

Tree indexes

Iztok Sarnik, FAMNIT

Slides & Textbook

- Textbook:
 - Raghu Ramakrishnan, Johannes Gehrke, *Database Management Systems*, McGraw-Hill, 3rd ed., 2007.
- *Slides*:
 - From „Cow Book“: R.Ramakrishnan, <http://pages.cs.wisc.edu/~dbbook/>

Introduction

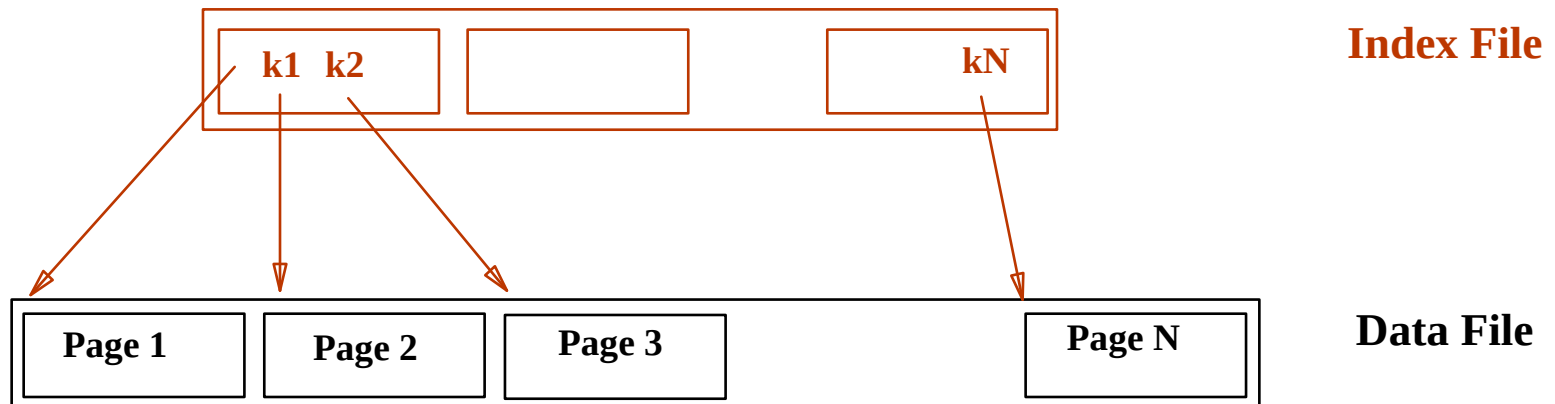
- *As for any index, 3 alternatives for data entries k^* :*
 - Data record with key value k
 - $\langle k, \text{rid of data record with search key value } k \rangle$
 - $\langle k, \text{list of rids of data records with search key } k \rangle$
- Choice is orthogonal to the *indexing technique* used 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.

Tree indexes

- Tree index structure appropriate for
 - Range search as well as
 - Equality search
- ISAM:
 - Static structure
 - Tree can become unbalanced
- B+ tree:
 - Dynamic structure
 - Tree is always balanced

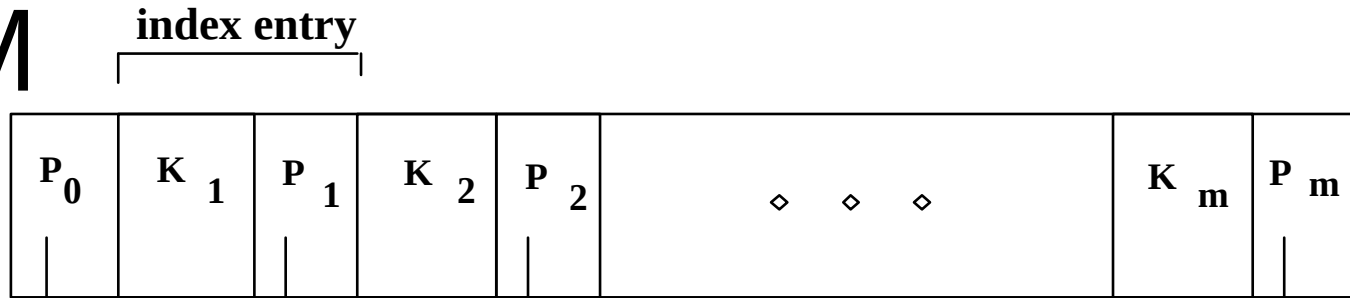
Range Searches

- *“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.

