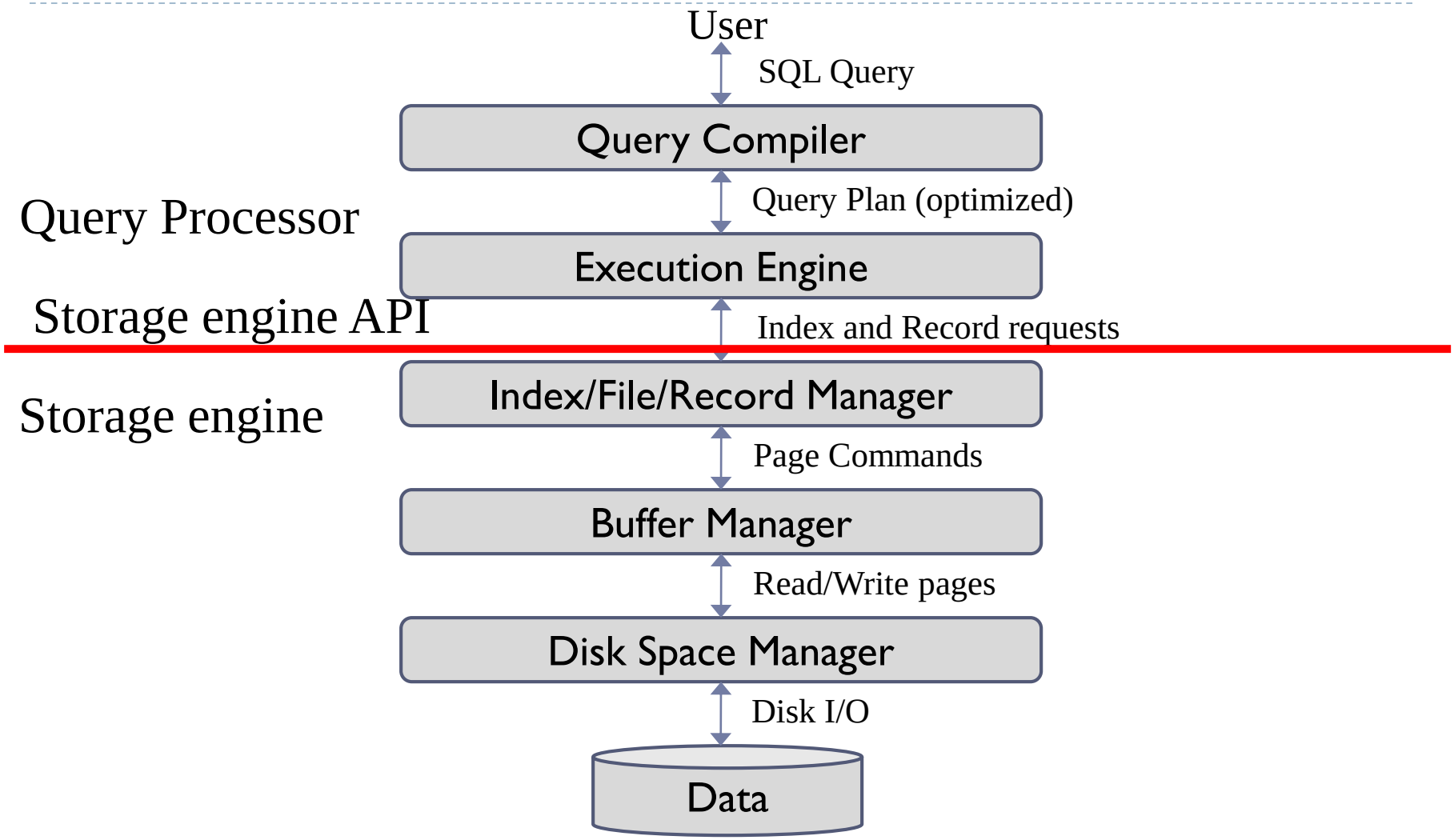
I/O Cost Model, Tree Indexes

CS634
Lecture 5, Feb 10, 2016

Slides based on "Database Management Systems" 3rd ed, Ramakrishnan and Gehrke

Storage Engine vs. QP (Query Processor)



A first course in database systems, 3rd ed, Ullman and Widom

Cost Model and Performance Analysis

- Focus on I/O cost (bulk of the processing cost)
  - **B** The number of data pages in file
  - **R** Number of records per page
  - **D** Average time to read or write disk page
  - **F** Fanout of tree index

- Operations to compare:
  - Scan
  - Equality search
  - Range selection
  - Insert a record
  - Delete a record

Storage Engine API

- Example: The mysql storage engine API has calls to "ha_open" a table, "ha_init_index" to specify the index to use if any, "index_read" to fetch a row by key using the current table and the current index.

- In general, the storage engine can work with one file of records, or two files, one of "data records" and one that is some kind of index for the first file. It can't process more files at once. Two index files are also possible, though not covered in this chapter.

- It's the job of the QP to break down the needed work into the small steps that the storage engine can do.

Compared File Organizations

- Heap file
- Sorted file
- Clustered B+ tree file
  - AKA clustered file
- Heap file with unclustered B+ tree index (2 files involved)
- Heap file with unclustered hash index (2 files involved)

Cost Model: Heap Files (unsorted)

- Given:
  - B The number of data pages in file
  - R Number of records per page
  - D Average time to read or write disk page
- Scan B data pages, each takes read time D
  - So i/o Cost = B*D
- Equality Search, for unique key: i/o Cost = 0.5*B*D
- Range Search: B*D
- Insert: read, write last page: 2*D
- Delete, given key: equality search, then rewrite page
- Delete, given RID: 2*D
- We never used R!
- One problem: D depends on how much seeking is needed

Size of D, time for one page access

- If pages are laid out sequentially on a track, the disk can pull them off in one rotation. D may be .1 ms. or less.
- If pages are randomly situated on the disk, the seek time dominates pages smaller than 1MB in size. D is about 3-5 ms.
- A Common page size is 8KB.
  - Our Oracle DB: 8KB pages
  - Our mysql DB: 16KB pages
- For this chapter, we paste over the difference and pretend D is constant, say 1 ms. for easy calculation.

