Tree-Structured Indexes

Review: Files, Pages, Records

- Abstraction of stored data is "files" with "pages" of "records".
  - Records live on *pages*
  - Physical Record ID (RID) = <page#, slot#>
- *Variable length* data:
  - Fields in Records: offset array in header
  - Records on Pages: Slotted pages w/internal offsets & free space area
- Can be "lazy" about issues such as free space management, exact ordering, etc. – do it in batches
- Files can be unordered (heap), sorted, or kinda sorted (i.e., "clustered") on a *search key.*
  - Tradeoffs are update/maintenance cost vs. speed of accesses via the search key.
  - Files can be clustered (sorted) at most one way.
- Indexes can be used to speed up many kinds of accesses. (i.e., "access paths")

Tree-Structured Indexes: Introduction

- Selections of form field <op> constant
- Equality selections (op is =)
  - Either "tree" or "hash" indexes help here.
- Range selections (op is one of <, >, <=, >=, BETWEEN)
  - "Hash" indexes don't work for these.
- More complex selections (e.g. spatial containment)
  - There are fancier indexes that can do this…

- Tree-structured indexing techniques support both *range selections* and *equality selections*.
- *ISAM*: static structure; early index technology.
- *B+ tree*: dynamic, adjusts gracefully under inserts and deletes.
- ISAM =Indexed Sequential Access Method

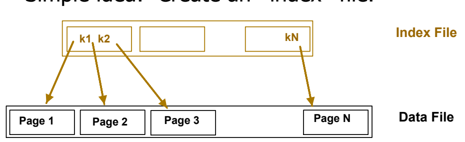
A Note of Caution

- ISAM is an old-fashioned idea
  - B+-trees are usually better, as we'll see
    - Though not *always*
- But, it's a good place to start
  - Simpler than B+-tree, many of the same ideas

- Upshot
  - **Don't** brag about ISAM on your resume
  - **Do** understand ISAM, and tradeoffs with B+-trees

Range Searches

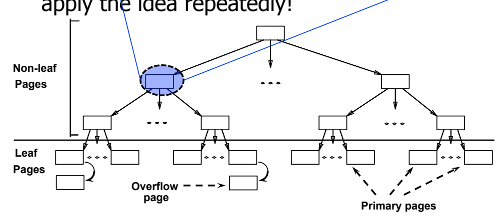
- "*Find all students with gpa > 3.0*"
  - Sorted file? Binary search to find first, scan to find others.
  - Cost of binary search in a database can be quite high. Q: Why?
- Simple idea: Create an `index' file.



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

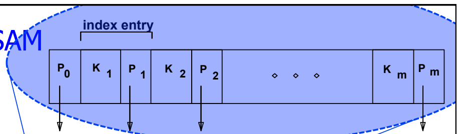
ISAM

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

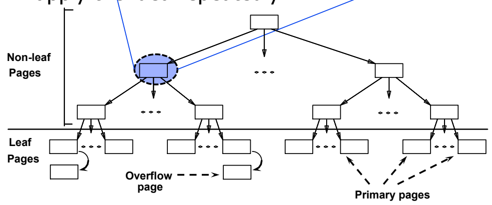


☛ *Leaf pages contain data entries.*

ISAM



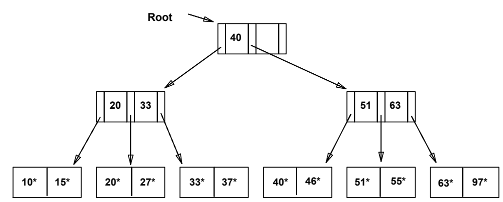
- Index file may still be quite large. But we can apply the idea repeatedly!



☛ *Leaf pages contain data entries.*

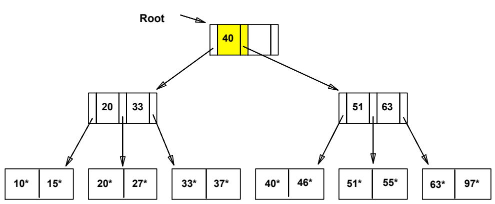
Example ISAM Tree

- ***Index entries***:**<search key value, page id>** **they direct search for data entries *in leaves.***
- **Example where each node can hold 2 entries;**