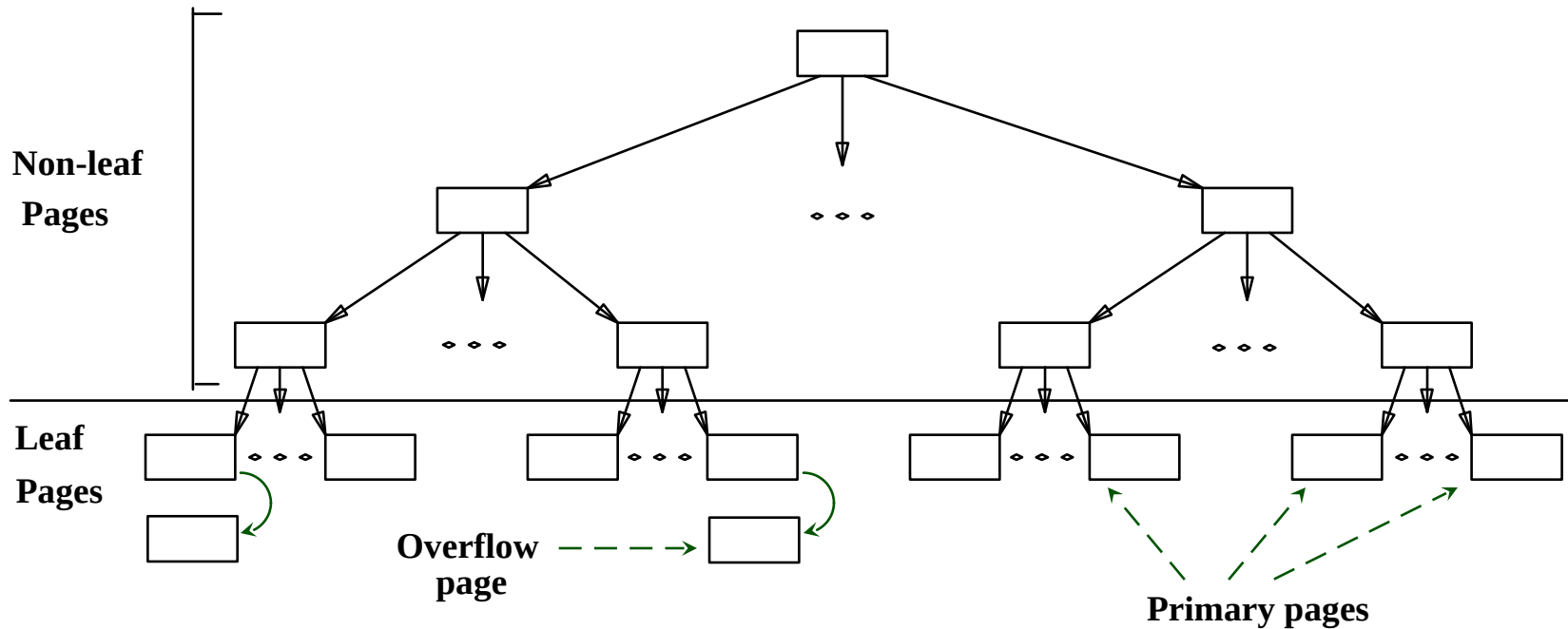


👉 *Can do binary search on (smaller) index file!*

ISAM



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



☛ *Leaf pages contain data entries.*

Comments on ISAM

Data
Pages

Index Pages
pages

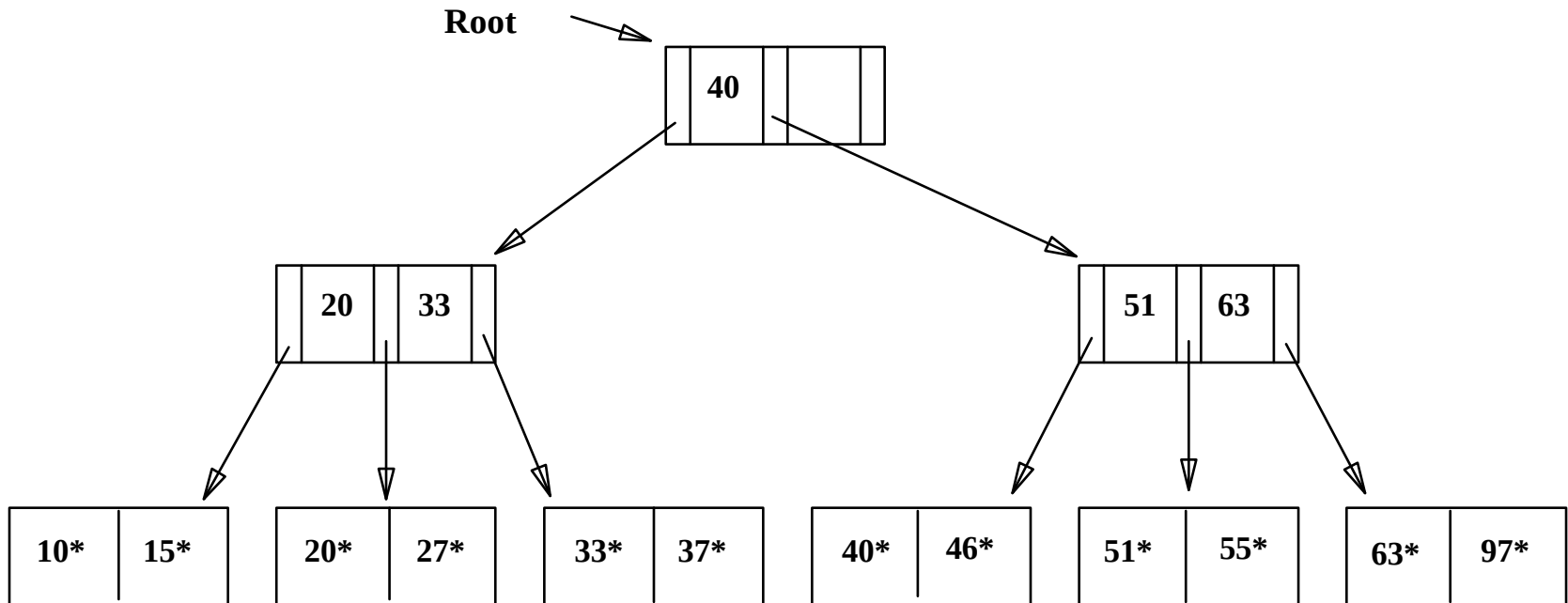
Overflow pages

- *File creation*: Leaf (data) pages allocated sequentially, sorted by search key; then index 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 $\log_F N$; $F = \# \text{ entries/index pg}$, $N = \# \text{ leaf pgs}$
- *Insert*: Find leaf data entry belongs to, and put it there.
- *Delete*: Find and remove from leaf; if empty overflow page, de-allocate.

