Tree-Structured Indexes

PS #2 due before midnight tonight
PS #3 should be available late tonight

Midterm is coming!
Next Thursday from 10/1

Introduction

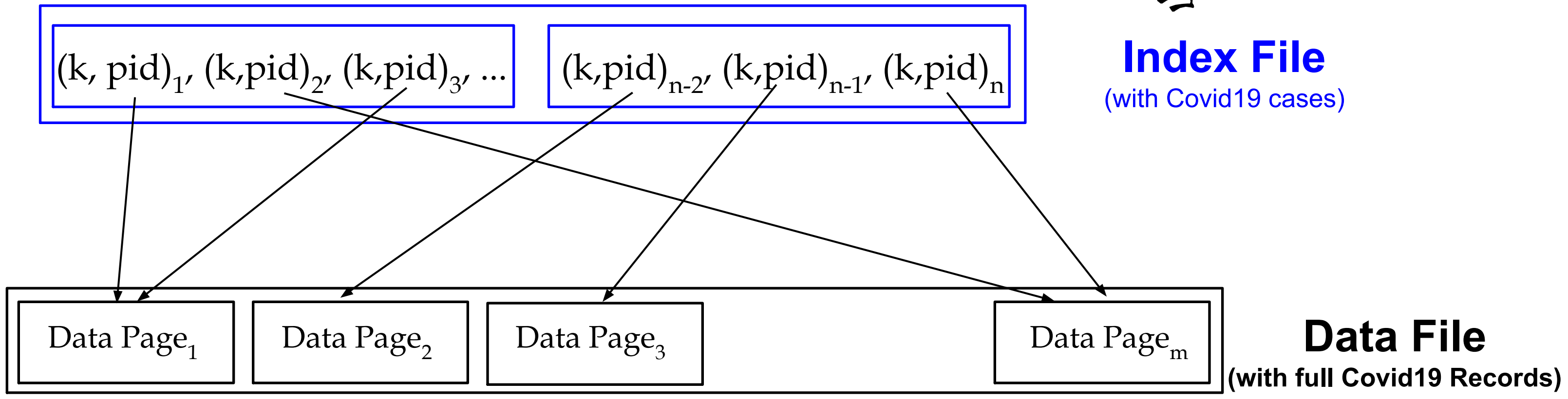
- *As for any index, 3 alternatives for data entries* **k***:
  - index refers to actual data record with key value **k**
  - index refers to list of <**k**, rid> pairs
  - index refers to list of <**k**, [rid list]>
  - Often the record id, "rid", is smply a page id "pid"
- Choice is orthogonal to the *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 counties and dates where 200 or more COVID-19 cases were reported"*
  - If Covid19 records are sorted by case counts, do binary search to find first such player, then scan to find rest.
  - Cost of binary search can be quite high (must read entire page to access one record).
- Simple idea: Create an 'index' file.

Data pages are probably not ordered by this key, but this range might still be an effective filter.

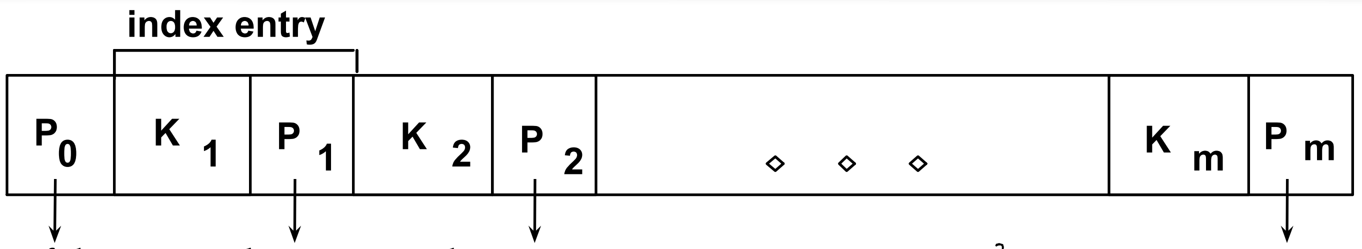
What if the (key, pid) pairs are large and occupy many pages?



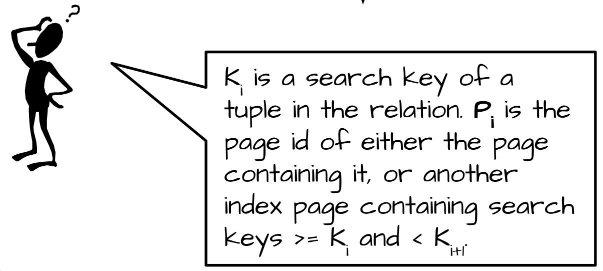
(k, pid)₁, (k,pid)₂, (k,pid)₃, ...
(k,pid)ₙ₋₂, (k,pid)ₙ₋₁, (k,pid)ₙ
Index File
(with Covid19 cases)
Data Page₁
Data Page₂
Data Page₃
Data Pageₘ
Data File
(with full Covid19 Records)

