COSC2406/2407 Database Systems

Tree Index Structures

Xiangmin (Emily) Zhou

RMIT University
Room : 14.11.04
Online Consultation via Collaborate Ultra(no appointment required): 10:20-11.20am Thursdays
Email : xiangmin.zhou@rmit.edu.au

Lecture 5

References: Ramakrishnan & Gehrke Chapter 10
Garcia-Molina et al. Chapter 13
Elmasri & Navathe Chapter 5
Diagrams courtesy Ramakrishnan & Gehrke

Overview

In this lecture, we will discuss tree index structures.
Specifically, we will discuss:

- The Indexed Sequential Access Method (ISAM)
- Dynamic B+-trees

Tree-structured Indexes

- As for any index, 3 alternatives for index data entries $k*$:
  1. Data record with search key value k
  2. (k, `rid` of data record with search key value k)
  3. (k, list of `rids` of data records with search key k)

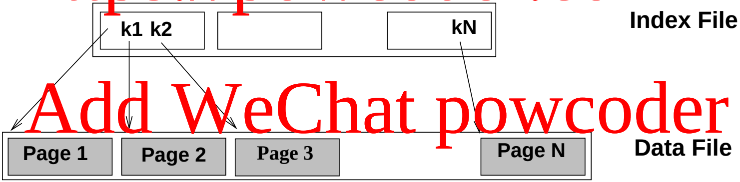
  – Choice is orthogonal to the indexing technique.

- Tree-structured indexing techniques support both range searches and equality searches
- ISAM (Indexed Sequential Access Method) is a tree-based static structure

- B+-tree is a dynamic structure that adjusts with insertions and deletions

Motivation for Tree Structures: Range Searches

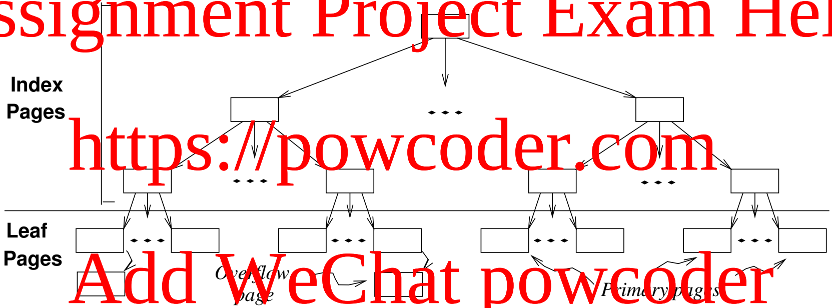
- *"Find all students older than 22."*
  - If data is in sorted file, we can binary search to find first such student, then scan to find other matches
  - The cost of binary search can be high
- Simple solution: create an “index” file (see Section 8.2)



- Now we can binary search on the (smaller) index file!

Motivation for Tree Structures: Range Searches ...

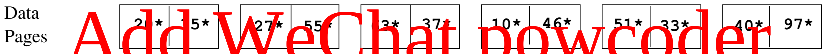
Index file may still be quite large. Apply idea repeatedly.



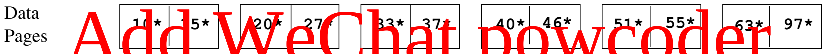
Leaf pages contain data entries.

Tree Indexes

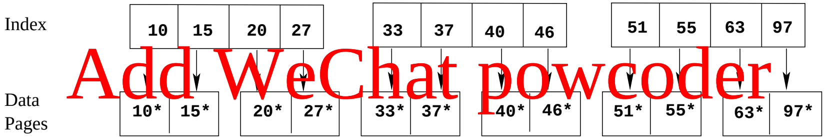
Consider a heap file, with 2 records per page:



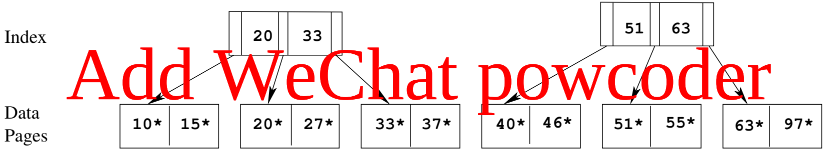
Sort file on search key:



Build an index on file:

