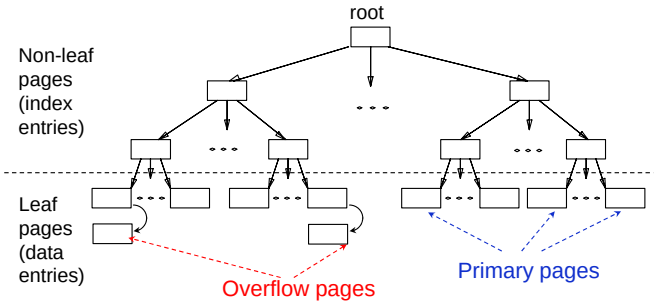
Topics

- ISAM
- B+ tree
  - Algorithm for search
  - Algorithm for insertion
  - Algorithm for deletion
  - Prefix key compression
  - Bulk loading
  - Order

Two Tree Index Structures

- ISAM (Indexed Sequential Access Method)
  - Static index structure
- B+ tree
  - Dynamic index structure
  - Adjusts gracefully under inserts and deletes

ISAM



ISAM (Cont.)

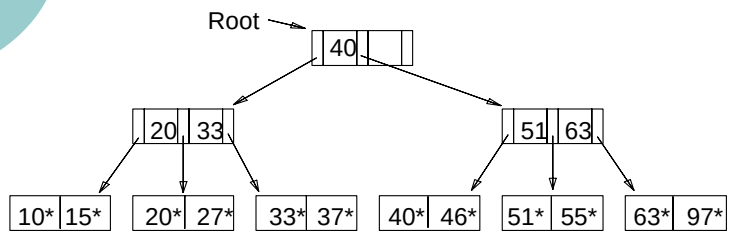
- Index file creation
  1) Primary pages are allocated sequentially
     - If alternative (1): all the data reside in the leaf pages, sorted by the search key value
     - If alternative (2), (3): the data records are stored in a separate file; sorted before allocating the leaf pages
  2) Index entry pages are then allocated
  3) Then space for overflow pages
     - Overflow pages are need if more entries inserted into a leaf cannot fit into a single page

ISAM (Cont.)

- Equality search
  - Start at root; use key comparisons to go to leaf
- Range search
  - Determine the starting point in the leaf level by equality search
  - Retrieve primary pages sequentially and overflow pages as needed by pointers from primary pages
- Insert
  - Find the leaf page the entry belong to, and put it there; add overflow page if needed
- Delete
  - Find and remove the entry from leaf page; if empty overflow page, de-allocate

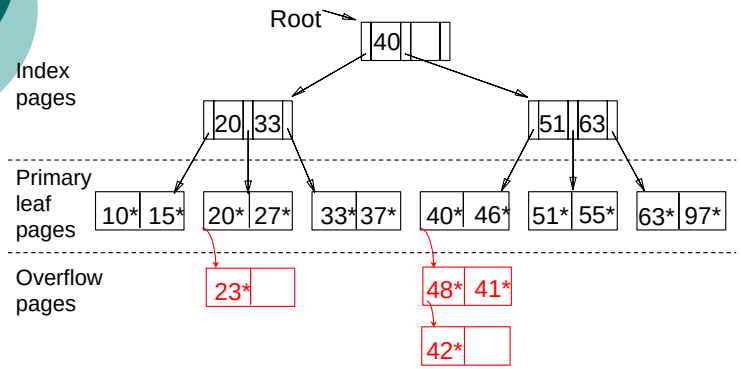
ISAM (Cont.)

- Example ISAM tree
  - Each node can hold 2 entries



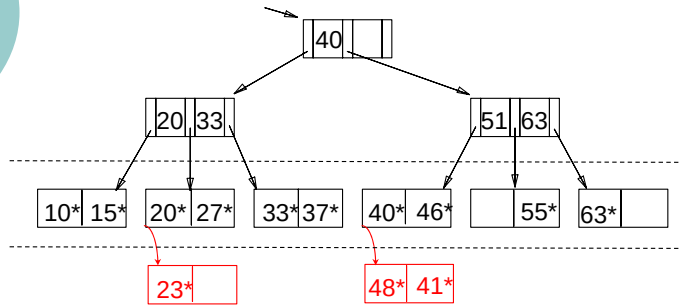
ISAM (Cont.)

