Tree-Structured Indexes

Chapter 10

Introduction

- Tree-structured indexing techniques support both *range searches* and *equality searches*.
  - *ISAM*: static structure (pre 1970)
  - B tree: dynamic structure (around 1971)
  - *B+ tree*: dynamic, adjusts gracefully under inserts and deletes. (improved B Tree, after 1975)

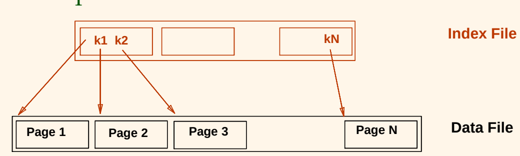
Slide 2

Introduction

- *As for any index, 3 alternatives for data entries* **k***:
  - Data record with key value **k**
  - <**k**, rid of data record with search key value **k**>
  - <**k**, list of rids of data records with search key **k**>
- Choice is orthogonal to the *indexing technique* used to locate data entries **k***.
- Tree-structured indexing techniques support both *range searches* and *equality searches*.
- *ISAM*: static structure; *B+ tree*: dynamic, adjusts gracefully under inserts and deletes.

Range Searches

- ``*Find all students with gpa* > *3.0*''
  - If data is in sorted file, do binary search to find first such student, then scan to find others.
  - Cost of binary search can be quite high.
- Simple idea: Create an `index' file.



- *Can do binary search on (smaller) index file!*

Example (sorted vs. tree indexing)

- 10000 pages sorted (contiguous)
- Search on the data pages takes $\log_2 10000$ or 14 page accesses ($\log_2 10000$ is 14 + 1 more for data access)
- Let the record size be 200 bytes; key size be 10 bytes, 40 records/page (8K page size)
- Now the data pages and leaf nodes of the index for the above are as follows:
  - 10000*40 = 400000 keys
  - Key, ptr pairs per page is 8000/20 = 400 (10bytes for pointer and 10 bytes for the key)
  - Fan out of the index tree is ~ 400
  - # of index leaf pages is 400000/400 = 1000 pages
- We reduce page access from 14+1 to 2+1 (why?) with the index
- $\log_{400} 1000$ is 2 (+ 1 more access for the data)

Indexes

- Binary Trees
- AVL Trees
- ISAM (used by DBMSs before B and B+ trees)
- B Trees (height Balanced)
- B+ trees (also height balanced)

Index Characteristics

- Static Vs. Dynamic
- Top Down Vs. Bottom up
- Fixed number Vs. dynamic number of index pages
- Balanced Vs. Unbalanced

Binary and AVL trees

- Binary trees
  - At most 2 descendents (or children) per node.
  - No constraint on the length of the paths from the root.
- AVL trees
  - An AVL tree is a binary tree in which the difference between the height of the right and left sub-trees (or the root node) is never more than one.

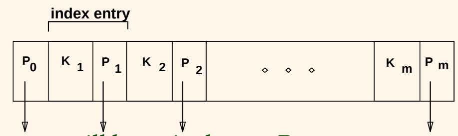
Binary and Avl trees

- Binary and AVL are top-down tree constructions. Once the wrong key is placed in the root of the tree (or in the root of any sub tree), it is difficult to balance the tree without significant overhead (reorganization)
- How can we guarantee that each of the pages contain at least some minimum number of keys (important for large page sizes)
- How can we guarantee that the heights of different paths are the same (or are not very different)

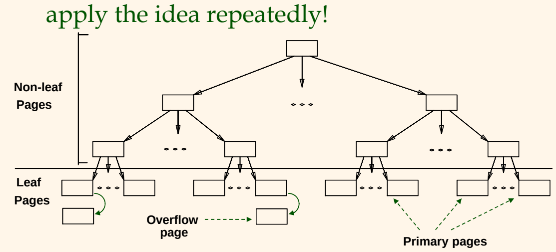
Balanced (B and B+) trees

- A balanced tree builds the tree upward from the bottom instead of downward (like AVL and binary trees) from the top.
- Rather than finding ways to undo a bad situation, the problem is avoided altogether from the very beginning.
- With balanced trees, you allow the root to split and merge, rather than set it up and find ways to change it.

ISAM

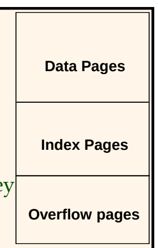


- Index file may still be quite large. But we can apply the idea repeatedly!