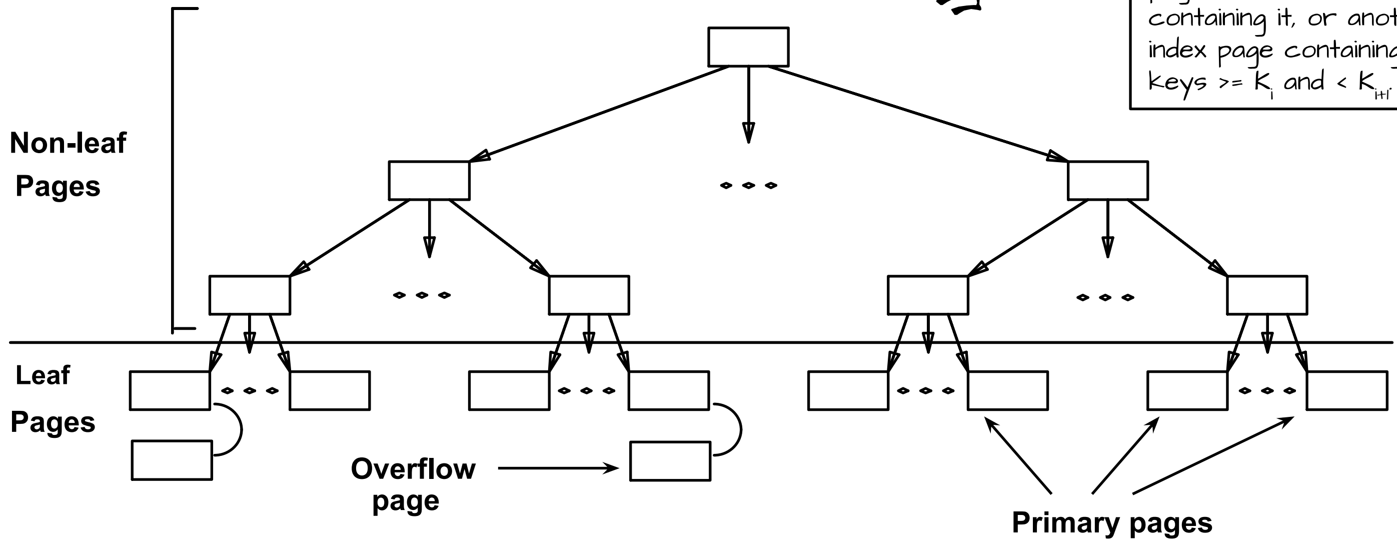
☞ *Can perform binary (or better) search on (smaller) index file!*

ISAM – *Indexed Sequential Access Method*



- Index file may be quite large.
- Can be applied hierarchically!

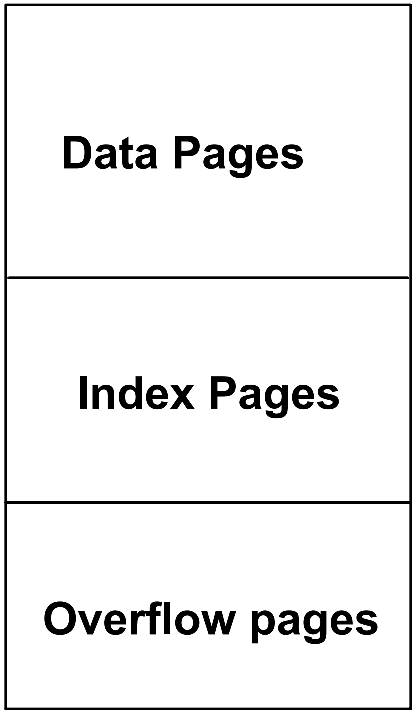

K_i is a search key of a tuple in the relation. P_i is the page id of either the page containing it, or another index page containing search keys >= K_i and < K_{i+1}.



☞ *Leaf pages contain data entries (i.e. actual records or <key, rid> pairs.*

Comments on ISAM

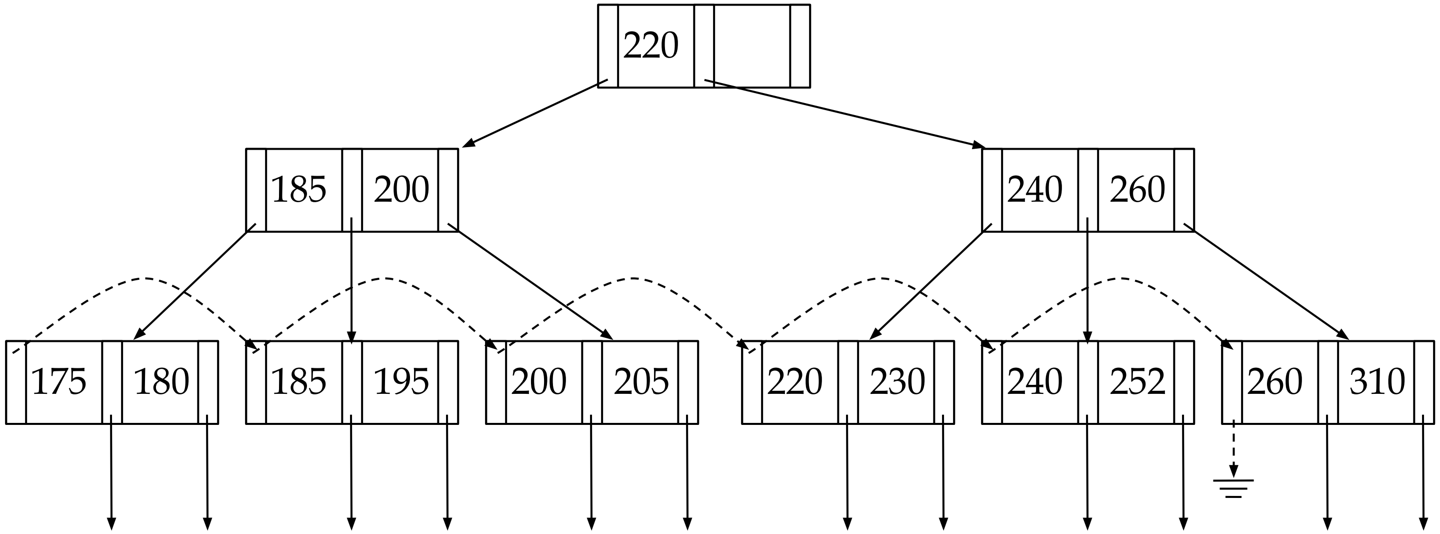
- *File creation*: Leaf index pages of <search key, pid> pairs are created first; then sorted by search key; then higher-level index pages are created as needed until all search keys fit into a single page.
- Index entries: <search key, pid>; they 'steer' the search for *data entries*, which are in leaf pages.
- *Search*: Start at root; use key comparisons to leaf. Cost: $\log_F N$ (page reads) F = # entries/index page, N = # leaf pages
- *Insert*: Find largest leaf search key entry less than or equal to the inserted record's key value, insert the new record on that page if space is available, else allocate an overflow page, put it there, and link it in.
- *Delete*: Use index to find data page with key value, delete the record, if data page is empty and is an overflow page, you can de-allocate it.