- After inserting 23*, 48*, 41*, 42* ...



ISAM (Cont.)

- After deleting 42*, 51*, 97*



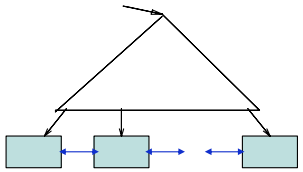
(51* appears in index level, but not in leaf!)

ISAM (Cont.)

- Static tree structure
  - Primary pages are sequentially allocated
  - Inserts / deletes affect leaf pages only; index entry pages are never modified
  - No need for "next page" pointers
  - Disadvantage: possibly long overflow chains, which are usually not sorted
  - Advantage: not locking index entry pages in case of concurrent transactions
  - ISAM might be preferable to B+ tree if overflow chains are rare

B+ Tree: Most Widely Used Index

- Tree is height-balanced; insert/delete at $\log_F N$ cost (F = fan-out, N = # leaf pages)
- A minimum 50% occupancy for each node except the root
  - Each node contains m entries
  - Non-root node: $d \le m \le 2d$; root, $1 \le m \le 2d$
  - d: order of the tree



Index Entries
(Direct search)

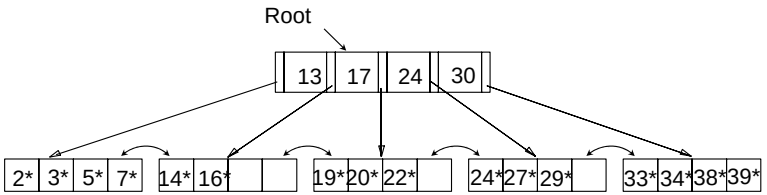
Data Entries
("Sequence set")

B+ Tree (Cont.)

- B+ tree in practice
  - Typical order: 100; typical fill-factor: 67%
    - Average fan-out = 133
  - Typical capacities
    - Height 4: 133^4 = 312,900,700 records
    - Height 3: 133^3 = 2,352,637 records
  - Top levels can often be held in buffer pool
    - Level 1 = 1 page = 8 Kbytes
    - Level 2 = 133 pages ≈ 1 Mbytes
    - Level 3 = 17,689 pages ≈ 133 Mbytes
- B+ tree can be viewed as a file of records

B+ Tree (Cont.)

- Example B+ tree below, order d = 2
- Search
  - Begins at root, and key comparisons direct it to a leaf (as in ISAM)
  - Search for 5*, 15*, all data entries ≥ 24* ...



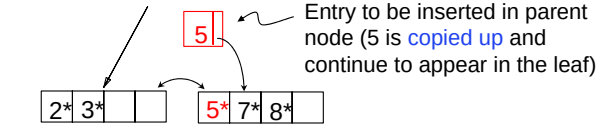
B+ Tree (Cont.)

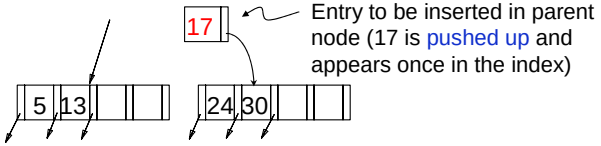
Algorithm for insertion

1) Find correct leaf L
2) Put data entry onto L
   - If L has enough space, done!
   - Else, split L (into L and a new node L2)
     - Redistribute entries evenly, copy up middle key
     - Insert an index entry pointing to L2 into the parent of L
3) Split index node(s) if needed
   - If the parent of L (an index node) is full: split the index node, redistribute entries evenly, but push up middle key (may happen recursively!)

B+ Tree (Cont.)