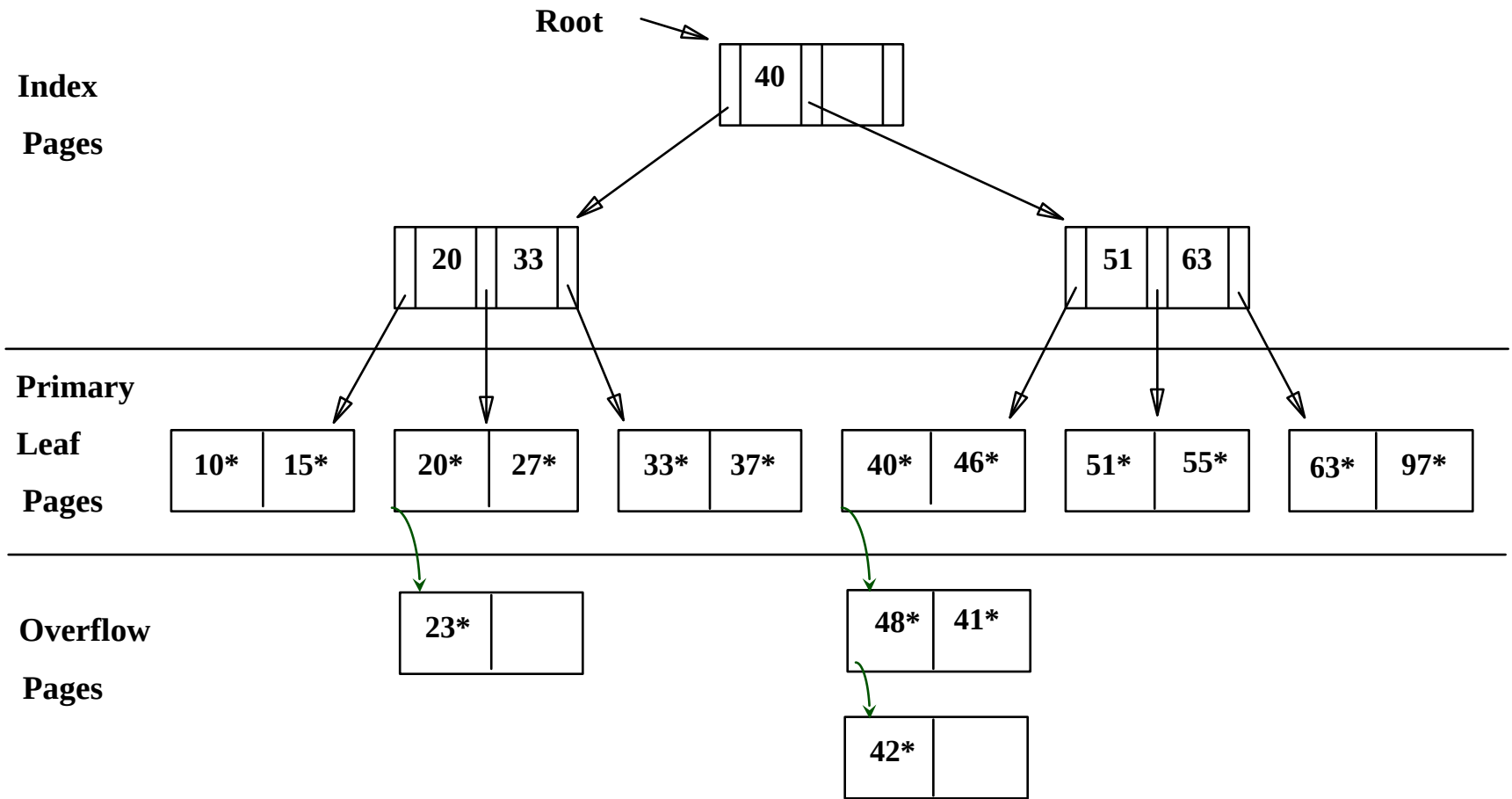
• **Static tree structure**: *inserts/deletes affect only leaf pages.*

Example ISAM Tree

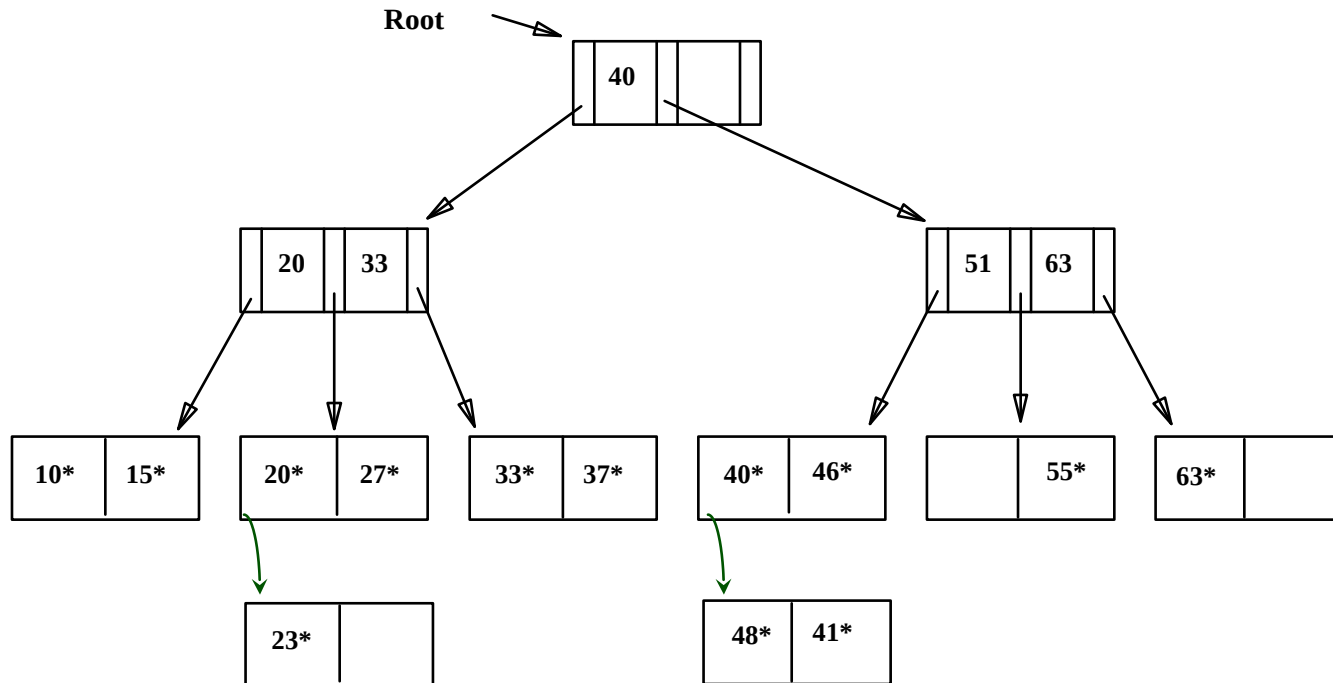
- Each node can hold 2 entries; no need for 'next-leaf-page' pointers. (Why?)