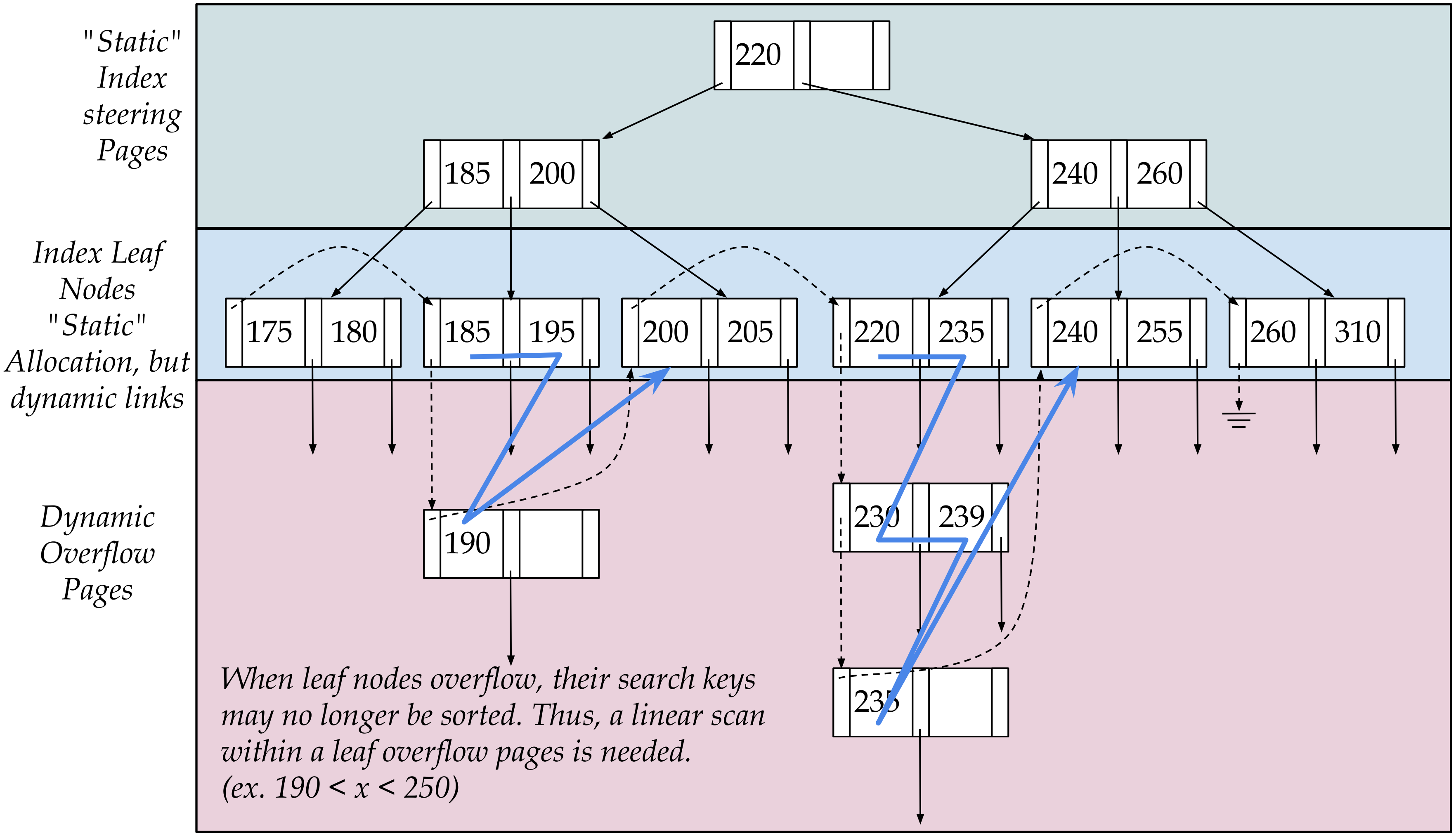
☞ **Static tree structure**: *inserts/deletes affect only data pages.*

Example ISAM Tree

- Trivial example where each index page holds only 2 search keys, and 3 page ids. Notice root, and last page of an index level might not be "full"