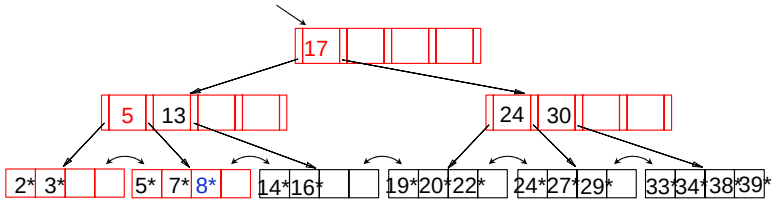
- Inserting 8*
  - Minimum occupancy is guaranteed in both leaf and index page splits





B+ Tree (Cont.)

- Example B+ tree after inserting 8*
  - The root was split, leading to increase in height
  - In this example, we can avoid split by redistributing entries; however, this is usually not done in practice



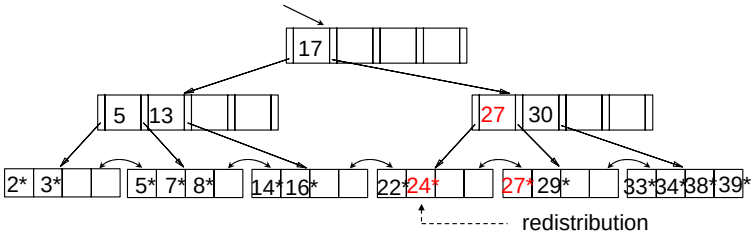
B+ Tree (Cont.)

- Algorithm for deletion

1) Start at root, find leaf L where entry belongs
2) Remove the entry
   - If L is at least half-full after deletion, done!
   - If L has only d-1 entries after deletion
     - Try to redistribute, borrowing from sibling
     - If redistribution fails, merge L and sibling, and delete index entry (pointing to L or sibling) from parent of L
     - Merge could propagate to root, decreasing tree height

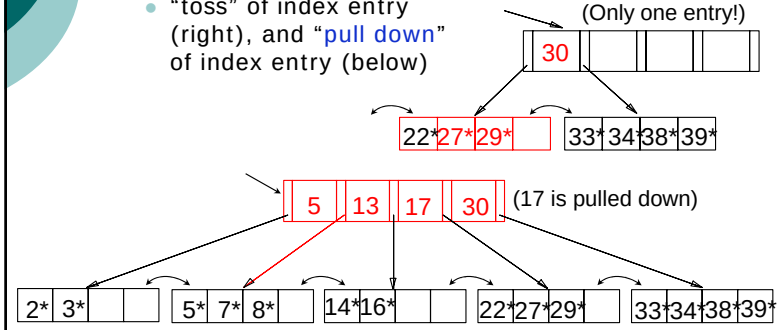
B+ Tree (Cont.)

- Example B+ tree after (inserting 8*, then) deleting 19* and 20* ...
  - Deleting 19* is easy
  - Deleting 20* is done with redistribution (middle key 27 is copied up)



B+ Tree (Cont.)

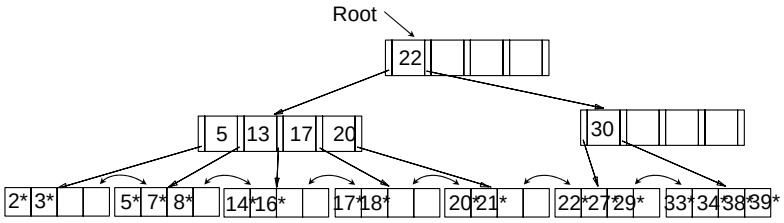
- And then deleting 24*
  - Must merge
  - "toss" of index entry (right), and "pull down" of index entry (below)



B+ Tree (Cont.)

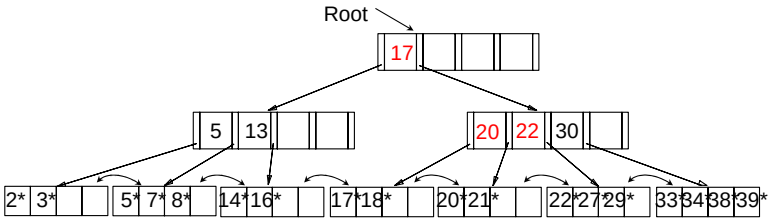
Example of non-leaf redistribution

- B+ tree during the deletion of 24*
  - In contrast to previous example, redistribution is possible (redistribute entry from left child of root to right child)



B+ Tree (Cont.)