Tree-Index Assumptions

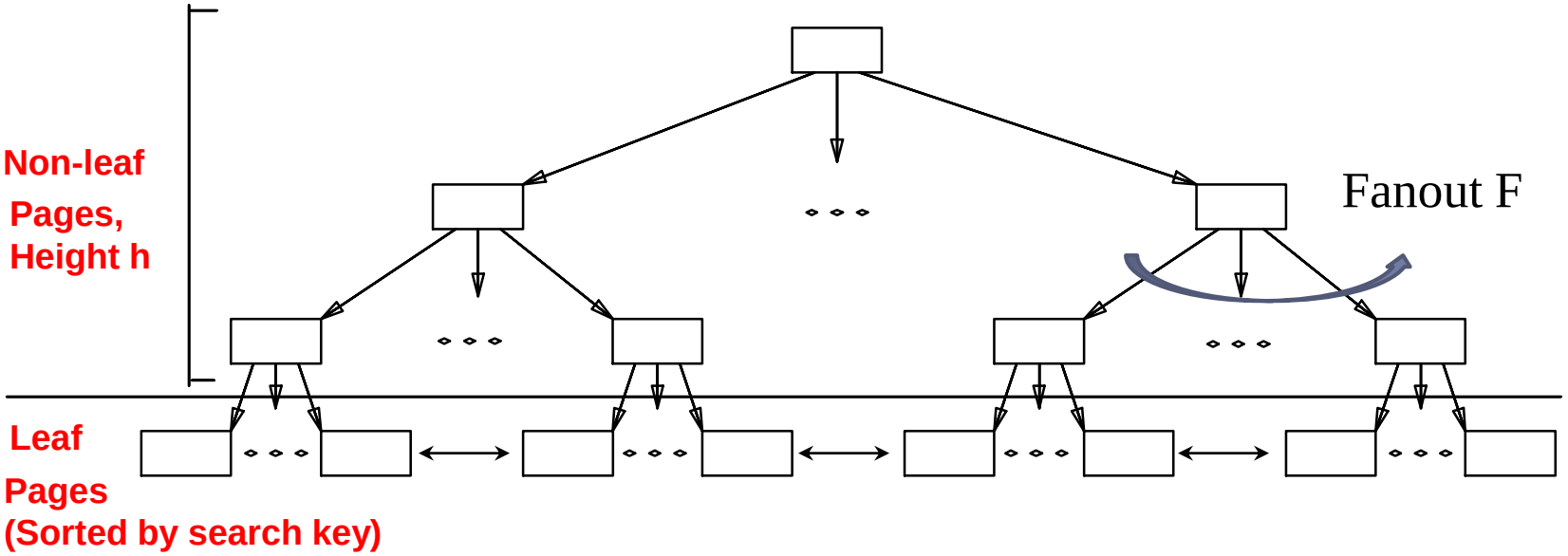
- Tree Indexes:
  - Alternative (1) data entry size = size of record
  - Alternative (2): data entry size = 10% size of record
  - 67% occupancy (this is typical) Note 1/.67 = 1.5
    - File size = 1.5 data size
- Scans:
  - Leaf levels of a tree-index are chained.
  - Index data-entries plus actual file scanned for unclustered indexes (unrealistic, will revisit)
- Range searches:
  - We use tree indexes to restrict the set of data records fetched, but ignore hash indexes for now

Basic Analysis for Tree Indexes

B = # pages to hold row data compactly ("data size")

- Alternative (1) data entry size = size of record
- Alternative (2) data entry size = 10% size of record
  - This is an assumption for simplicity of analysis.
- 67% occupancy, Note 1/.67 = 1.5, so 50% expansion on disk to allow 33% space on pages for new rows.
  - (Actually, follows from B-Tree algorithms)
  - File size = 1.5 data size,
  - Alternative (1): NLeafPgs = 1.5*B
  - Alt. (2): NDataPgs = B, NLeafPgs = .10*1.5*B = .15*B

Tree Index Measurements



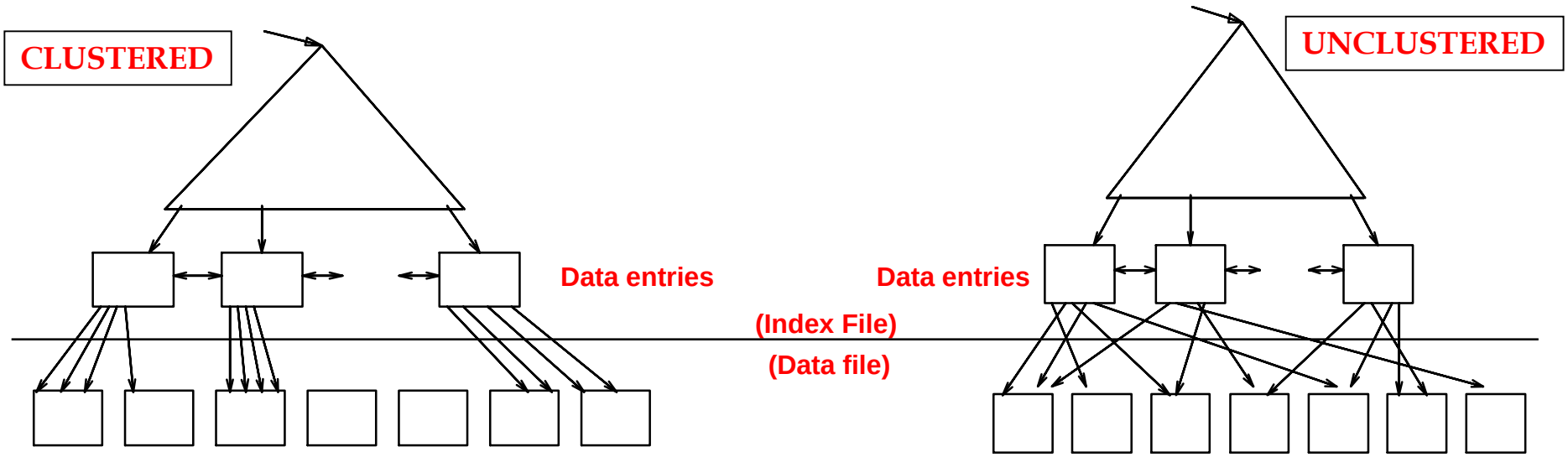
NLeafPgs ~ F^h

So tree height h ~ $\log_F$(NLeafPgs) = $\log_F$(1.5*B), alt (1)

h ~ $\log_F$(NLeafPgs) = $\log_F$(.15*B), alt (2)

Search down tree takes time = h*D, assuming nothing in buffer

Clustered vs. Unclustered Index



- To build clustered index, first sort the heap file, leaving some free space on each page for future inserts
- Overflow pages may be needed for inserts
  - Hence order of data records is close to the sort order
- This is an **Alternative 2** clustered index. For Alternative 1, the rows are in the data entries, so everything is in one FILE.

Cost of Operations (Page 291)

B=#data pgs, R=#recs/pg, D=disk time, F=fanout

	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap	**BD**	**0.5BD**	**BD**	**2D**	**Search +D**
(2) Sorted	**BD**	**Dlog $_2$B**	**D(log $_2$ B + # pgs with match recs)**	**Search + BD**	**Search +BD**
(3) Clustered	**1.5BD**	**Dlog $_F$ 1.5B**	**D(log $_F$ 1.5B + # pgs w. match recs)**	**Search + D**	**Search +D**
(4) Unclust. Tree index	**BD(R+0.15)**	**D(1 + log $_F$ 0.15B)**	**D(log $_F$ 0.15B + # pgs w. match recs)**	**Search + 2D**	**Search + 2D**
(5) Unclust. Hash index	**BD(R+0.125)**	**2D**	**BD**	**Search + 2D**	**Search + 2D**

Scan of Unclustered tree index

- Fig. 8.4 says i/o cost = BD(R+0.15)
- But this means actively using the index and chasing each data entry into the heap.
- No serious database does this.
- Instead, the database scans the underlying heap, at cost BD.
- Similarly for the unclustered hash index

Tree Indexes

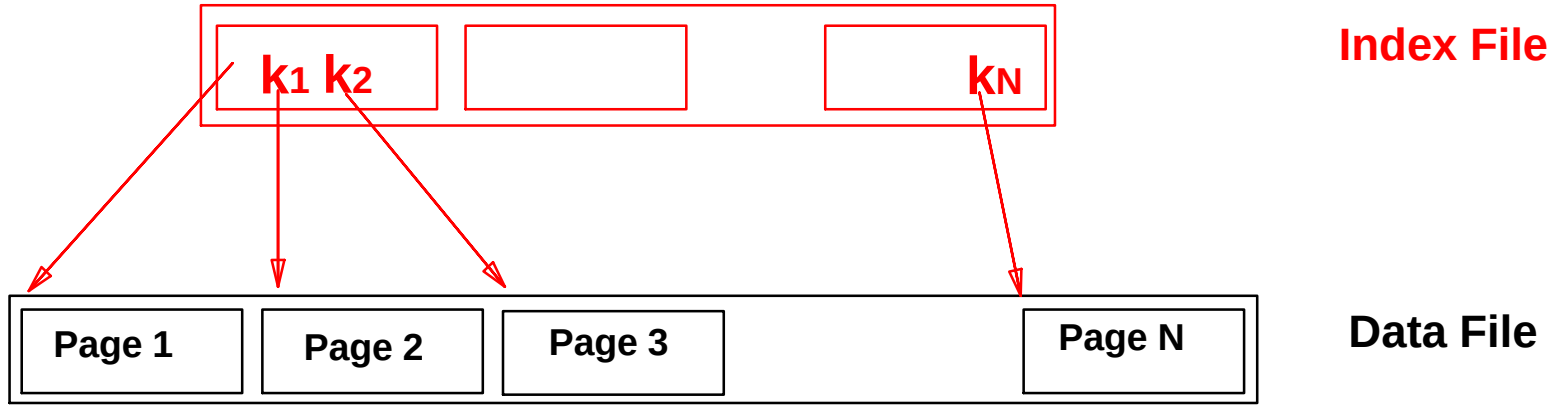
- Tree-structured indexing techniques support efficiently
  - *range searches*
  - *equality searches*