- *Leaf pages contain data entries.*

Comments on ISAM

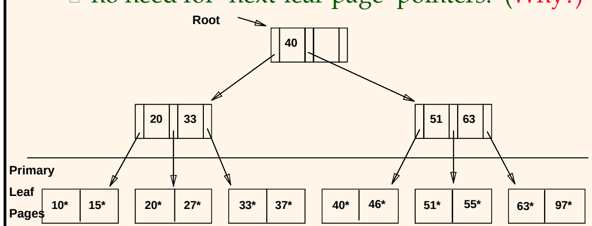


- *File creation*: Leaf (data) pages allocated sequentially (or contiguously), sorted by search key; then index pages allocated, then space for overflow pages.
- *Index entries*: <search key value, page id>; they `direct' search for *data entries*, which are in leaf pages.
- *Search*: Start at root; use key comparisons to go to leaf. Cost $\propto \log_F N$; F = # entries/index pg, N = # leaf pgs
- *Insert*: Find leaf where data entry belongs to, and put it there.
- *Delete*: Find and remove from leaf; if empty overflow page, de-allocate.

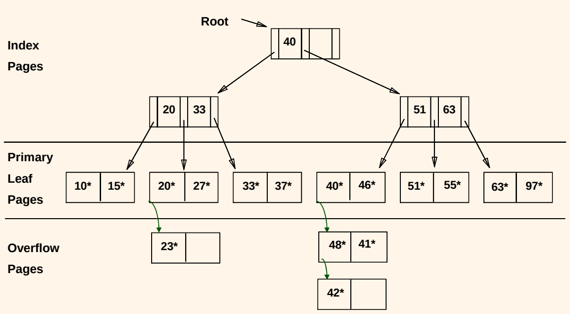
- **Static tree structure**: *inserts/deletes affect only leaf pages.*

Example ISAM Tree

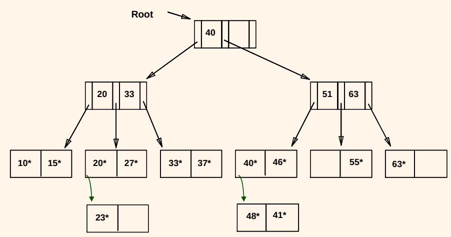
- Each node can hold 2 entries;
- no need for `next-leaf-page' pointers. (Why?)



After Inserting 23, 48*, 41*, 42* ...*



... Then Deleting 42, 51*, 97**



- *Note that 51* appears in index levels, but not in leaf!*

Summary of ISAM

- ISAM structure is created for a given file
- Number of index pages (entries) do not change
- Overflow pages are added and deleted as needed
- The number of primary leaf pages do not change
- Leaf pages are allocated sequentially (hence no need for pointers!)
- No need to lock index pages (more concurrency) why?
- The index tree is NOT balanced dynamically (balanced statically at creation time)
- Index value exists, but may not be a record for that value
- Potentially long overflow lists
- May need to re-create ISAM index to overcome the above

B and B+ trees

- The common theme of all index structures is that they associatively map some attribute of a data object to some locator information which can be used to retrieve the actual data object.
- Typically, index scans are separated from record lookup:
  - Allows to scan an index without retrieving data from the data file (reduces I/O for count and other computations)
  - For B+ trees, leaves can be accessed sequentially
  - Joining of indices using record id (rid/tid)

B and B+ trees

- Both are balanced trees; i.e., the length of the path from the root to any leaf node is the same.
- Both store pointers to data (in Alt 2 and 3) in the index nodes. Alt 1 is typically not used!!
- The data records are stored on separate pages.
- Both are constructed in a bottom-up manner
- However, there are some fundamental differences between them

FIGURE 14.10
B-tree structures. (a) A node in a B-tree with q – 1 search values. (b) A B-tree of order p = 3. The values were inserted in the order 8, 5, 1, 7, 3, 12, 9, 6.