- After redistribution
  - Entries are redistributed by "pushing through" the splitting entry in the parent node
  - It suffices to redistribute index entry with key 20; "17" is redistributed as well for illustration

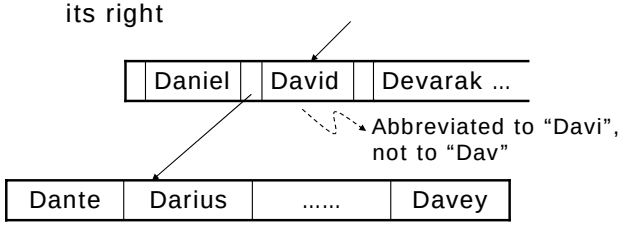


Prefix Key Compression

- It is important to increase fan-out
  - Smaller tree height means less disk I/Os to retrieve a data entry
- Increasing fan-out by decreasing the size of index entries
  - Search key values in index entries only "direct traffic" to leaf level
  - Search key values can often be compressed: we need not store search key values in their entirety in index entries

Prefix Key Compression (Cont.)

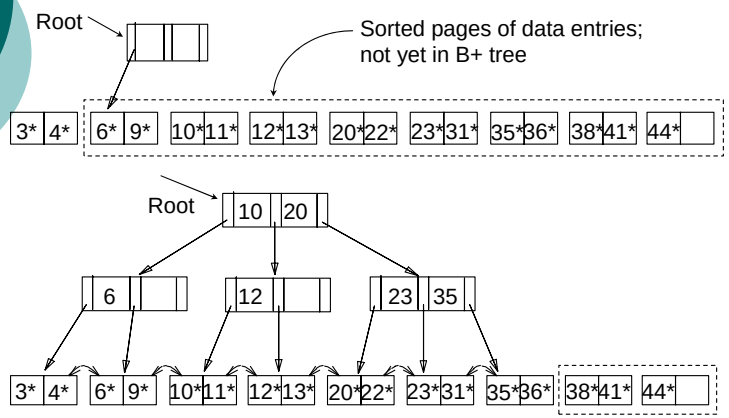
- Prefix key compression
  - In general, while compressing, an index entry must be greater than every key value in any subtree to its left, and less than or equal to every key value in any subtree to its right



Bulk Loading a B+ Tree

- To create a B+ tree with a large collection of records
  - Doing so by multiple inserts (repeatedly inserting records) is very slow
- Bulk loading
  - Initialization: sort all data entries, allocate an empty page as root, and insert a pointer to first (leaf) page into the root
  - Add one entry to the root for each page, till the root is full; split the root
  - Always insert entries for the leaf pages into the right-most index page just above the leaf level; when this fills up, it is split

Bulk Loading (Cont.)



Bulk Loading (Cont.)

- Option 1: multiple inserts
  - Slow
  - Does not give sequential storage of leaves
- Option 2: bulk loading
  - Fewer I/Os during build
  - Leaves will be stored sequentially (and linked, of course)
  - Can control “fill factor” on pages

A Note on Order

- Order (d) concept should be relaxed and replaced by physical space criterion in practice (e.g., at least half-full)
  - Index pages can typically hold many more entries than leaf pages
  - Variable sized records and search keys mean different nodes will contain different numbers of entries
  - Even with fixed length fields, multiple records with the same search key value (duplicates) can lead to variable-sized data entries if we use Alternative (3)

Summary

- Tree-structured indexes are ideal for range searches, also good for equality searches
- ISAM is a static structure
- B+ tree is a dynamic structure
- Prefix key compression increases fan out, reduces height
- Bulk loading can be much faster than repeated inserts for creating a B+ tree on a large data set
- B+ tree structure is most widely used index; one of the most optimized components of a DBMS