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



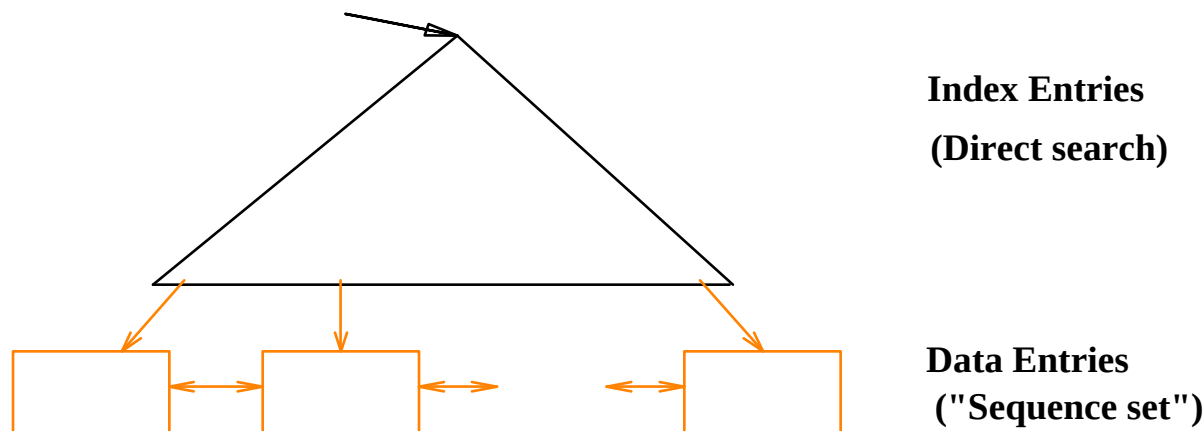
... Then Deleting 42*, 51*, 97*



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

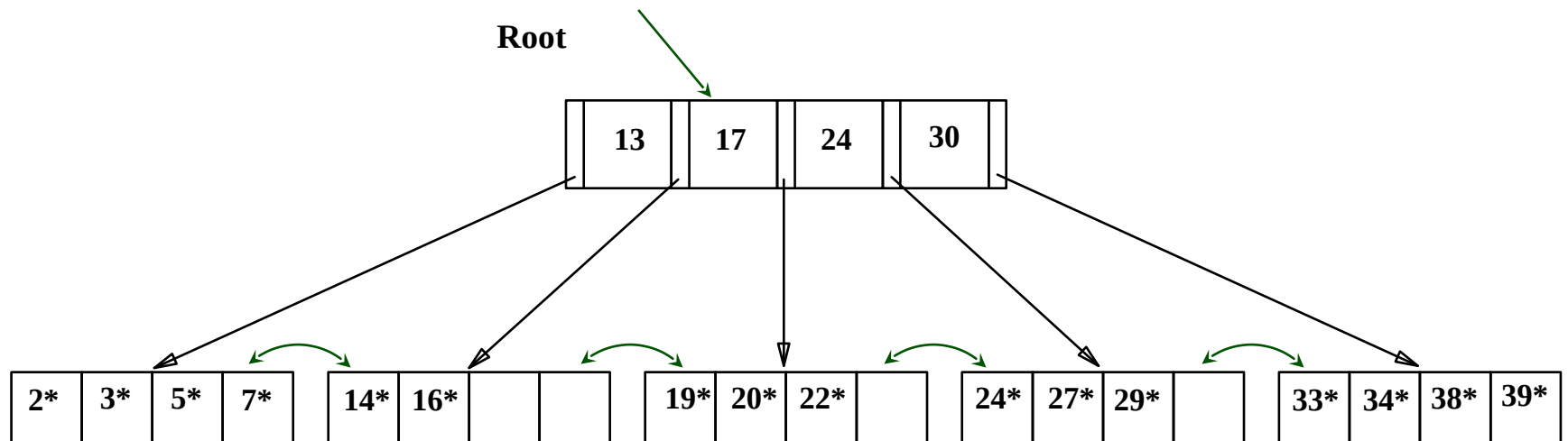
B+ Tree: Most Widely Used Index

- 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 $d \leq m \leq 2d$ 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 $\geq 24^*$...