- *ISAM*:
  - Indexed Sequential Access Method, developed by IBM long ago
  - Static tree structure
  - ISAM has another meaning: An API allowing indexed lookup by key and next-ing through successive rows/records after the first lookup.

- *B+ tree*:
  - Dynamic structure
  - Adjusts gracefully under inserts and deletes

Tree Indexes Intuition

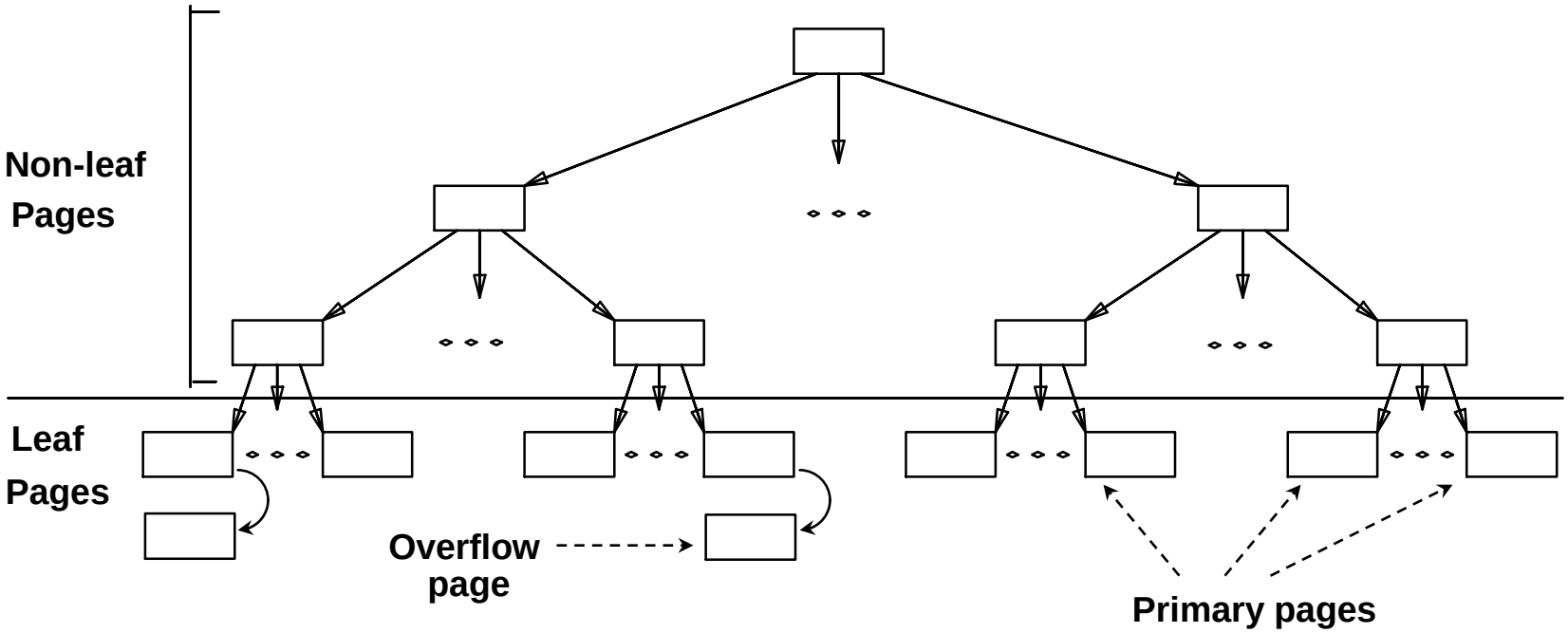
- How to answer efficiently range query on *k*?
  - Option: store data in sorted file on *k* and do binary search
  - Find first matching record, then scan to find others
  - But cost of binary search is high (not to mention inserts)
- Simple idea: Create an **index** file
  - First key value in each data page placed in an index page



Perform binary search on (smaller) index file!

ISAM

- Index file may still be quite large
  - Apply the idea repeatedly on previous index level!



Leaf pages contain data entries

Building an ISAM

- First, create leaf level (data entries)
  - Allocate index leaf pages sequential on disk
  - Alternative 1 natural choice for ISAM (data records inside data entries) – this way data are also sequential on disk
  - Sequential allocation increases performance of range queries
- Next, create internal (index) nodes
  - These will never change after initialization
  - Index entries: <search key value, page id>
  - direct search for *data entries*, which are in leaf pages
- Finally, allocate overflow pages
  - If needed, when insertions are performed

ISAM Operations

- Search
  - Start at root, use keys to guide search towards leaf nodes
  - Cost is $\log_F N$
  - F = # entries/index node page, N = # leaf pages in tree
- Insert
  - Find leaf data entry where records belongs
  - If no space, create an overflow page
- Delete
  - Find and remove from leaf
  - If empty overflow page, de-allocate

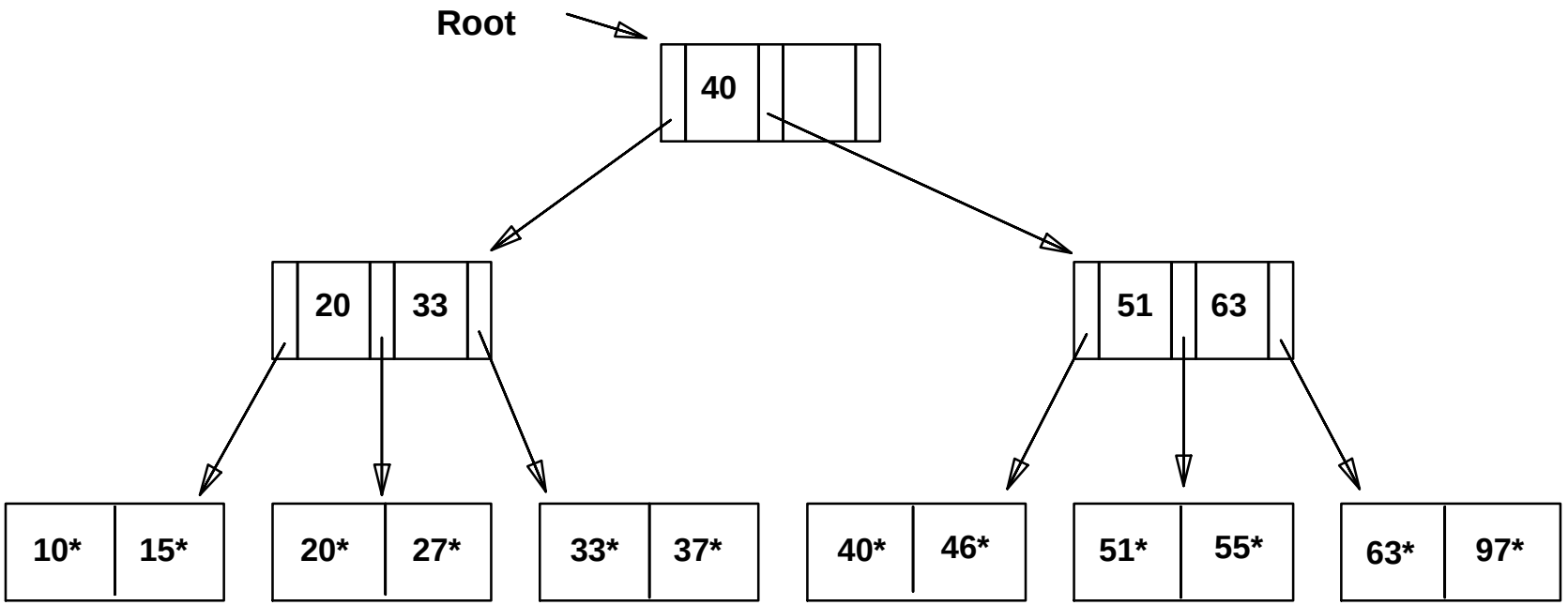
Static tree structure: *inserts/deletes affect only leaf pages!*