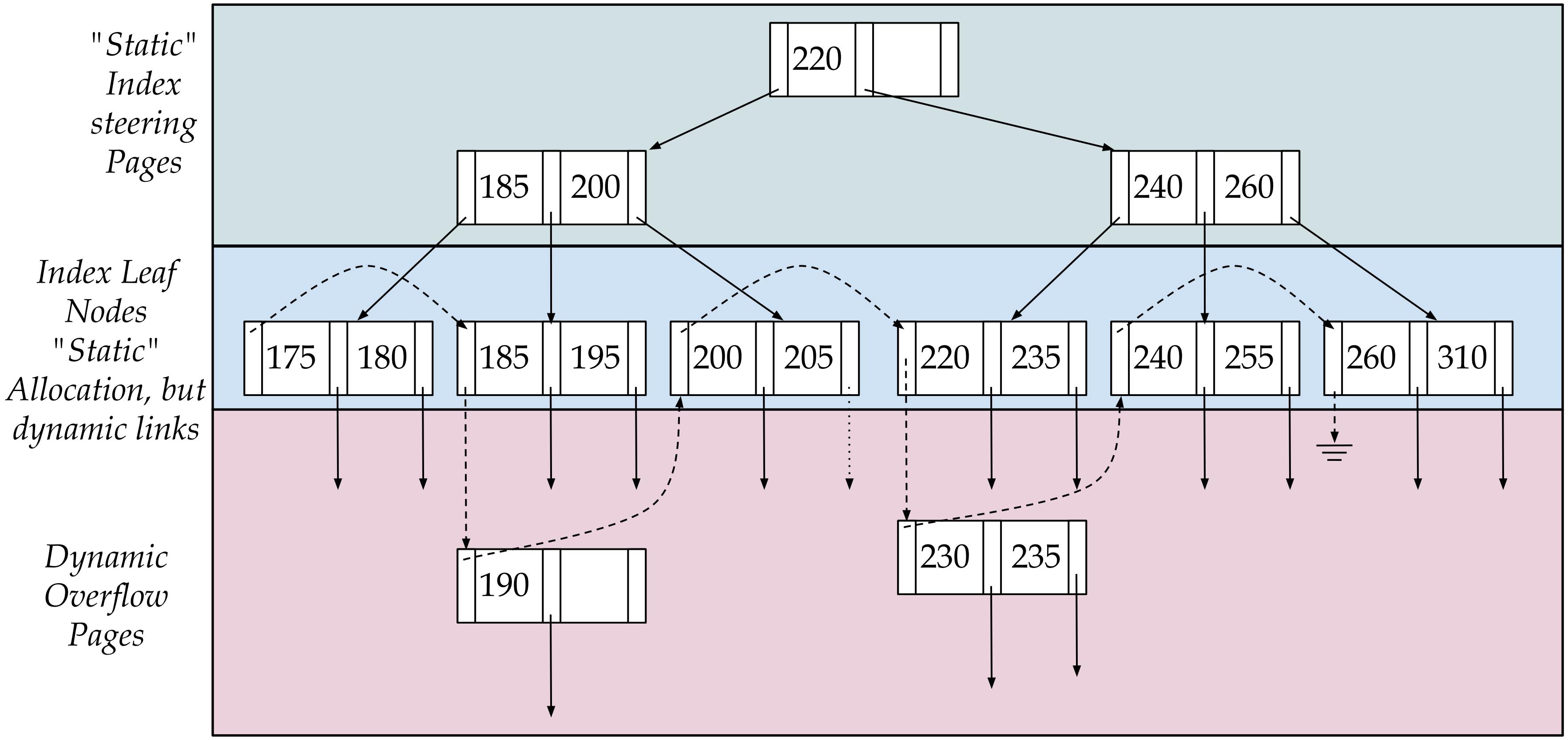


After Inserting 190, 230, 239, 235



When leaf nodes overflow, their search keys may no longer be sorted. Thus, a linear scan within a leaf overflow pages is needed. (ex. 190 < x < 250)

… then delete the records 239, 205



"Static" Index steering Pages
Index Leaf Nodes "Static" Allocation, but dynamic links
Dynamic Overflow Pages
220
185
200
240
260
175
180
185
195
200
205
220
235
240
255
260
310
190
230
235

☞ *Now that 205 appears in index, but not in page!*

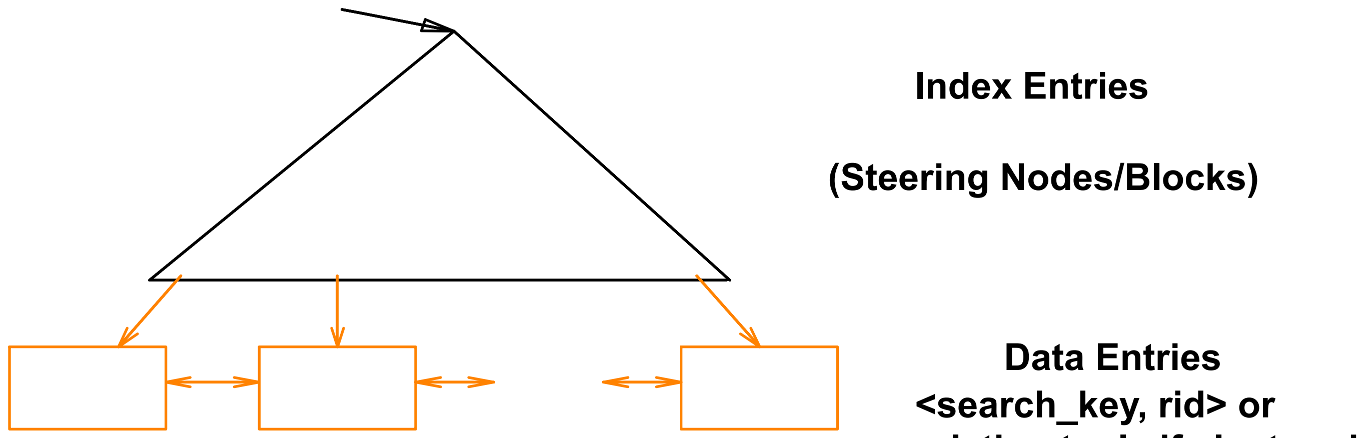
ISAM conclusions and issues

- ISAM indices can be built quickly and and are efficient in terms of space uitilzation
- Generally there are hundreds of keys per page
  - The tree depth is seldom greater than 3 or 4

- Great for a read-mostly tables
- If Inserts, Deletes, and Updates of keys are common
  - ISAM can become stale (many linear leaf scans)
  - An ISAM tree can become unbalanced
- Frequently this requires rebuilding of the index
- By default sqlite is using an ISAM index