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

B+ Trees in Practice

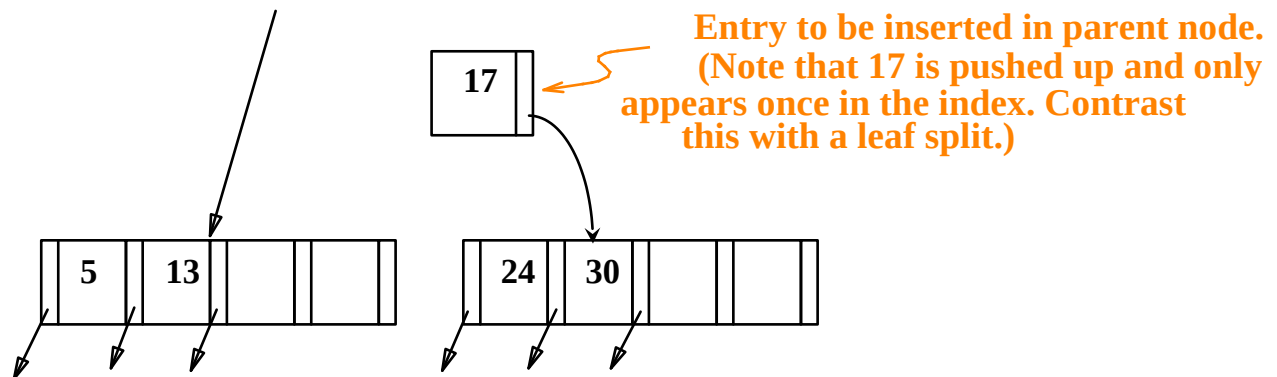
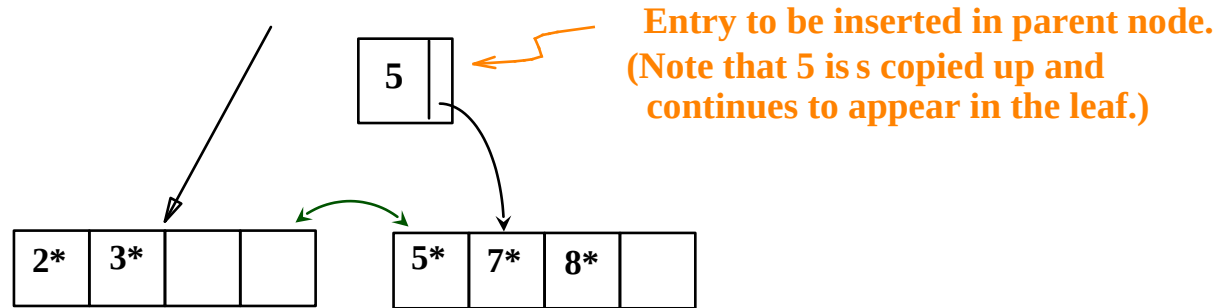
- Typical order: 100. Typical fill-factor: 67%.
 - average fanout = 133
- Typical capacities:
 - Height 4: $133^4 = 312,900,700$ records
 - Height 3: $133^3 = 2,352,637$ records
- Can often hold top levels in buffer pool:
 - Level 1 = 1 page = 8 Kbytes
 - Level 2 = 133 pages = 1 Mbyte
 - Level 3 = 17,689 pages = 133 MBytes

Inserting a Data Entry into a B+ Tree

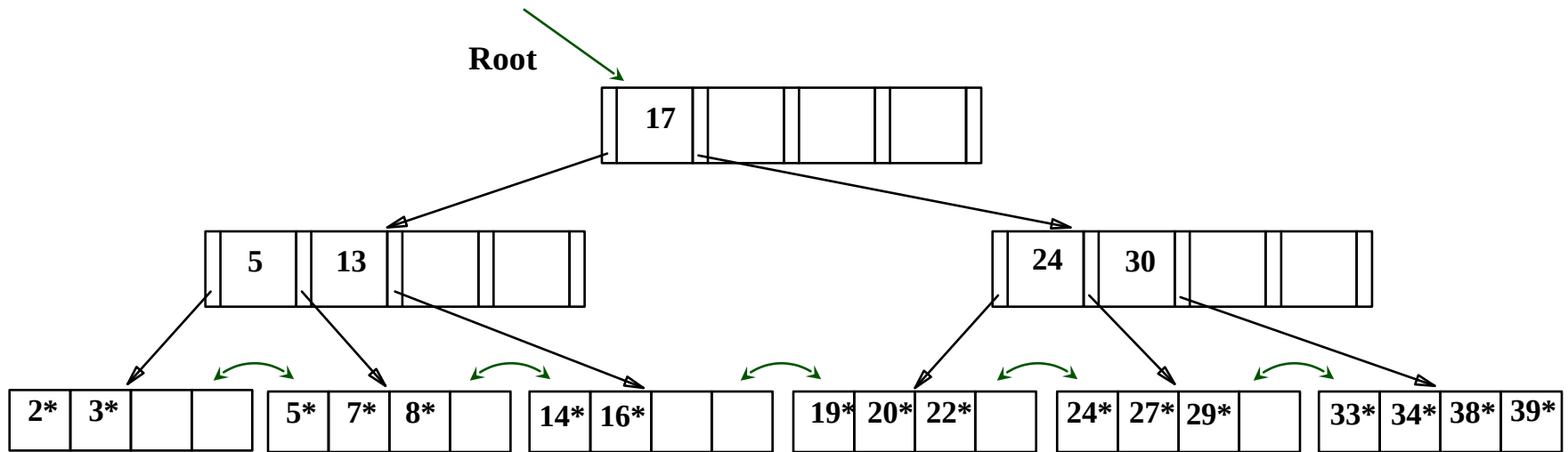
- Find correct leaf L .
- Put data entry onto L .
 - If L has enough space, *done!*
 - Else, must split L (into L and a new node $L2$)
 - Redistribute entries evenly, copy up middle key.
 - Insert index entry pointing to $L2$ into parent of L .
- This can happen recursively
 - To split index node, redistribute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

Inserting 8* into Example B+ Tree

- 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.