Does not require locking the internal nodes!

- *But not available as an access method in any DB, even DB2.*

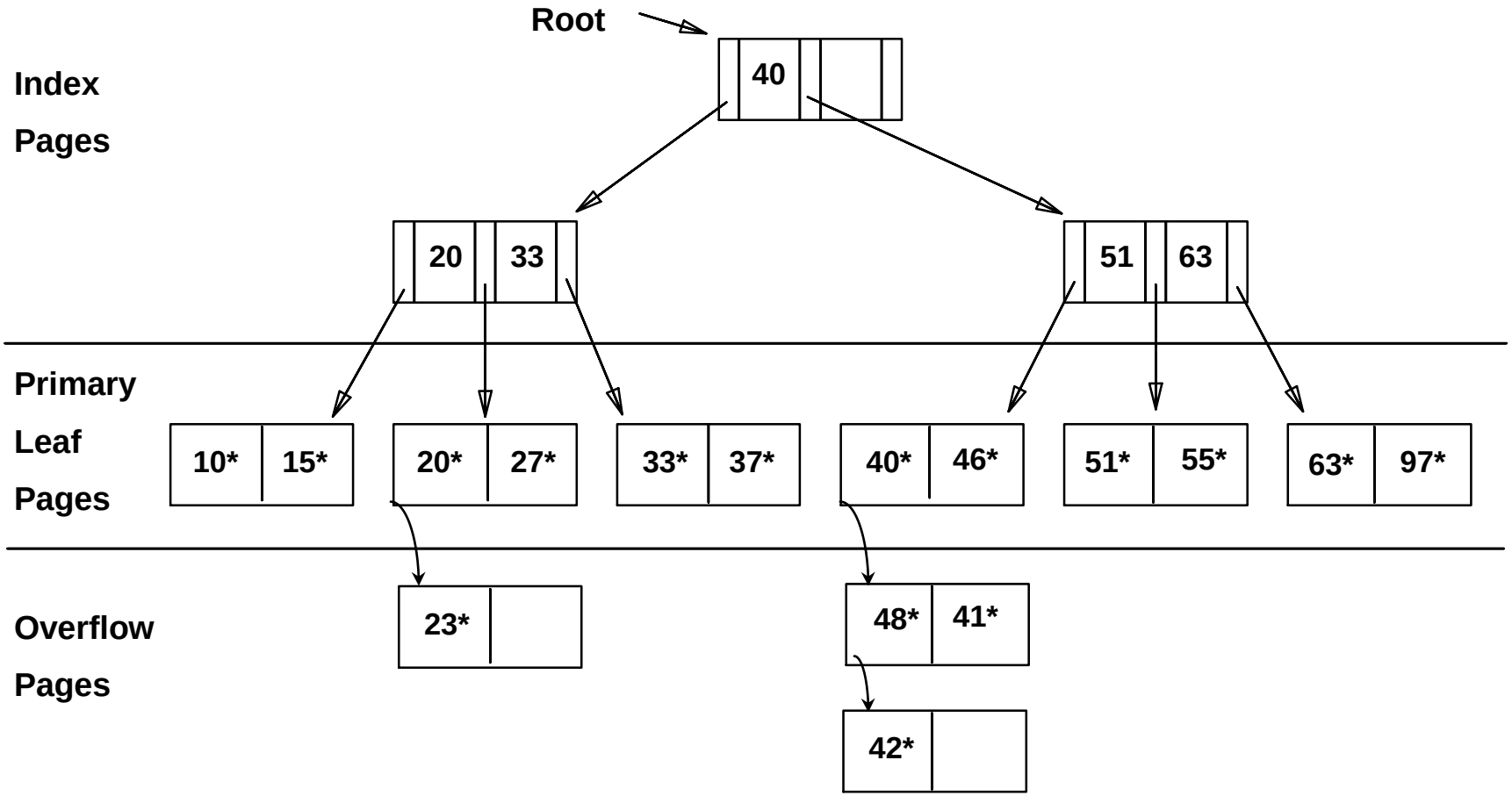
Example ISAM Tree

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

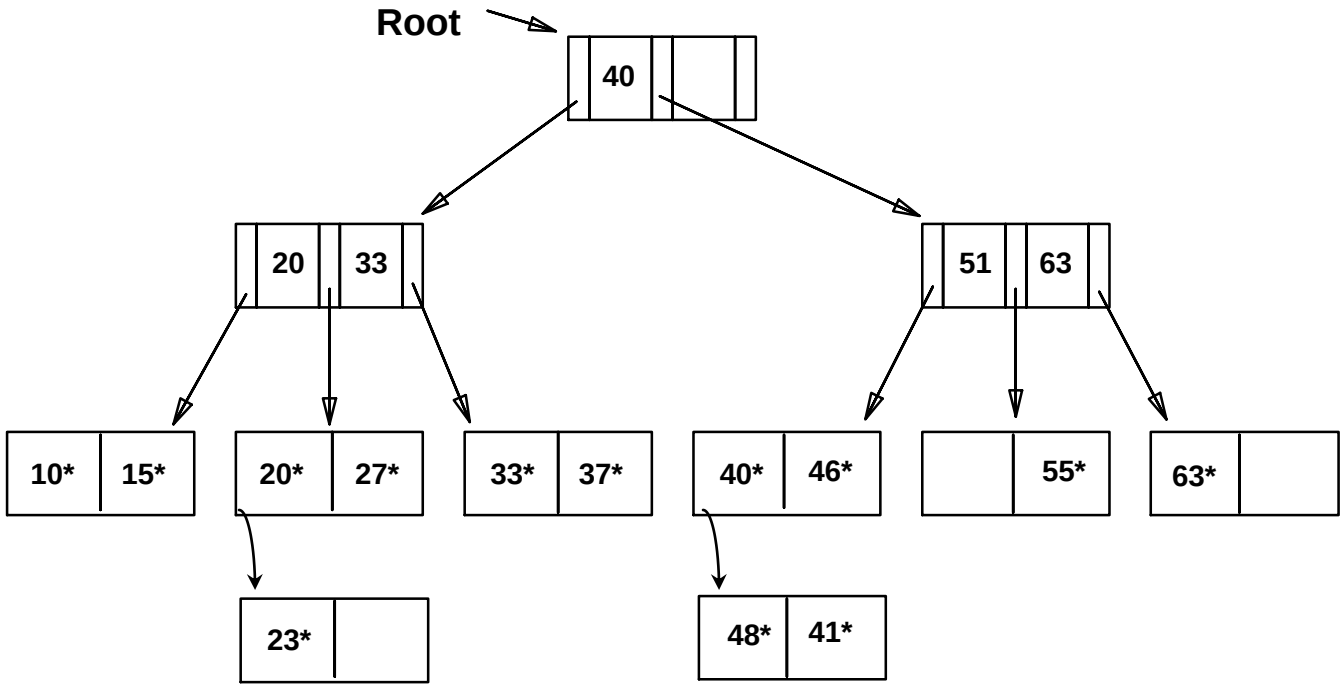


These "primary" leaf pages are sequential on disk!