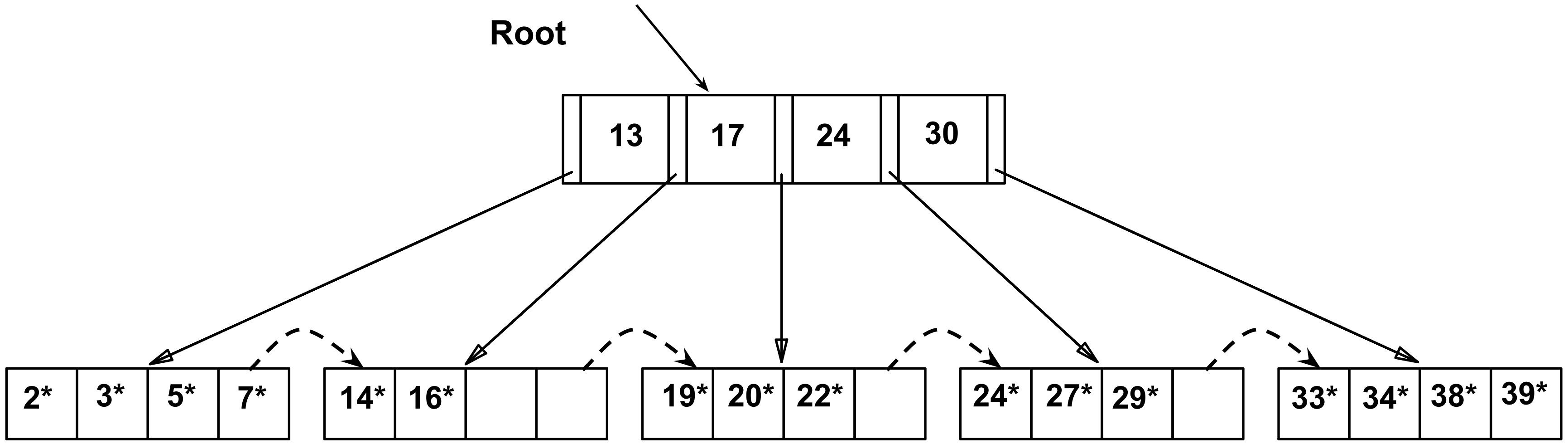
B+ Tree: A Widely Used Tree Index

- Insert/delete at log $_F$ N cost; *maintains a balanced tree*. (F = fanout, N = # leaf pages)
- Minimum 50% node occupancy, with the possible exception of the root. Each internal non-root node contains ½ **d** ≤ *m* ≤ **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).
- We use * to indicate a search key with an associated data-page id
- Search for 5*, 15*, all data entries ≥ 24* ...



☞ *Based on the search for 15*, we* ***know*** *it is not in the tree!*

B+ Trees in Practice

- Typical order: 200, which implies a maximum of 200 children with 199 search keys. (8Kb page, 40 bytes per index entry)
  - average fill-factor: 67%.
  - average fanout = 133
- Typical capacities:
  - Height 4: 133^4 = 312,900,700 records
  - Height 3: 133^3 = 2,352,637 records
- Top levels of B+ tree are often pinned in buffer pool:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 Mbytes

Inserting into a B+ Tree

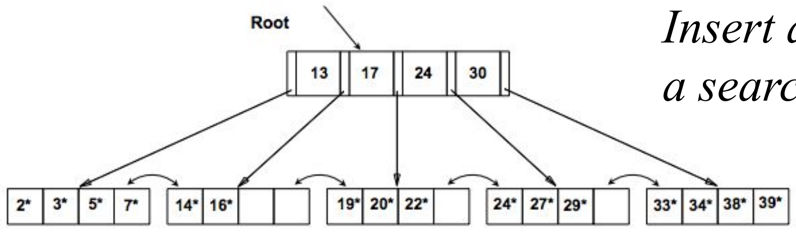
- Find correct leaf *L.*
- Put data entry onto *L*.
  - If *L* has enough space, *done*!
  - Else, must *split* *L (into L and a new node L2)*
    - Allocate new node
    - Redistribute entries evenly
    - ***Copy up*** middle key.
    - Insert index entry pointing to *L2* into parent of *L*.

We maintain the invariant that all steering nodes, besides the root, are at least half full.

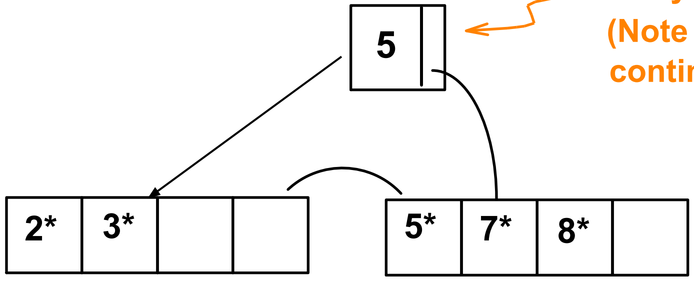
- This happens recursively
  - To split index node, redistribute entries evenly, but ***push up*** middle key (first key in new block). (Contrast with leaf splits.)
- Splits "grow" tree; root split increases height.
  - Tree grows *wider* and *one level taller at top.*

Inserting 8 into Example B+ Tree*

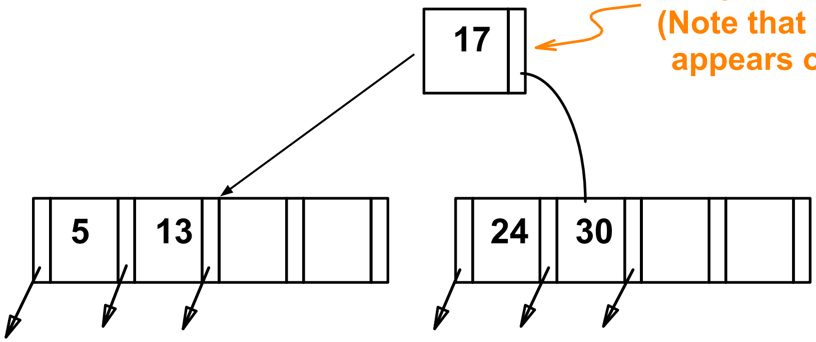
- Observe how minimum occupancy is guaranteed in both leaf and index page splits.
- Note difference between *copy-up* and *push-up;* be sure you understand the reasons for this.