Example B+ Tree After Inserting 8*



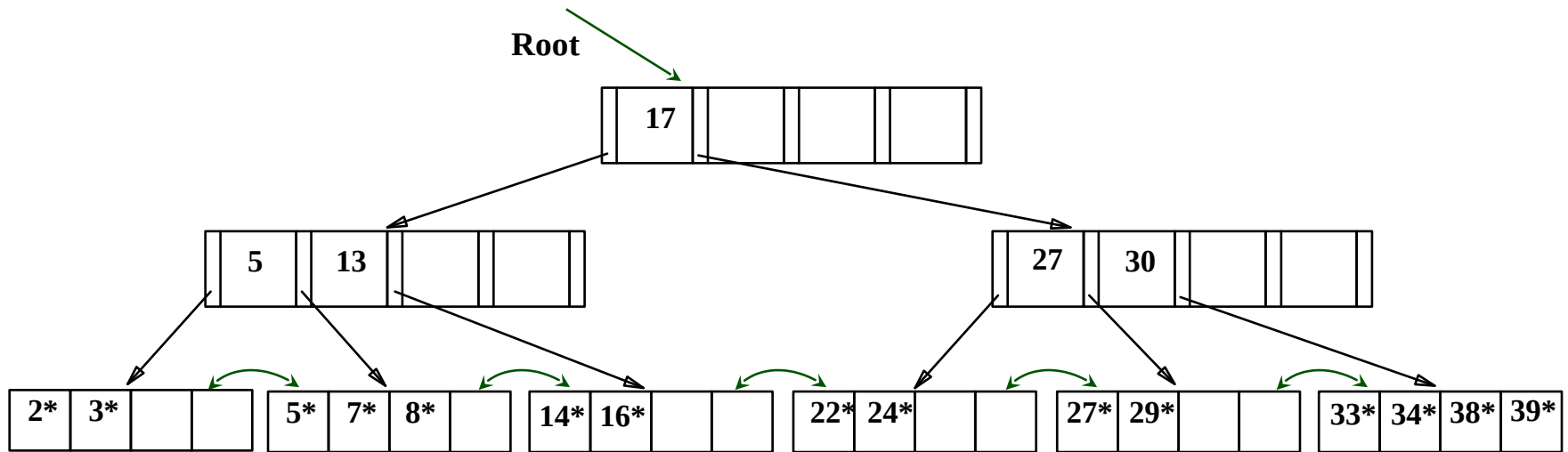
❖ Notice that root was split, leading to increase in height.

- 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*).
 - 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.

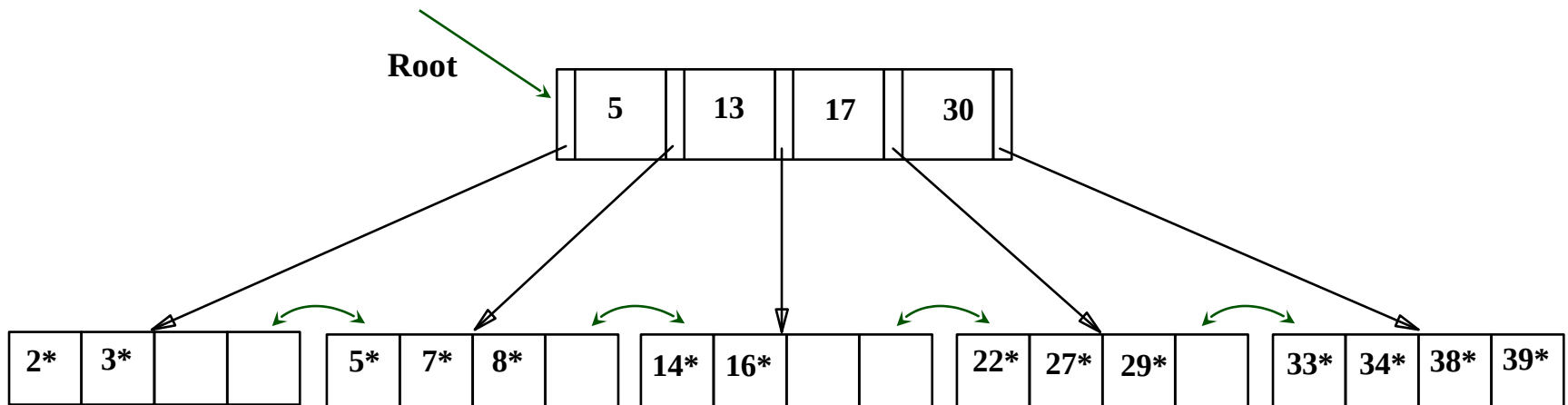
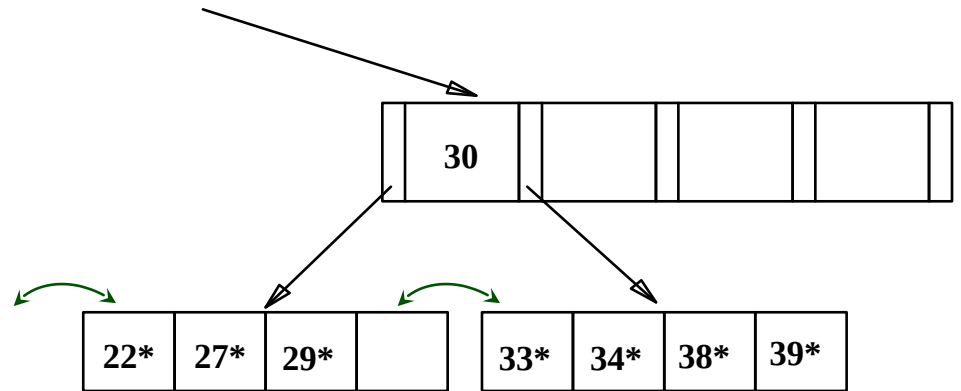
Example Tree After (Inserting 8*, Then) Deleting 19* and 20* ...