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



Then Deleting 42*, 51*, 97*



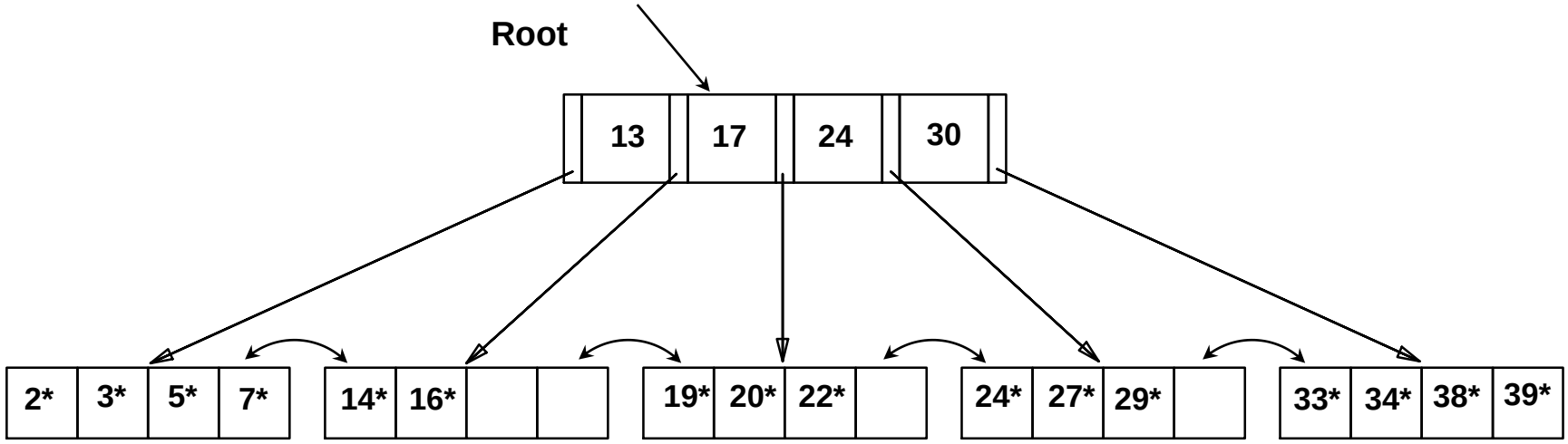
Note that 51 appears in index levels, but not in leaf!*

B+ Tree

- Most Widely Used Index
- Dynamic data structure (as opposed to ISAM)
- Tree is *height-balanced*
  - Height is $\log_F N$ (F = fanout, N = # leaf pages)
  - But fanout isn't really a constant…
- Minimum 50% occupancy constraint
  - Each node (except root) contains **d** <= m <= 2**d** entries
  - Parameter **d** is called the *order* of the tree (min fanout)
- Search just like in ISAM
  - But insert/delete more complex due to occupancy constraint
  - Insert/delete may trigger re-structuring at all levels of tree

B+ Tree Example

- Search begins at root, key comparisons direct it to a leaf

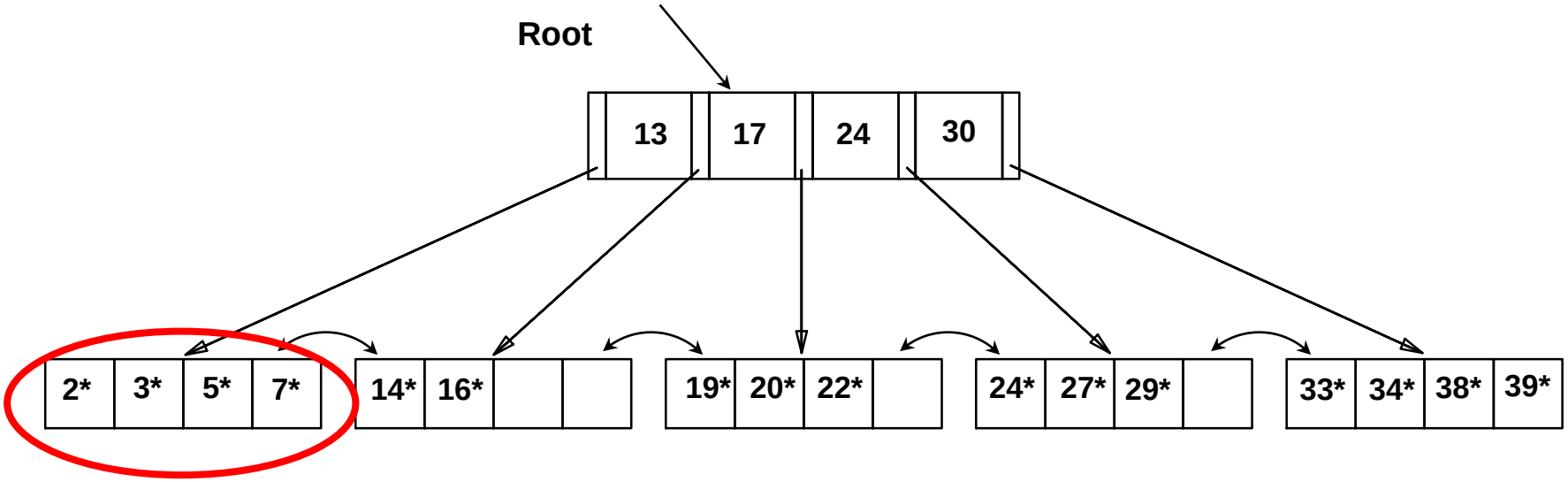


Based on the search for 15, we know it is not in the tree!*

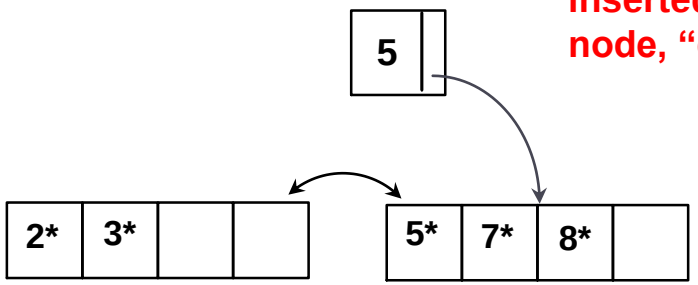
B+ Trees in Practice

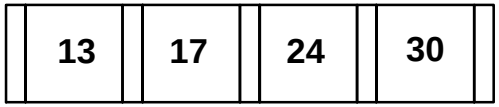
- Recall rule d <= m <= 2d,
  - Here d = order, m = # entries/page = fanout
- Typical order: 100 (min fanout)
  - So max fanout = 2*100 = 200, for full page
- Typical fill-factor: 67%
  - Results on average fanout F = .67*200 = 133
- Typical capacities:
  - Height 4: 133^4 = 312,900,700 leaf pages
  - Height 3: 133^3 = 2,352,637 leaf pages
  - Each leaf page corresponds to .67 page of row data (alt (1)) or 6.7 pages of row data (alt(2) using 10% size assumption)
- Can often hold top levels in buffer pool: assuming 8KB pages:
  - Level 1 = 1 page = 8 KB
  - Level 2 = 133 pages = 1 MB
  - Level 3 = 17,689 pages = 133 MB
  - Level 4 = 2,352,637 pages = 18 GB (probably not in pool)