Insert a record with a search key = 8

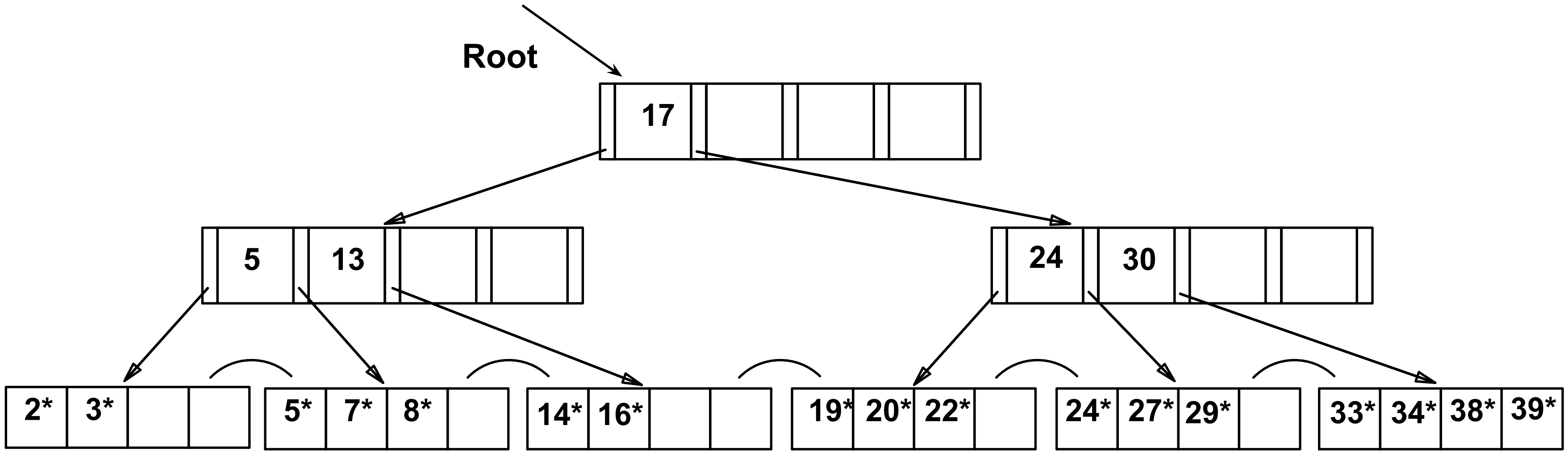


Entry to be inserted in parent node. (Note that 5 is ***copied up*** and continues to appear in the leaf.)

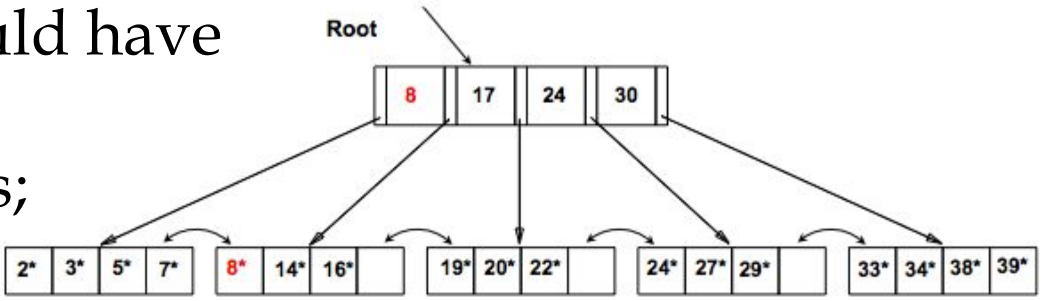


Entry to be inserted in parent node. (Note that 17 is ***pushed up*** and only appears once in the index.)

Example B+ Tree After Inserting 8*



- ❖ Notice that root was split, leading to increase in height.
- ❖ In this example, we could have avoided splitting by *redistributing* key entries; however, this is seldom done in practice.

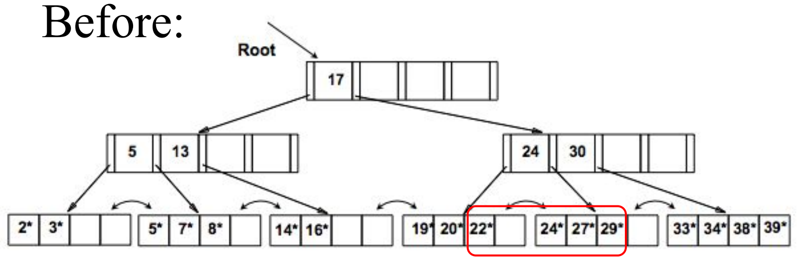


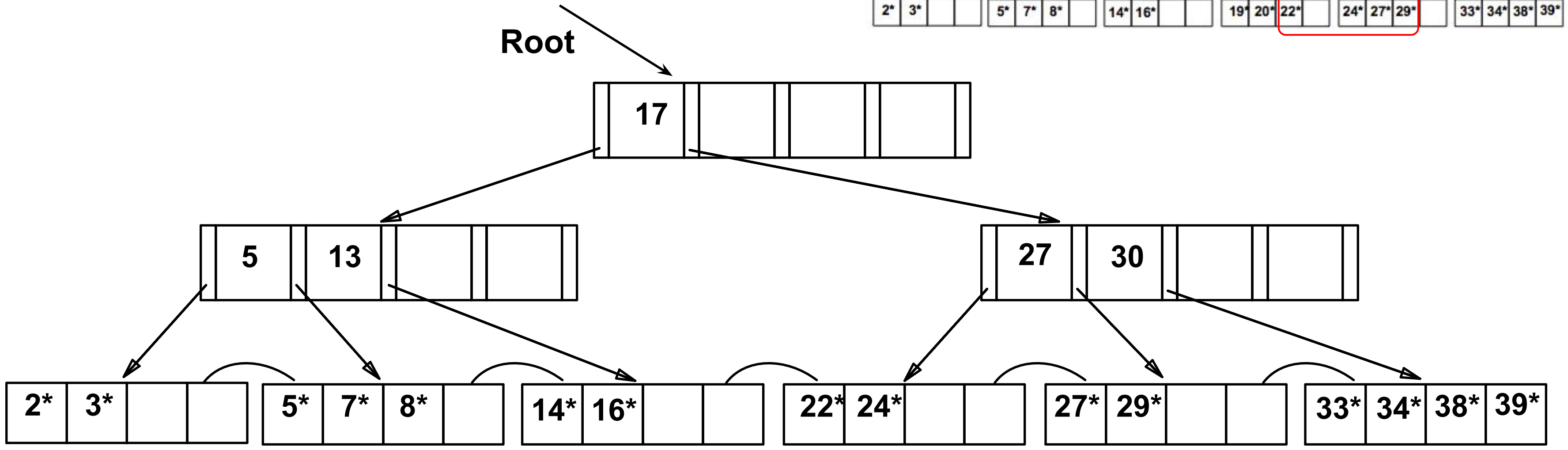
Deleting an entry from a B+ Tree

- Start at root, find leaf page *L* with entry, if it exists.
- Remove the entry.
  - If L is at least half-full, *done!*
  - If L has only ½ **d - 1** entries,
    - Try to re-distribute, borrowing keys from *sibling* *(adjacent node with same parent as L)*.
    - If redistribution 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.