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

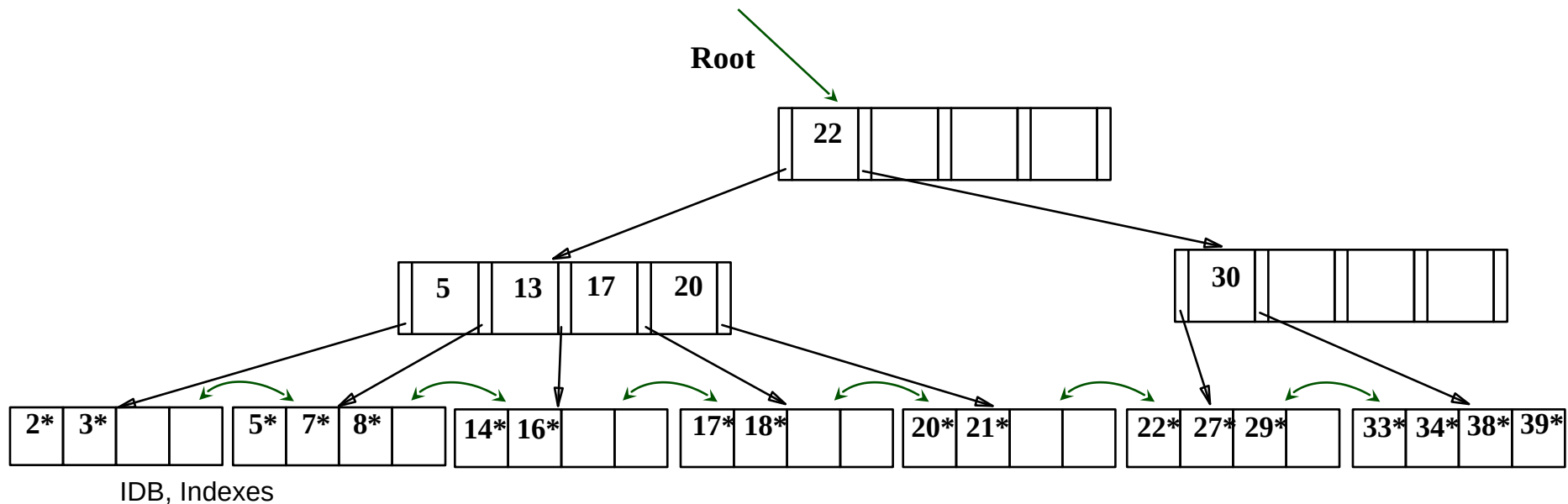
... And Then Deleting 24*

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



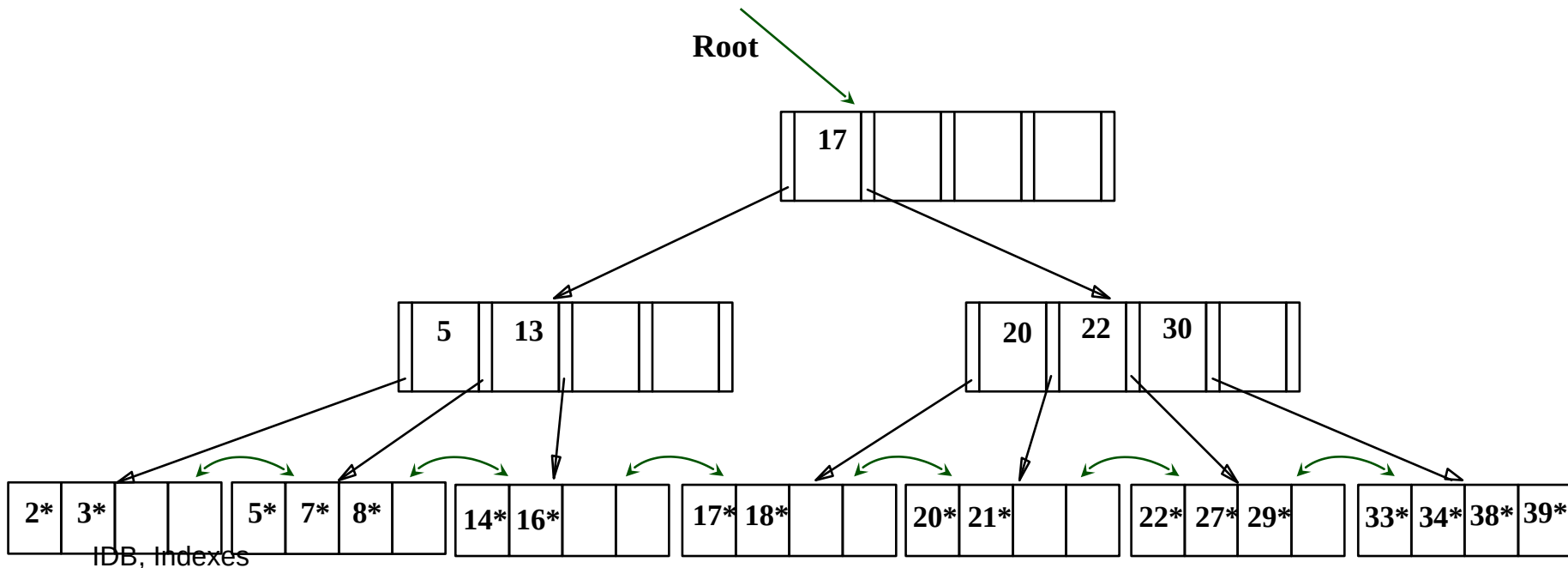
Example of Non-leaf Re-distribution

- 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.