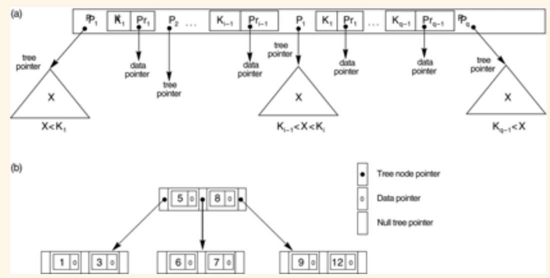
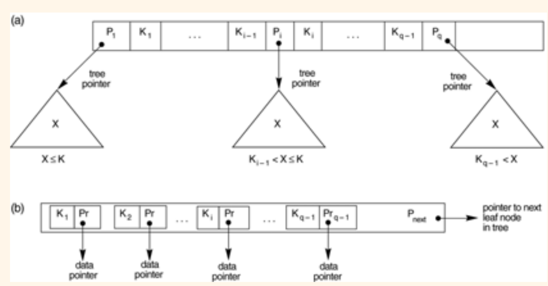


FIGURE 14.11
The nodes of a B+-tree. (a) Internal node of a B+-tree with q –1 search values. (b) Leaf node of a B+-tree with q – 1 search values and q – 1 data pointers.



B-Tree

- B tree index nodes/pages (both leaf and non-leaf) contain
  <key, data ptr, child ptr> whereas a
- B+ tree non-leaf index node/page contains
  <key, child ptr> pairs and
- B+ tree index leaf node/page contains
  <key, data ptr> pairs
- B+ tree leaf index pages/nodes are double linked
- Data pages are separate in both the cases

B Tree -- Implications

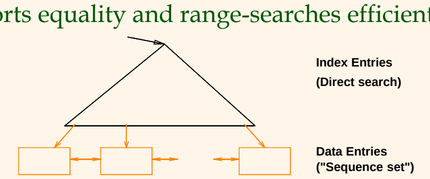
- In a B tree (as compared to a B+ tree):
  - Packing density of an index page is less (why?)
  - Key values are NOT repeated in the index (why?)
  - Leaf index pages cannot be traversed for scan (why?)
  - For a key that exists in the file, search may be stopped before reaching the leaf node (why?)

B Tree -- Implications

- Sequential access of the entire file requires touching every node in the index (why?)
  - What kind of traversal is needed?
- Perhaps good (why?) when the key forms most of the data record.
  - i.e., the ratio of data record size to key size is closer to 1!

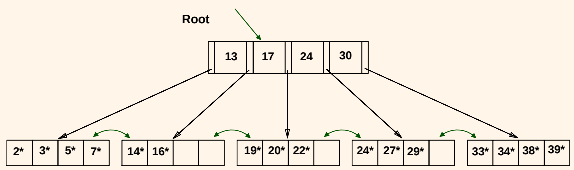
B+ Tree: Most Widely Used Index

- Insert/delete at log $_F$ N cost; keep tree *height-balanced*.   (F = fanout, N = # leaf pages)
- Minimum 50% occupancy (except for root). Each node contains **d** <= <u>*m*</u> <= 2**d** entries. The parameter **d** is called the *order* of the tree.
- Supports equality and range-searches efficiently.



Example B+ Tree

- Search begins at root, and key comparisons direct it to a leaf (as in ISAM).
- Search for 5*, 15*, all data entries >= 24* ...



- *Based on the search for 15*, we <u>know</u> it is not in the tree!*

B+ Trees in Practice

- Typical order: 100+. Typical fill-factor: 67%.
  - average fanout = 133
- Typical capacities (not including root):
  - Height 1: 133^1 = 133 nodes (can access 133^2 data recs)
  - Height 2: 133^2 = 17689 nodes
  - Height 3: 133^3 = 2,352,637 nodes
  - Height 4: 133^4 = 312,900,700 nodes
    - (can access 4.16^{10} records)
- Remember that each leaf index node holds 133 keys and 133 record pointers
- Can often hold top levels in buffer pool:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes

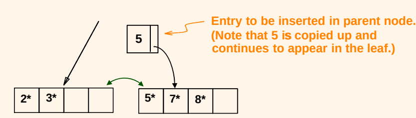
Inserting a Data Entry into a B+ Tree

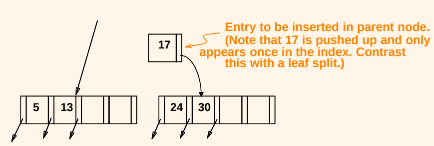
- Find correct leaf node *L.*
- Put data entry onto *L.*
  - If *L* has enough space, *done*!
  - Else, either
    - i) Redistribute entries evenly, **<u>copy up</u>** middle key, or
    - ii) *<u>split</u> L (into L and a new node L2) and* Insert index entry pointing to *L2* into parent of *L*.