B+ Tree Example: Insert 8* (d=2)



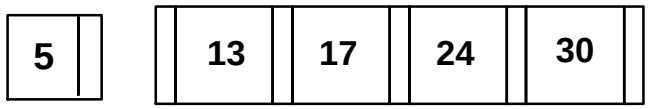
Entry to be inserted in parent node, “copy-up”

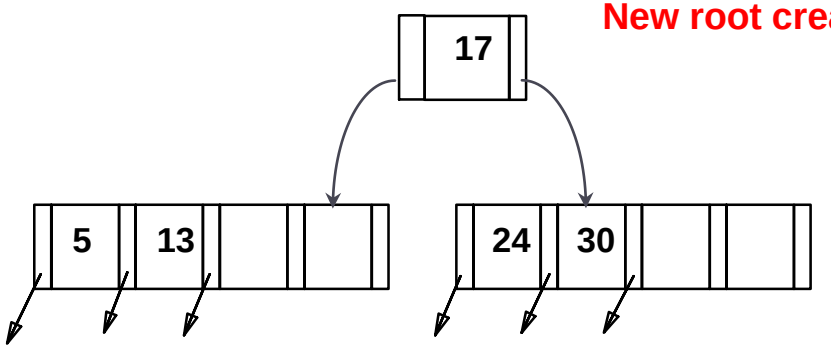




But root is full!

B+ Tree Example: Insert 8* (d=2)

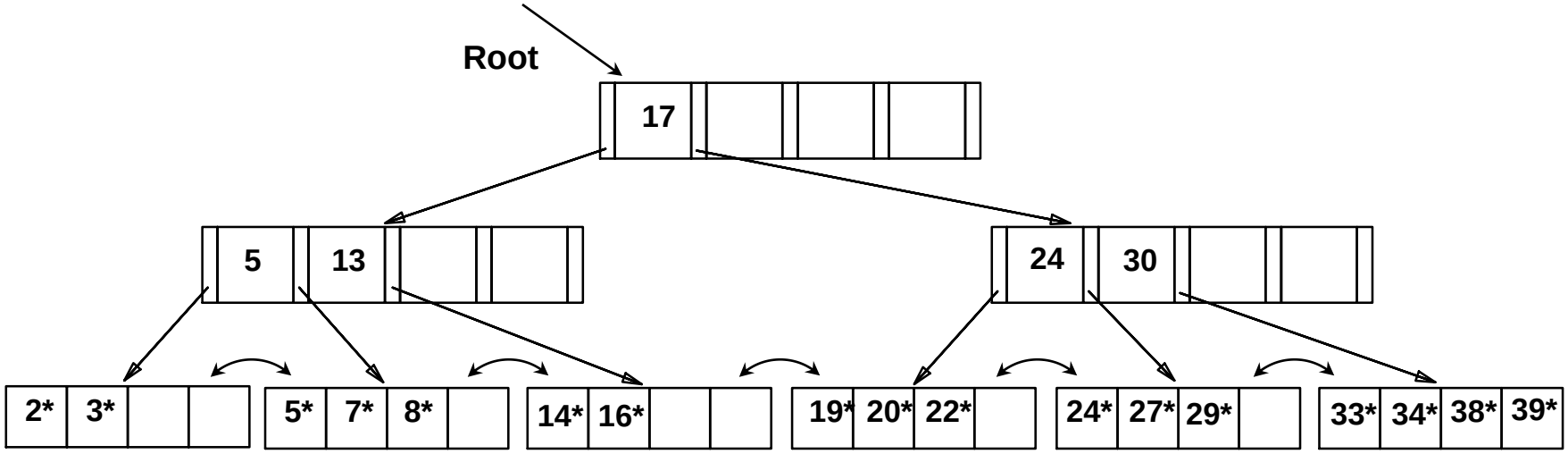




New root created!

Note that 17 is “pushed up”; contrast this with “copy-up” for leaf nodes
Push up: deletes key from lower level and inserts it in upper level
Copy up: key is left at lower level

Example B+ Tree After Inserting 8*



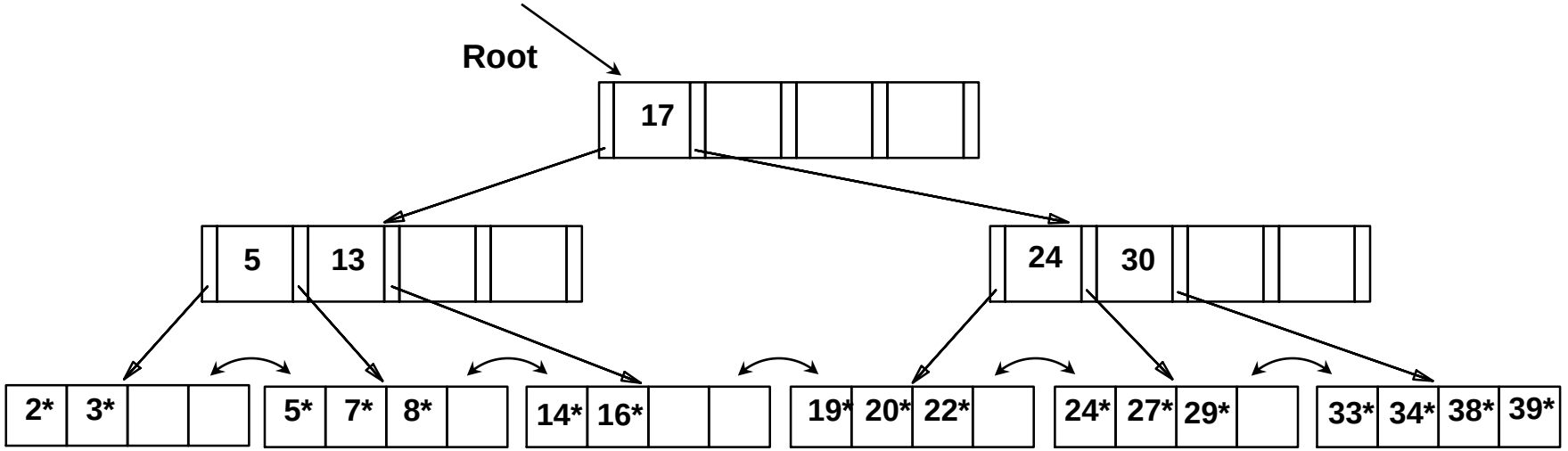
Root was split, leading to increase in height

We can avoid split by re-distributing entries, but this is usually not done in practice for insertions

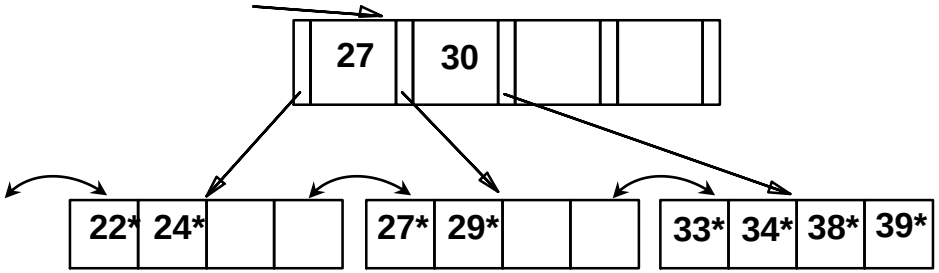
B+ Tree Insertion Algorithm

- Find correct leaf *L*
- Place data entry in *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
- Splits "grow" tree; root split increases height.
  - Tree growth: gets wider or one level taller at top