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

Before:

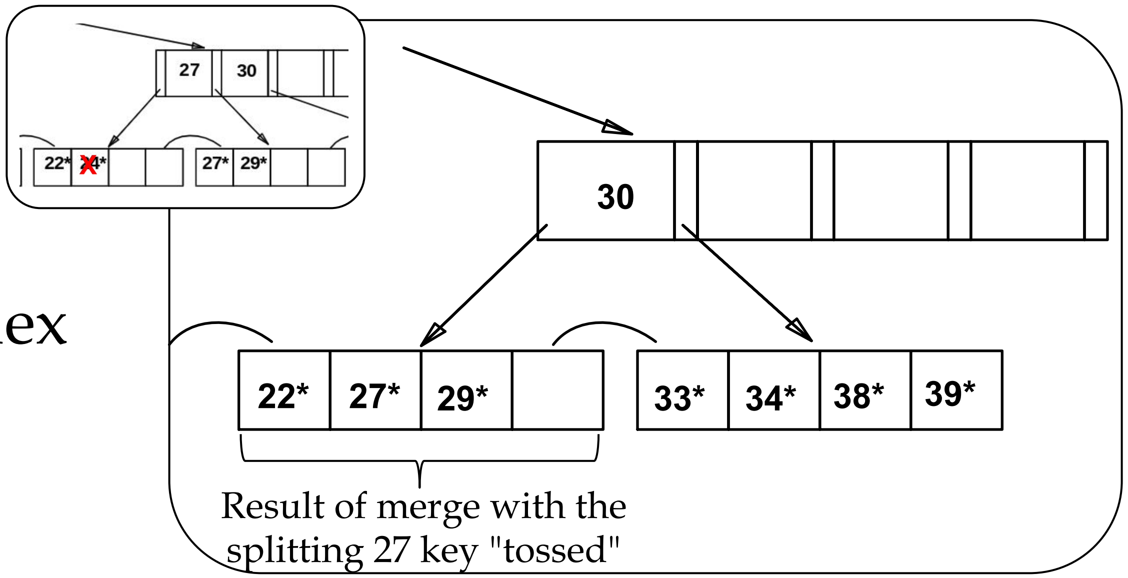




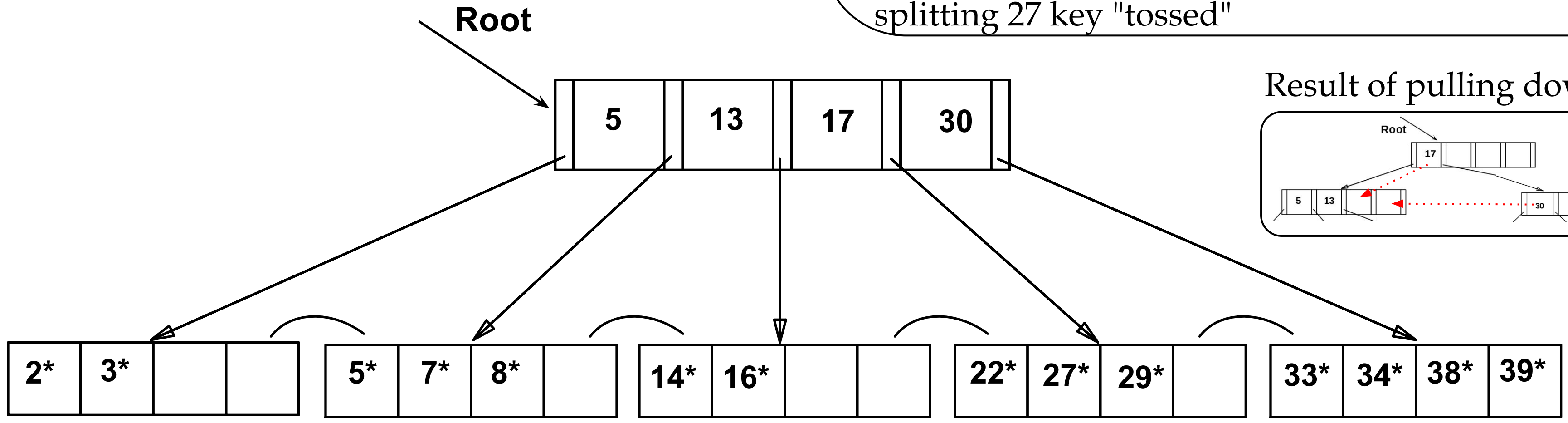
- Deleting 19* was easy (leaf node stayed at least half full).
- Deleting 20* required redistribution (moving serach keys and page ids between blocks). Notice how middle key, 27, is *copied up*, replacing 24.

… and then Delete 24*

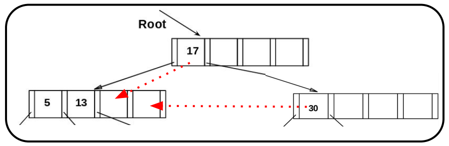
- Must merge.
- Observe '*toss*' of index entry (27)
- And '*pull down*' of index entry from above (17).
- Tree-height shrinks



Result of merge with the splitting 27 key "tossed"



Result of pulling down "17"