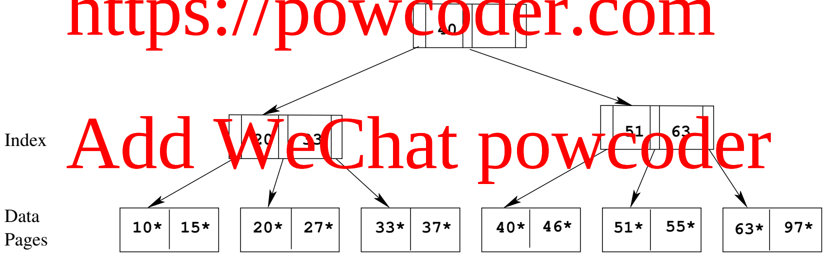


Build a sparse index on file:



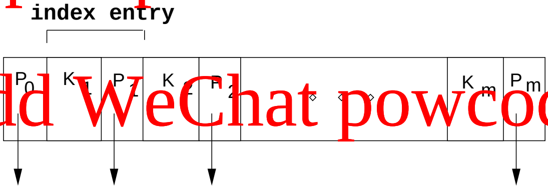
Tree Indexes...

Build an index on the index:



ISAM

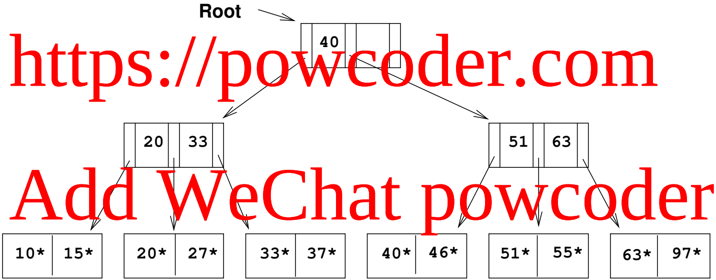
- The tree we have constructed is a so-called ISAM (Indexed Sequential Access Method) structure. (See Section 10.2)
- Each index page has *index data entries* of the form (key, pointer).
- Each index page contains one pointer more than the number of keys.
- A key serves as a *separator* between the contents of the pages pointed to by the pointers to its left and right.



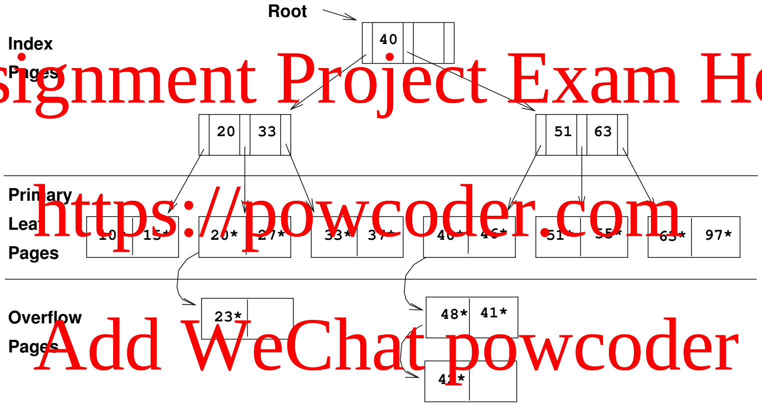
Comments on ISAM

- File creation: Leaf (data) pages allocated sequentially, sorted by search key; then index pages allocated, then space for overflow pages.
- Index entries: (search key value, page id); these “direct” the search for data entries, which are in leaf pages.
- Search: Start at the root; use key comparisons to go to a leaf. Cost: $\log_F N$, where F is *fan-out* of the index page and N is number of leaf pages.
- Insert: Find the leaf page where the data entry belongs to and put it there. Create overflow pages if necessary.
- Delete: Find and remove the entry from the leaf; if this empties an overflow page, de-allocate the page.

Each node can hold 2 entries; no need for 'next-leaf-page' pointers (because of the sequential allocation of leaf pages).

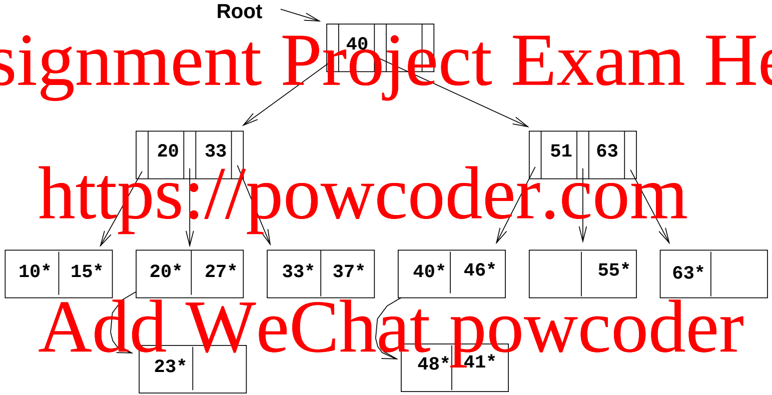


After Inserting 23*, 48*, 41*, 42* ...