B+ Tree Example: Delete 19*, then 20*



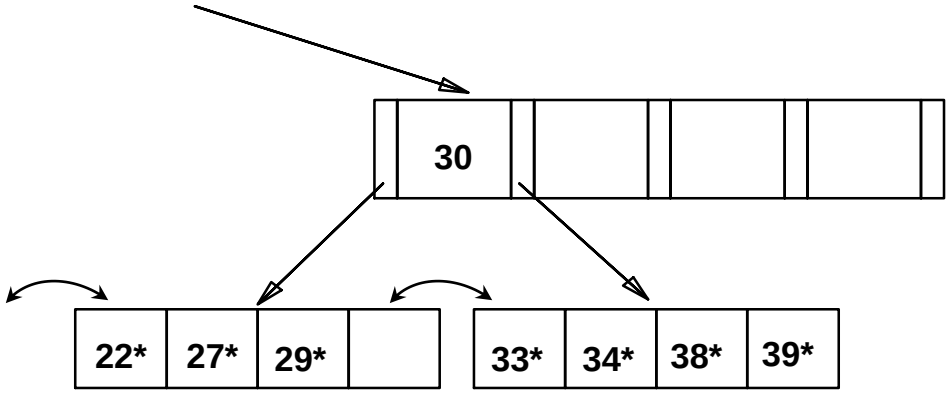
19* does not pose problems,
but 20* creates underflow



Entry re-distribution, middle key is *copied up*

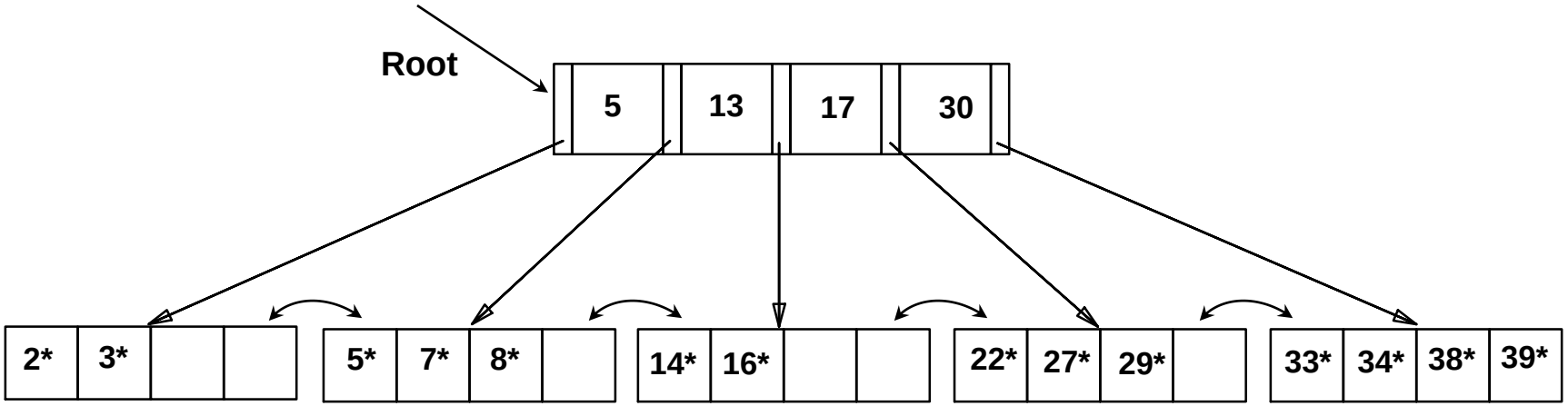
Then Delete 24*

- Must merge with sibling
- Index entry 27 is deleted, due to removal of leaf node