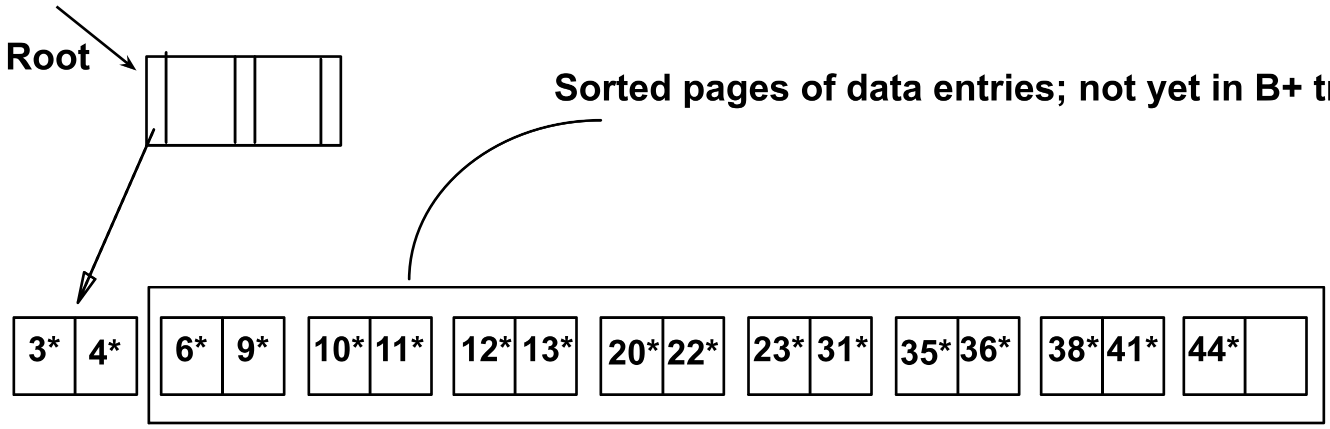


Prefix Key Compression

- Important to increase fan-out.
- Common with composite search keys, and strings
- 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*) Why?
    - 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.

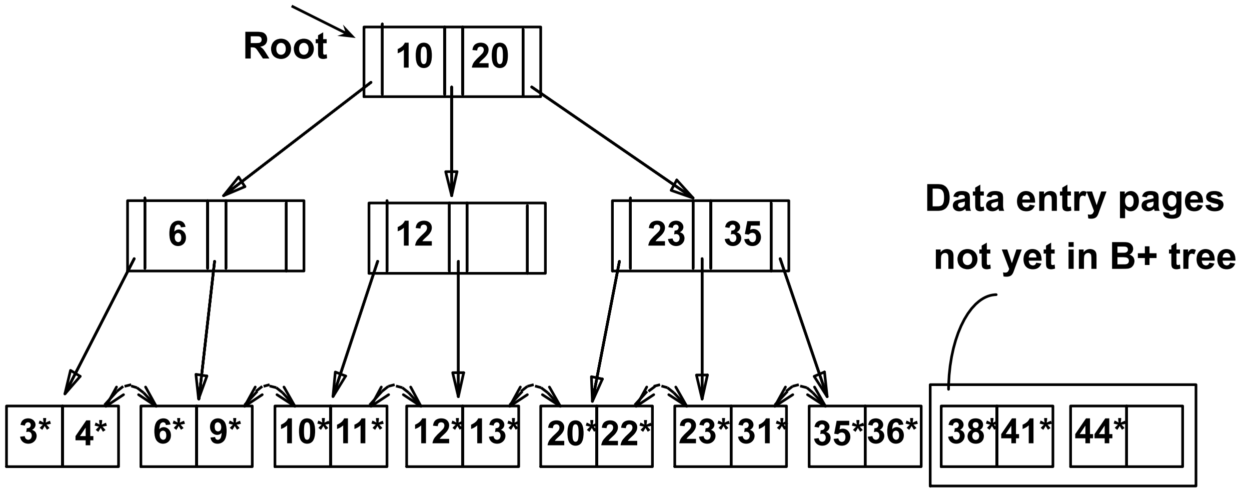
Creating a B+ Tree index

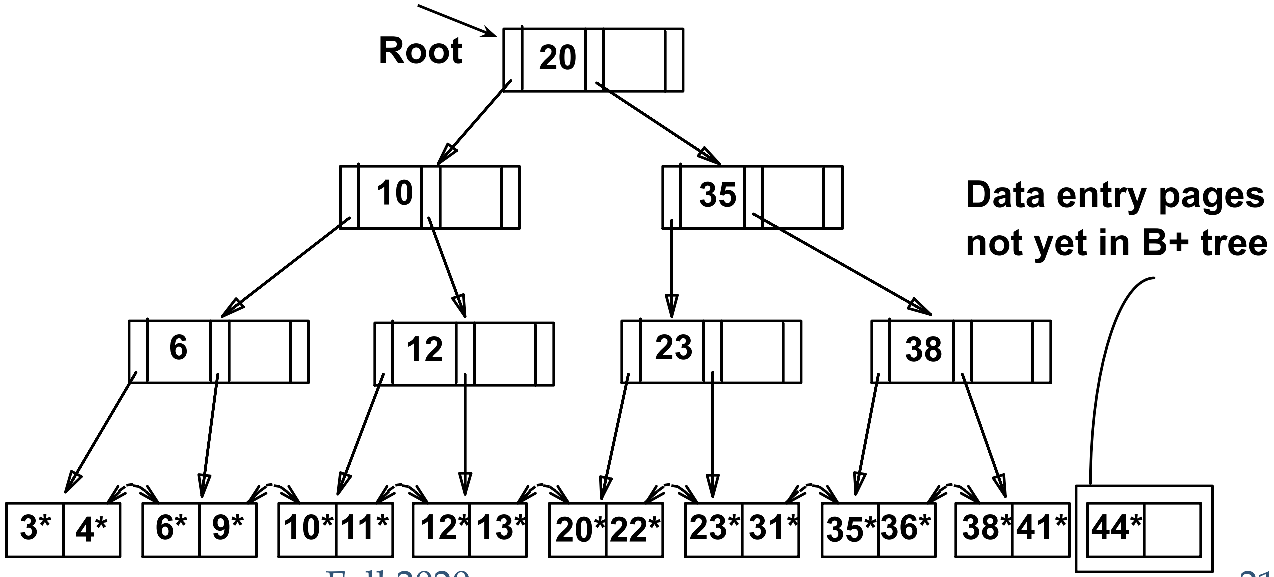
- If we have a large collection of records, and we want to create a B+ tree on some field, doing so by repeatedly inserting records is very slow.
- Index creation is usually done via *Bulk Loading* which can be done efficiently.
- *Initialization*: Sort all data entries, insert pointer to first (leaf) 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, especially if one considers locking!





Summary of Bulk Loading

- Option 1: multiple inserts.
  - Slow.
  - Does not give sequential storage of leaves.
- Option 2: *Bulk Loading*
  - 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.

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.

Next Time

Hash Indices