Advantage: Less locking problems in ISAM than in other structures, since no index page is changed.

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



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

Overflow Chains

Once an ISAM file is created, inserts and deletes affect only the leaf page content.

As a result, long overflow chains can develop. This affects the time required to retrieve a record, since the chains have to be searched as well.

To reduce this problem, the tree can be created with some free space (20%, say) for future insertions.

Otherwise, the only way to get rid of overflow chains requires a reorganisation of the whole file structure.

ISAM and Locking

The fact that only leaf pages are adjusted gives a big advantage for *concurrent* access:

When a page is accessed, it is usually *locked* to ensure that it is not concurrently modified by another user. This can result in long queues of users waiting to access a page. Such a situation can cause significant performance issues, especially if the locked node is near the root of a tree.

Since ISAM index pages are never modified, they do not need to be locked.

B+-tree: The Most Widely Used Index

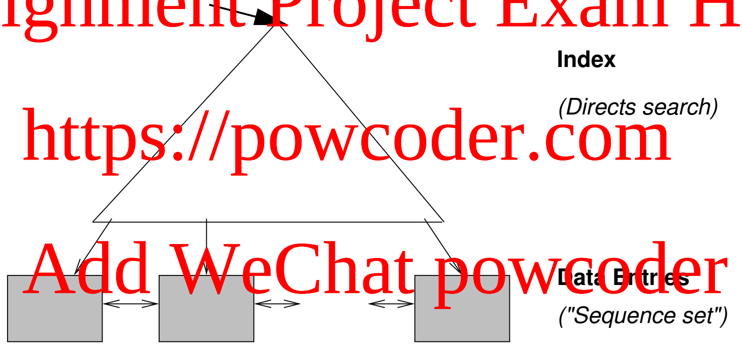
The B+-tree is the most widely used index structure (see Section 10.3). It has the following characteristics:

- Internal nodes direct the search, index data entries are in the leaf
- The tree is kept *height-balanced*
- Insert and delete at $\log_F N$ cost, where F is the fan-out and N the number of leaves

- Minimum 50% occupancy of nodes (except for the root).
- Each node contains $d <= m <= 2d$ entries, where the parameter d is called the *order* of the tree.

- Supports equality and range-searches efficiently
- Leaf pages are organised as a double linked list for fast traversal and reorganisation

B+-tree Index File



Node Format

Non-leaf nodes contain *m* index entries, with $m + 1$ pointers to children (like ISAM structure).

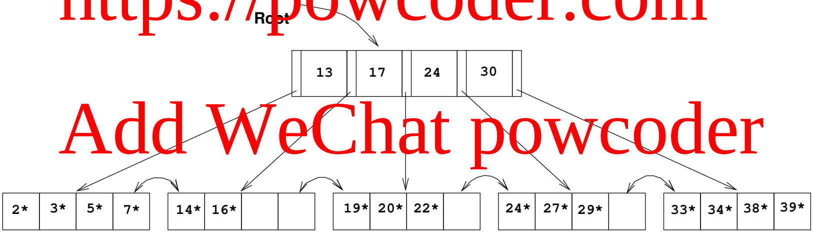
Leaf nodes contain data entries, either:

- actual data records (alternative 1); the B+-tree is an *integrated* index, where the file contains the index structure as well as the data
- pointers to data records elsewhere on disk (alternatives 2 & 3); the B+-tree is an *index file*, distinct from the file which contains the records

If the indexed field is of fixed-length, so is the index entry. Otherwise, the index entries are variable-length.

Example B+-tree

- A search begins at the root, and key comparisons direct the search to a leaf (as in ISAM)
- (Try a search for 5*, 15*, and all data entries $>=$ 24*)



Inserting a Data Entry into a B+-Tree

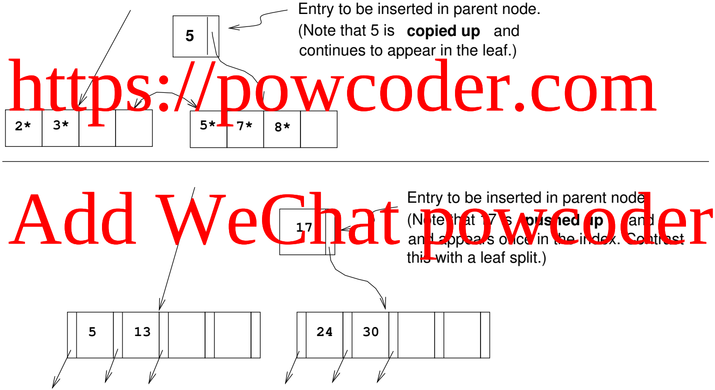
- Find correct leaf *L*.
- Put data entry into *L*.
  - If *L* has enough space, done!
  - Else, must *split* *L* (into *L* and a new node *L*2)
    - Redistribute entries evenly, *copy up* middle key.
    - Insert index entry pointing to *L*2 into parent of *L*.