Index node with 30 underflow,

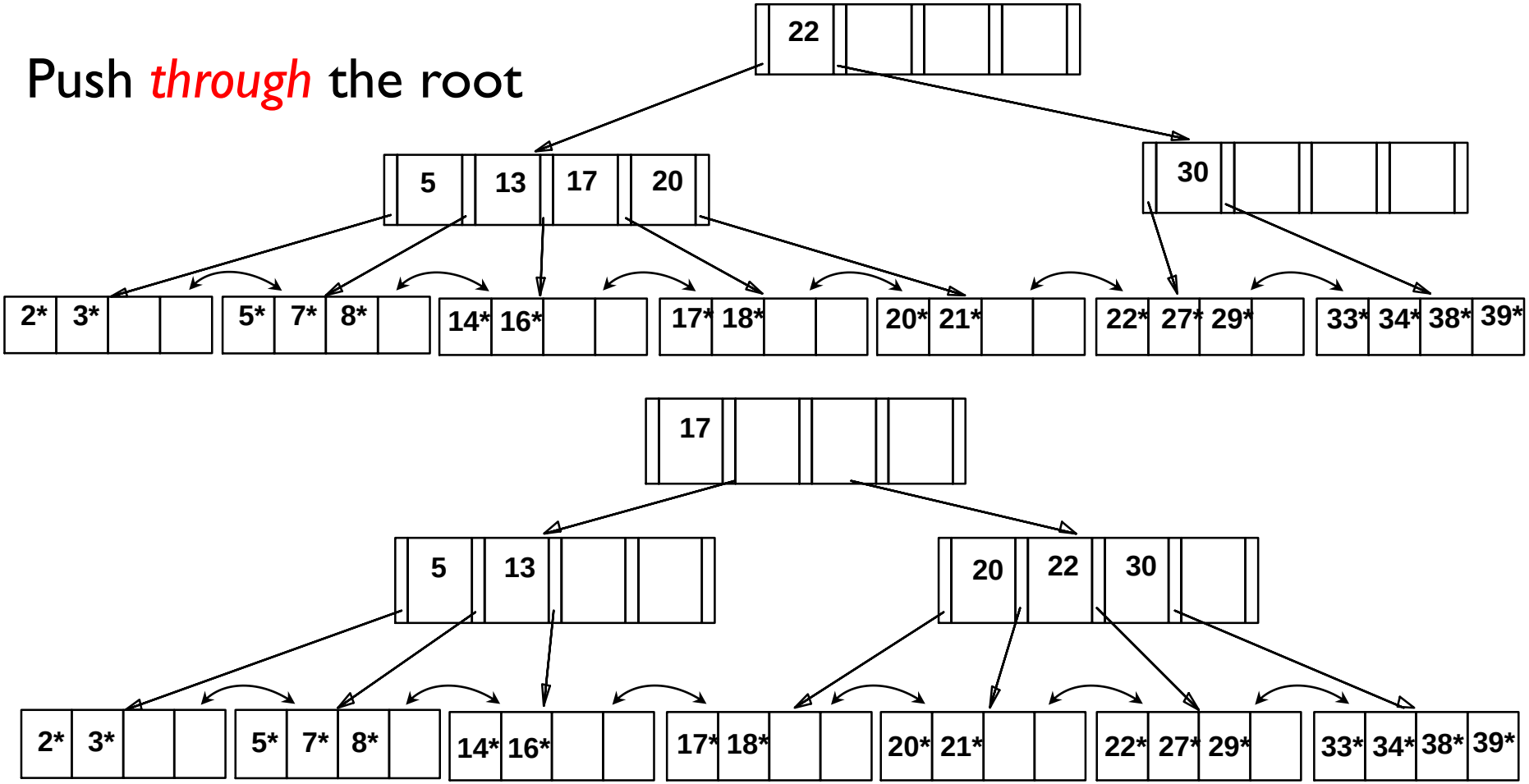
pull down of index key from parent



Non-leaf Re-distribution

- If possible, re-distribute entry from index node sibling

Push *through* the root

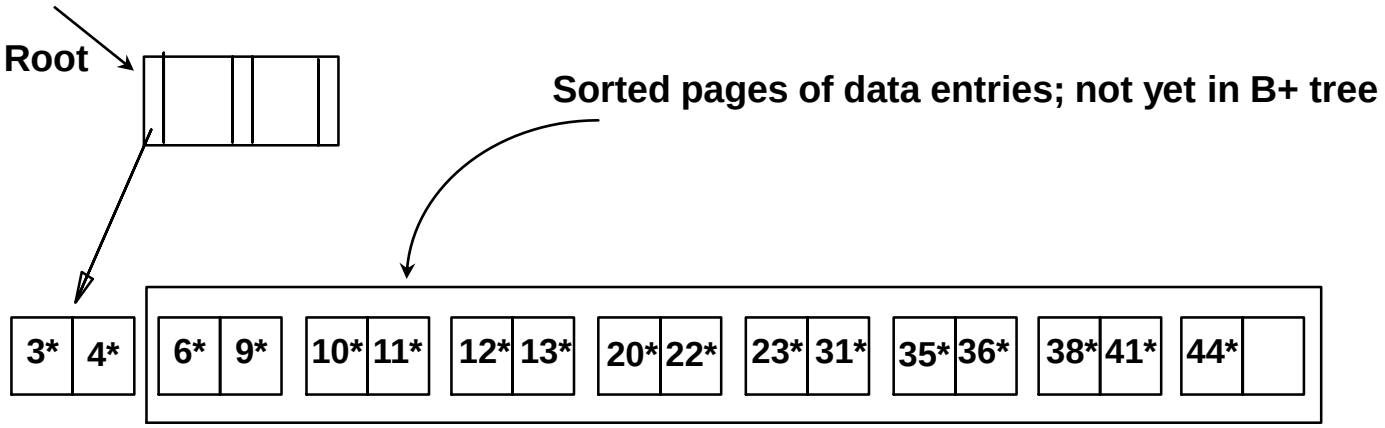


B+ Tree Deletion Algorithm

- 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
    - If re-distribution fails, merge *L* and sibling
- If merge occurred, must delete entry from parent of *L*
- Merge could propagate to root, decreasing height

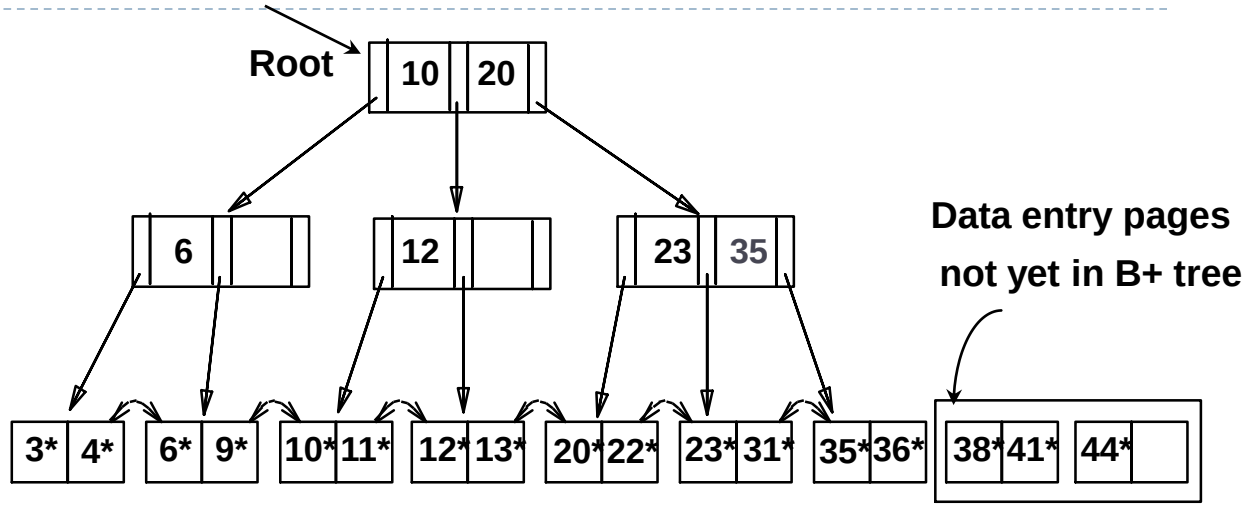
Bulk Loading of a B+ Tree (d=1)

- Method to efficiently build a tree for first time
  - Much better than doing repeated insertions
  - Can place pages sequentially on disk
- Sort all data entries, insert pointer to first leaf page in a new root page

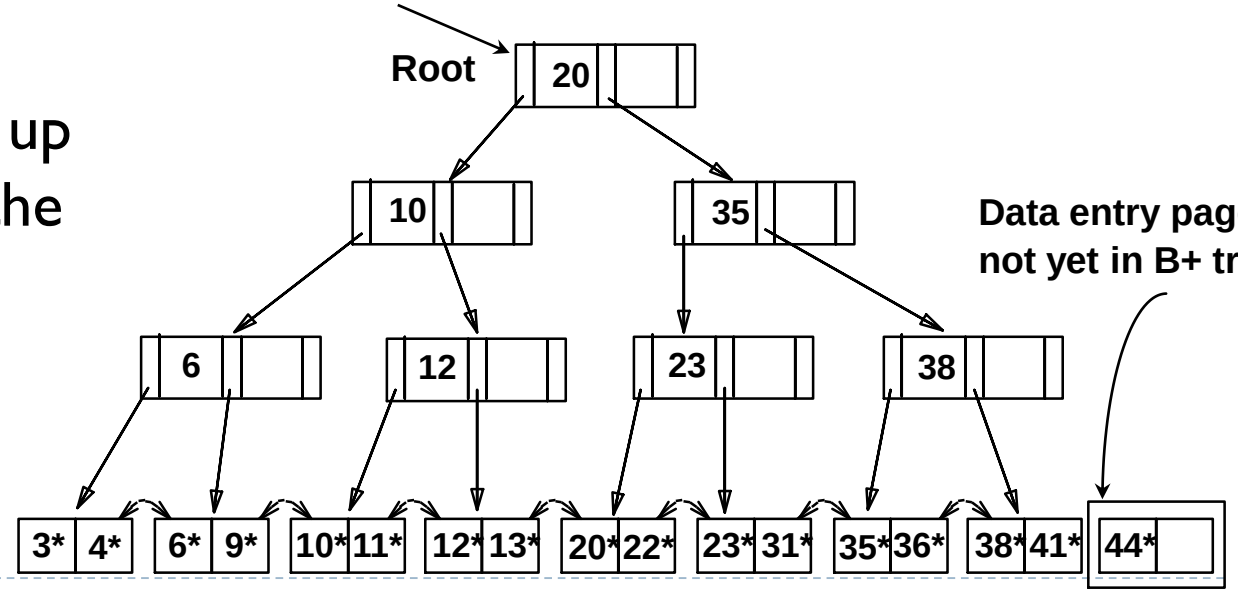


Bulk Loading (Contd.)

- 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)



B+ Tree: Prefix Key Compression

- Good I/O performance due to low tree height
  - Important to increase fan-out
- Key values in index entries only direct search
  - Sometimes keys can be long (e.g., long names)
  - It is possible to compress them
- Key Compression Rule
  - Each index entry greater than every key value (in any subtree) to its left
- Insert/delete must be suitably modified

Summary

- Tree-structured indexes are ideal for range-searches, also good for equality searches.
- ISAM tree 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; adjusts to growth gracefully.
- But concurrency control (locking) is easier in ISAM
- 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.