    // Can avoid split sometimes by redistributing
    // but need a proper sibling, and update parent node!
- This can happen recursively
  - For an index node, you can either redistribute entries evenly or split, and **<u>push up</u>** middle key. (Contrast with leaf splits.)
- Splits "grow" tree; root split increases height.
  - Tree growth: gets *<u>wider</u>* or *<u>one level taller at top.</u>*

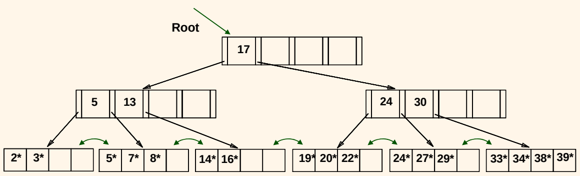
Inserting 8 into Example B+ Tree*

- Observe how minimum occupancy is guaranteed in both leaf and index pg splits.
- Note the difference between *copy-up* and *push-up;* be sure you understand the reasons for this.





*Example B+ Tree After Inserting 8**

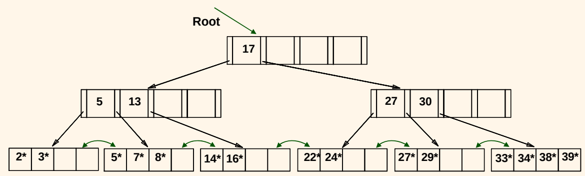


- Notice that root was split, leading to increase in height.
- In this example, we can avoid split by re-distributing entries; however, this is usually not done in practice.

Deleting a Data Entry from a B+ Tree

- Start at root, find leaf *L* where entry belongs.
- Remove the entry.
  - If L is at least half-full, *done!*
  - If L has only **d-1** entries,
    - Try to re-distribute, borrowing from *sibling (adjacent node with same parent as L)*. Middle key is copied up
    - If re-distribution fails, *merge L* and sibling.
- If merge occurred, must delete entry (pointing to *L* or sibling) from parent of *L*.
- Merge could propagate to root, decreasing height.
- If re-distribution, update the parent node to reflect this (copy up)! Not move up!
- Recursive algorithm

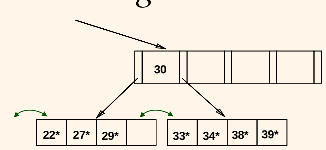
Example Tree After (Inserting 8, Then) Deleting 19* and 20* ...*

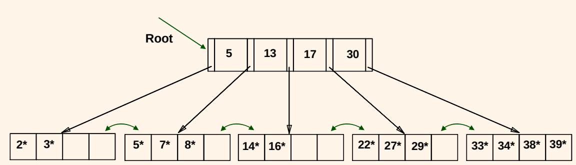


- Deleting 19* is easy.
- Deleting 20* is done with re-distribution. Notice how middle key is *copied up*.

*... And Then Deleting 24**

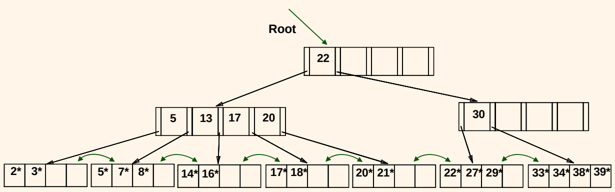
- Must merge.
- Observe *`toss'* of index entry (on right), and *`pull down'* of index entry (below).





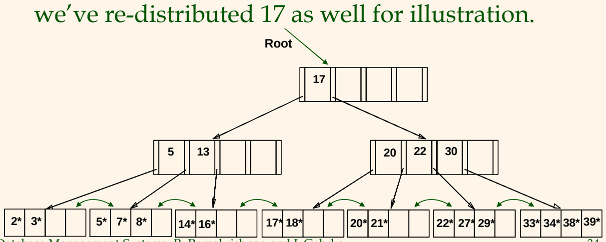
Example of Non-leaf Re-distribution

- Tree is shown below *during deletion* of 24*. (What could be a possible initial tree?)
- In contrast to previous example, can re-distribute entry from left child of root to right child.



After Re-distribution

- Intuitively, entries are re-distributed by `*pushing through*' the splitting entry in the parent node.
- It suffices to re-distribute index entry with key 20; we've re-distributed 17 as well for illustration.