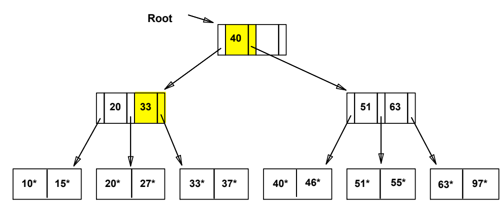
Example ISAM Tree

- ***Index entries***:**<search key value, page id>** **they direct search for data entries *in leaves.***
- **Example where each node can hold 2 entries;**



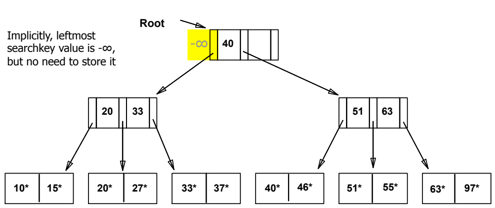
Example ISAM Tree

- ***Index entries***:**<search key value, page id>** **they direct search for data entries *in leaves.***
- **Example where each node can hold 2 entries;**



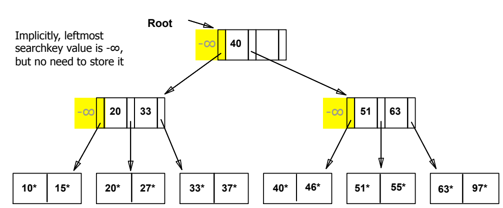
Example ISAM Tree

- ***Index entries***:**<search key value, page id>** **they direct search for data entries *in leaves.***
- **Example where each node can hold 2 entries;**



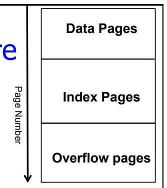
Example ISAM Tree

- ***Index entries***:**<search key value, page id>** **they direct search for data entries *in leaves.***
- **Example where each node can hold 2 entries;**



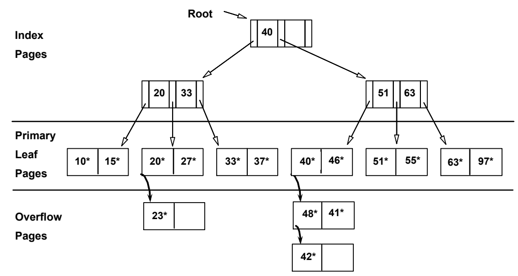
ISAM is a STATIC Structure

- *File creation*:
  - Leaf (data) pages stored in order by search key
  - then index pages
  - then overflow pgs.
- *Search*: Start at root; use key comparisons to go to leaf.
- Cost = $\log_F N$
  - F = # entries/pg (i.e., fanout)
  - N = # leaf pgs
  - no need for `next-leaf-page' pointers. (Why?)
- *Insert*: Find leaf that data entry belongs to, and put it there. **Overflow page** if necessary.
- *Delete*: Seek and destroy! If deleting a tuple empties an overflow page, de-allocate it and remove from linked-list.

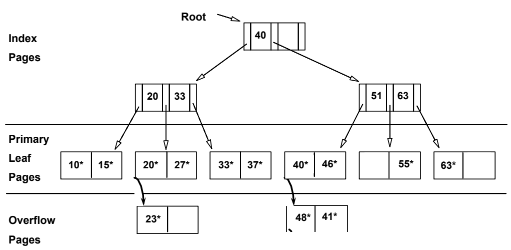


Static tree structure: *inserts/deletes affect only leaf pages.*

Example: Insert 23*, 48*, 41*, 42*



... then Deleting 42*, 51*, 97*



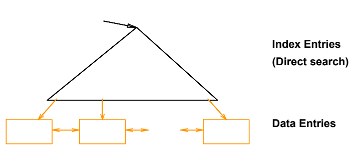
☛ *Note that 51* appears in index levels, but not in leaf!*

ISAM ---- Issues?

- Pros
  - ????
- Cons
  - ????

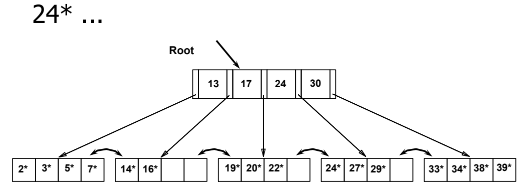
B+ Tree: The Most Widely Used Index

- Insert/delete at $\log_F N$ cost; keep tree *height-balanced*.
  - F = fanout, N = # leaf pages
- Tree is not width-balanced: different nodes have different fanouts
  - BUT: minimum 50% node occupancy (except for root).
  - Each node contains *m entries where* d <= *m* <= 2d entries.
  - "d" is called the *order* of the tree.
- Supports equality and range-searches efficiently.
- As in ISAM, all searches go from root to leaves, but structure is dynamic.