- This can happen recursively.
  - To split index node, redistribute entries evenly, but *push-up* middle key. (Contrast to leaf splits!)

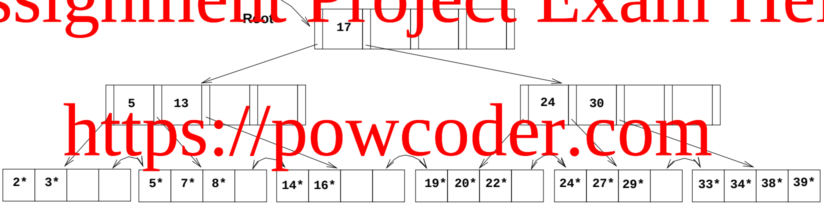
- Splits "grow" tree; root split increases height.
  - Tree growth: gets wider or one level taller at top.

Inserting 8* into Example B+-tree

- Observe how minimum occupancy level is guaranteed in both leaf and index pages splits.
- Note difference between *copy-up* and *push-up*!



Example B+-tree After Inserting of 8*



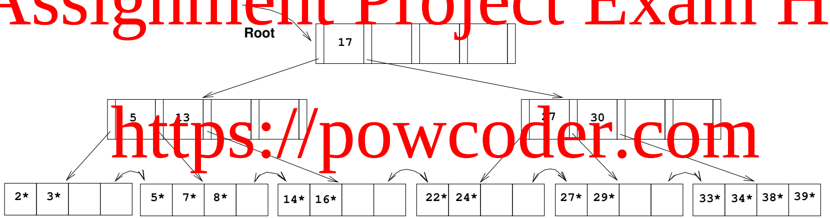
- Notice that root was split, leading to increase in height.
- In this example, we can avoid split by re-distributing entries; however, this is usually not done in practice.

Deleting a Data Entry from a B+-Tree

- Start at root; find the leaf *L* where the 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*) - If re-distributing 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.

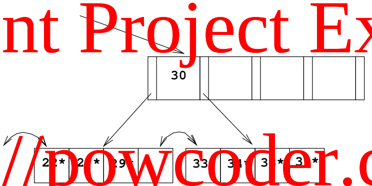
Refer to Section 10.6 for an example.

Garcia-Molina et al. illustrates a B-tree example in Section 13.3.6.

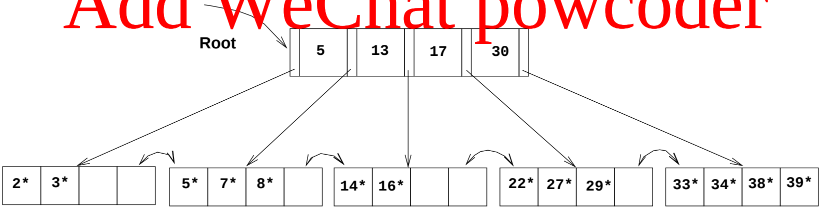


Deletion of 20* is solved by re-distribution from sibling node. Note that the new splitting (middle) key is copied up.

Leaf nodes must be merged:



Then, index nodes need to be merged and entry 17 is pulled down:



- Typical order (minimum node size): 100. Typical fill-factor: 67%.
  - Average fan-out = 133
    (increased by *key compression*)
- Typical Capacities:
  - Height 4: 133^4 = 312,900,700 records
  - Height 3: 133^3 = 2,352,637 records
- Can often hold top levels in the buffer manager's pool:
  - `Level 1 = 1 page = 8 Kbytes`
  - `Level 2 = 133 pages = 1 Mbyte`
  - `Level 3 = 17,689 pages = 133 Mbytes`

Read more indexes in MongoDB (including use of B+-trees)

`https://docs.mongodb.com/manual/indexes`

Read more about how B+-trees are implemented in Derby:

`http://db.apache.org/derby/papers/btree_package.html`

Summary: Tree Index Structures

We have discussed tree index structures in this lecture. Specifically, we have discussed Indexed Sequential Access Method (ISAM) and dynamic B+-trees.