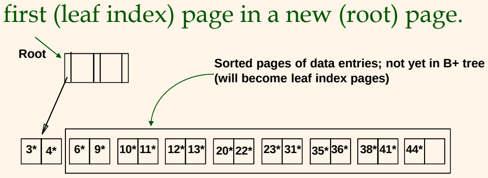


Prefix Key Compression

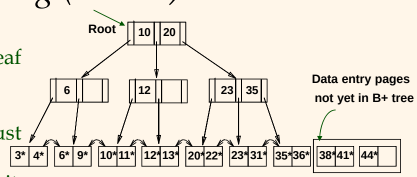
- Important to increase fan-out. (Why?)
- Key values in index entries only `direct traffic'; can often compress them.
  - E.g., If we have adjacent index entries with search key values *Dannon Yogurt*, *David Smith* and *Devarakonda Murthy*, we can abbreviate *David Smith* to *Dav*. (The other keys can be compressed too ...)
    - Is this correct? Not quite! What if there is a data entry *Davey Jones*? (Can only compress *David Smith* to *Davi*)
    - In general, while compressing, must leave each index entry greater than every key value (in any subtree) to its left.
- Insert/delete must be suitably modified.

Bulk Loading of a B+ Tree

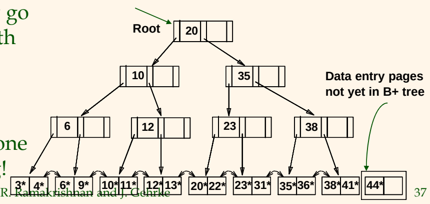
- If we have a large collection of records, and we want to create a B+ tree on some field, doing so by repeatedly inserting records is very slow.
- <u>*Bulk Loading*</u> can be done much more efficiently.
- *Initialization*: Sort all data entries, insert pointer to first (leaf index) page in a new (root) page.



Bulk Loading (Contd.)



- Index entries for leaf pages always entered into right-most index page just above leaf level. When this fills up, it splits. (Split may go up right-most path to the root.)
- Much faster than repeated inserts, especially when one considers locking!



Cost of bulk loading (creating an index)

1) Creating the data entries to insert in the index
2) Sort the data entries
3) Building the index

Cost of 1) is (R+E), R #pages containing records and E is the # of pages with data entries; access each page of R to get the data entries

Sort cost: approximately 4E

Cost of writing out all the index pages

Summary of Bulk Loading

- Option 1: multiple inserts.
  - Slow.
  - Does not give sequential storage of leaves.
- Option 2: <u>*Bulk Loading*</u>
  - Has advantages for concurrency control.
  - 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 replaced by physical space criterion in practice (`*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)).

A Note on Alt 3

- *Storing multiple data pointers is not desirable as it gives rise to variable size records in leaf index pages!*
- *For clustered B+ tree index, this can be done relatively easily by scanning additional data records on the same page as it is clustered on key value*
- *For non-clustered B+ tree index, the above cannot be done. (why?)*
  - *One possibility is to add overflow pages from the leaf index page that points to multiple data records!*

Summary

- B and B+ trees can be viewed as multi-level indexes or height balanced trees.
- B and B+ trees overcome the problems associated with binary and AVL trees
  - Binary requires too many i/os (log 2)
  - May be expensive to keep the index sorted
  - Very sensitive to order of inserts
  - Avl is a height balanced 1-tree or HB(1) tree
- B and B+ trees are height-balanced 0-tree or H(0) or completely balanced

Summary

- Tree-structured indexes are ideal for range-searches, also good for equality searches.
- ISAM is a static structure.
  - Only leaf pages modified; overflow pages needed.
  - Overflow chains can degrade performance unless size of data set and data distribution stay constant.
- B+ tree is a dynamic structure.
  - Inserts/deletes leave tree height-balanced; $\log_F N$ cost.
  - High fanout (**F**) means depth rarely more than 3 or 4.
  - Almost always better than maintaining a sorted file.

Summary (Contd.)

  - Typically, 67% occupancy on average.
  - Usually preferable to ISAM, modulo *locking* considerations; adjusts to growth gracefully.
  - If data entries are data records, splits can change rids!
- Key compression increases fanout, reduces height.
- Bulk loading can be much faster than repeated inserts for creating a B+ tree on a large data set.
- Most widely used index in database management systems because of its versatility. One of the most optimized components of a DBMS.

Thank You !