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* ...



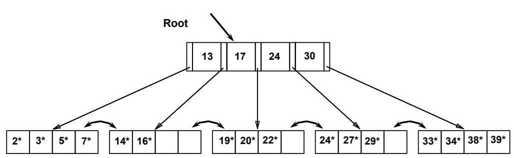
B+ Trees in Practice

- Typical order: 100. Typical fill-factor: 67%.
  - average fanout = 133
- Typical capacities:
  - Height 2: 133^3 = 2,352,637 entries
  - Height 3: 133^4 = 312,900,700 entries
- 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

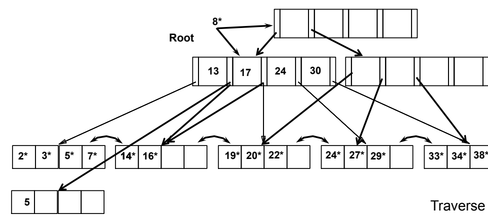
Inserting a Data Entry into a B+ Tree

- Find correct leaf *L.*
- Put data entry onto *L*.
  - If *L* has enough space, *done*!
  - Else, must *split* *L (into L and a new node L2)*
    - Redistribute entries evenly, **copy up** middle key.
    - Insert index entry pointing to *L2* into parent of *L*.
- This can happen recursively
  - To split index node, redistribute entries evenly, but **push up** middle key. (Contrast with leaf splits.)
- Splits "grow" tree; root split increases height.
  - Tree growth: gets *wider* or *one level taller at top.*

Example B+ Tree – Inserting 8*



Animation: Insert 8*



Traverse to leaf

Split Leaf

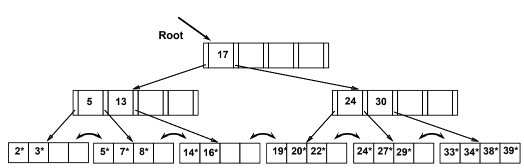
Copy up middle key

Split Parent

New Root

Push up middle key

Final B+ Tree - Inserting 8*

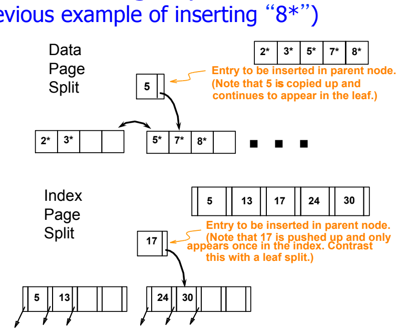


❖ Notice that root was split, leading to increase in height.

❖ In this example, we can avoid split by re-distributing entries; however, this is usually not done in practice.

Data vs. Index Page Split
(from previous example of inserting "8*")

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

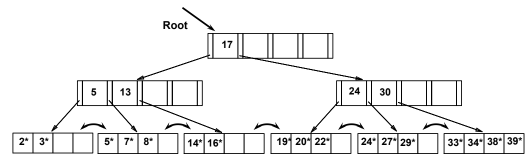


Deleting a Data Entry from a B+ Tree

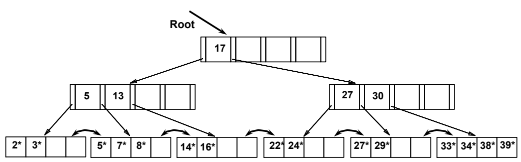
- Start at root, find leaf *L* where entry belongs.
- Remove the entry.
  - If *L* is at least half-full, *done!*
  - If L has only **d-1** entries,
    - Try to re-distribute, borrowing from *sibling (adjacent node with same parent as L).*
    - If re-distribution fails, *merge L* and sibling.
- If merge occurred, must delete entry (pointing to *L* or sibling) from parent of *L*.
- Merge could propagate to root, decreasing height.

In practice, many systems do not worry about ensuring half-full pages. Just let page slowly go empty; if it's truly empty, just delete from tree and leave unbalanced.

Example Tree (including 8*) Delete 19* and 20* ...



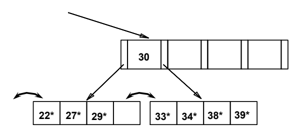
Example Tree (including 8*) Delete 19* and 20* ...

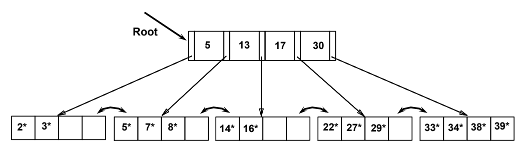


- Deleting 19* is easy.
- Deleting 20* is done with re-distribution. Notice how middle key is *copied up*.