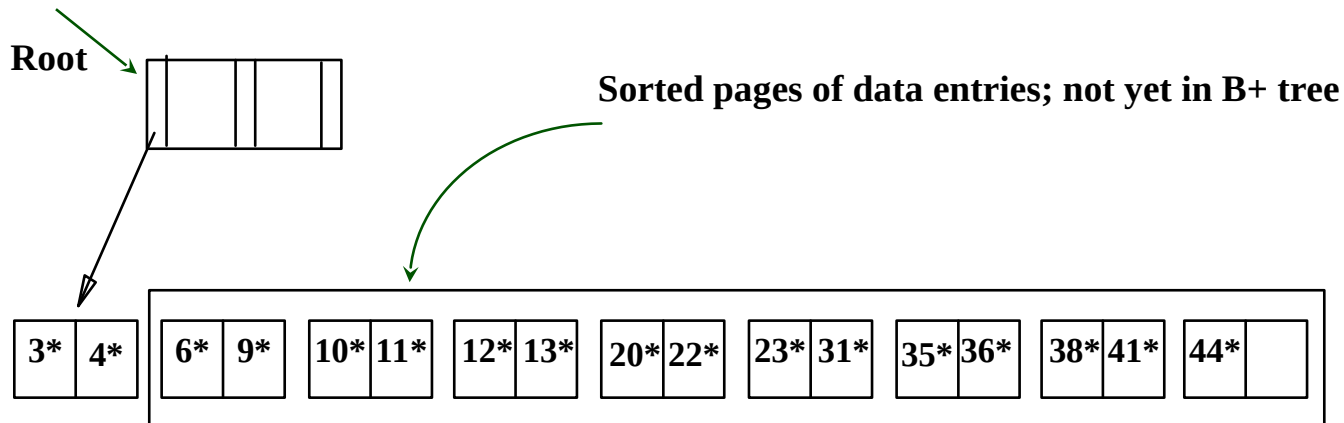


Prefix Key Compression

- 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.

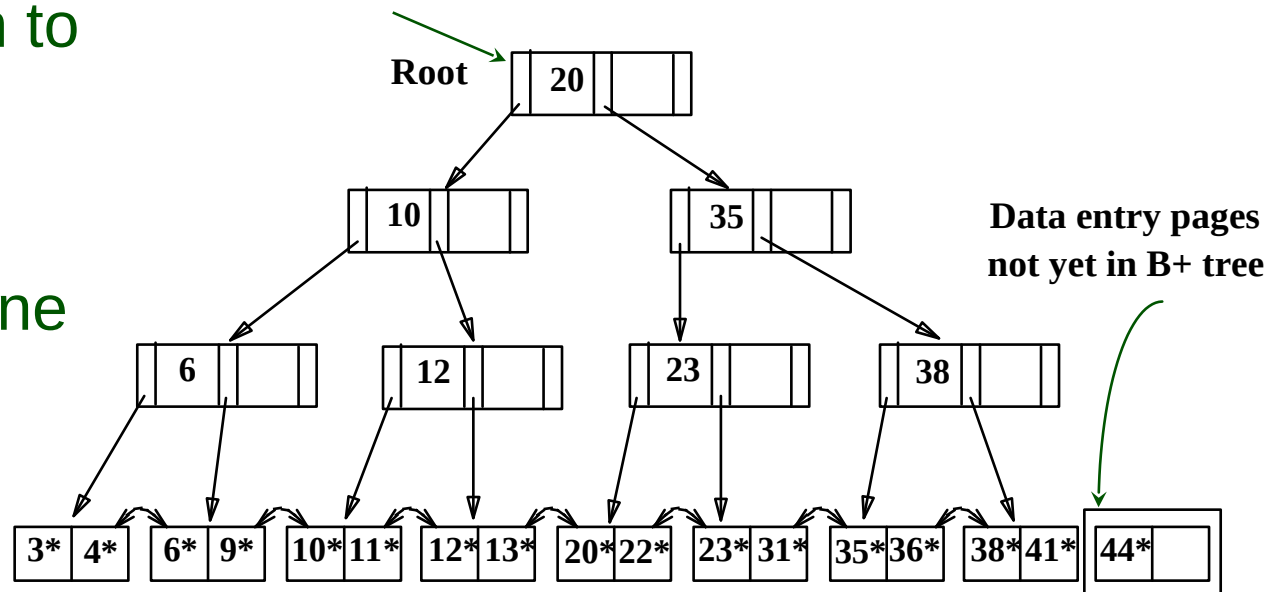
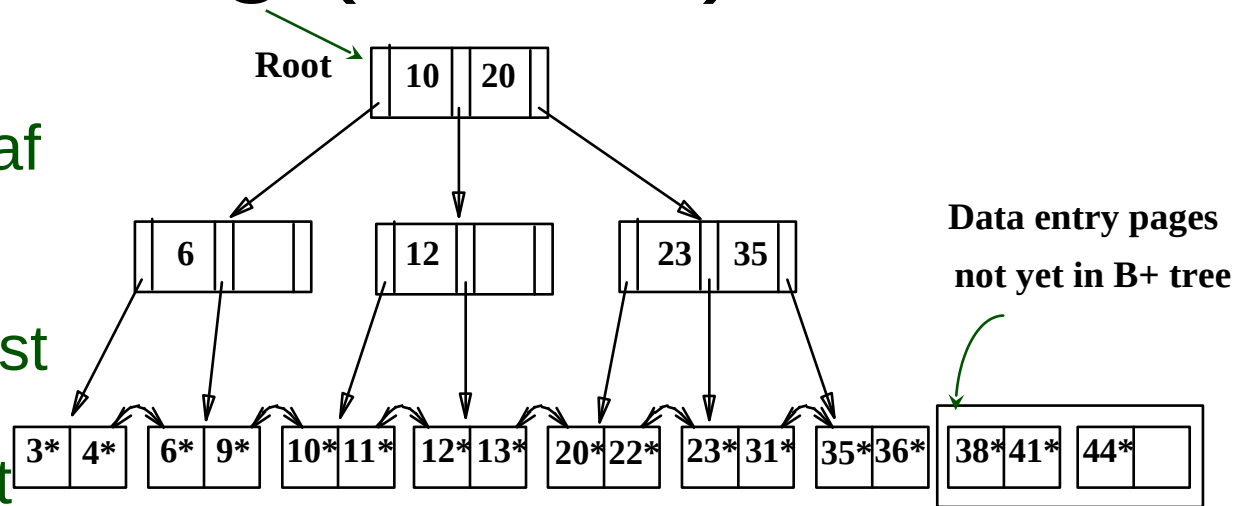
Bulk Loading of a B+ Tree

- 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.
- 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, especially when 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.

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)).

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.