... And Then Deleting 24*

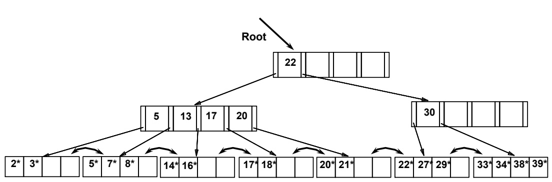
- Must merge.
- Observe `*toss*' of index entry (on right), and `*pull down*' of index entry (below).





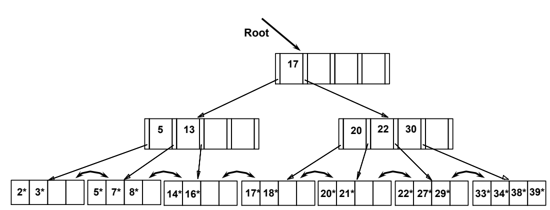
Example of Non-leaf Re-distribution

- 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.

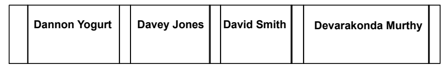


Nice Animation On-Line

- https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html
- One small difference to note:
  - Upon deletion of leftmost value in a node, it updates the parent index entry
  - Incurs unnecessary extra writes

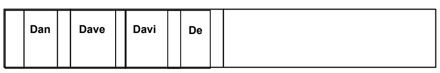
Prefix Key Compression

- Important to increase fan-out. (Why?)
- Key values in index entries just 'direct traffic'; can often compress them.



Prefix Key Compression

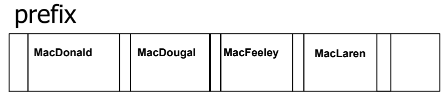
- Important to increase fan-out. (Why?)
- Key values in index entries just 'direct traffic'; can often compress them.



  - Is this correct?
  - Ensure that each index entry is greater than every key value (in any descendant leaf) to its left.
  - In practice we compress upon "copy up" from leaf.

Suffix Key Compression

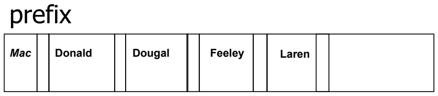
- If many index entries share a common prefix



- Particularly useful for composite keys
  - Why?

Suffix Key Compression

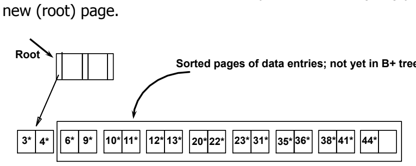
- If many index entries share a common prefix



- Particularly useful for composite keys
  - Why?

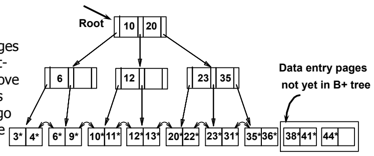
Bulk Loading of a B+ Tree

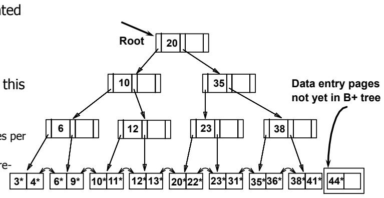
- Given: large collection of records
- Desire: B+ tree on some field
- Bad idea: repeatedly insert records
  - Slow
  - Also poor leaf space utilization. Why?
    - Play with the web animation: try to make up values that keep leaves full!
- *Bulk Loading* can be done much more efficiently.
- *Initialization*: Sort all data entries, insert pointer to first (leaf) 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.
- Exercise: what kind of buffer pool hit rate will this give you for different policies?
  - Q1: how many references per page?
  - Q1: how often are they re-referenced?





Summary of Bulk Loading

- 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**) makes little sense with variable-length entries
- Use a physical criterion in practice (`*at least half-full*').
  - Index pages often hold many more entries than leaf pages.
  - Variable sized records and search keys:
    - different nodes have different numbers of entries.
  - Even with fixed length fields, Alternative (3) gives variable length
- Many real systems are even sloppier than this --- only reclaim space when a page is *completely* empty.

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.
  - Typically, 67% occupancy on average.
  - Usually preferable to ISAM; adjusts to growth gracefully.
  - If data entries are data records, splits can change rids!

Summary (Contd.)

- 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.
- B+ tree widely used because of its versatility.
  - One of the most optimized components of a DBMS.