CAS CS 460/660
Introduction to Database Systems

Tree Based Indexing: B+-tree

Slides from UC Berkeley

How to Build Tree-Structured Indexes

- Tree-structured indexing techniques support both *range searches* and *equality searches*.

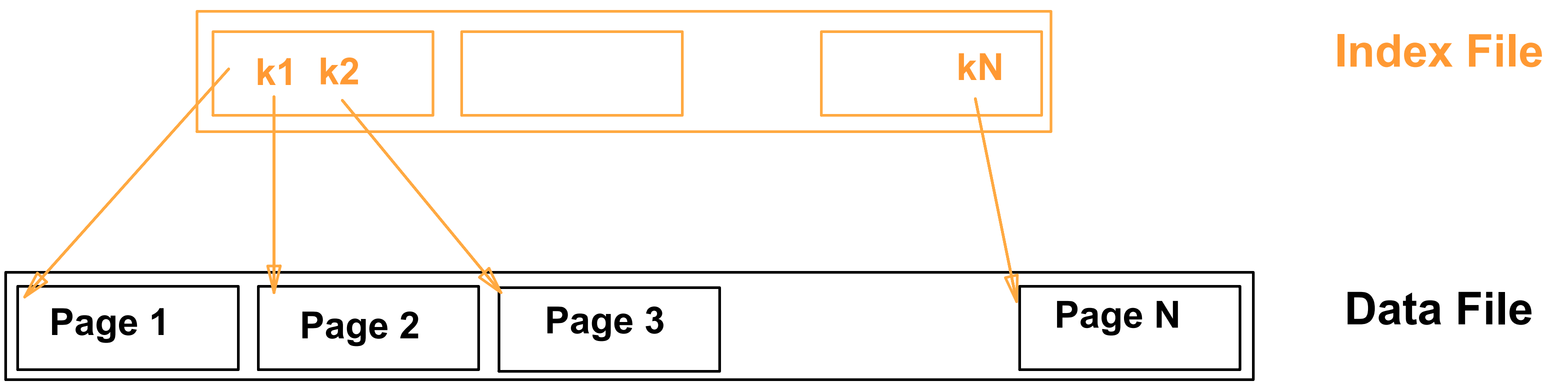
- Two examples:
  - *ISAM*: static structure; early index technology.
  - *B+ tree*: dynamic, adjusts gracefully under inserts and deletes.

Indexed Sequential Access Method

- 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, but many of the same ideas

Range Searches

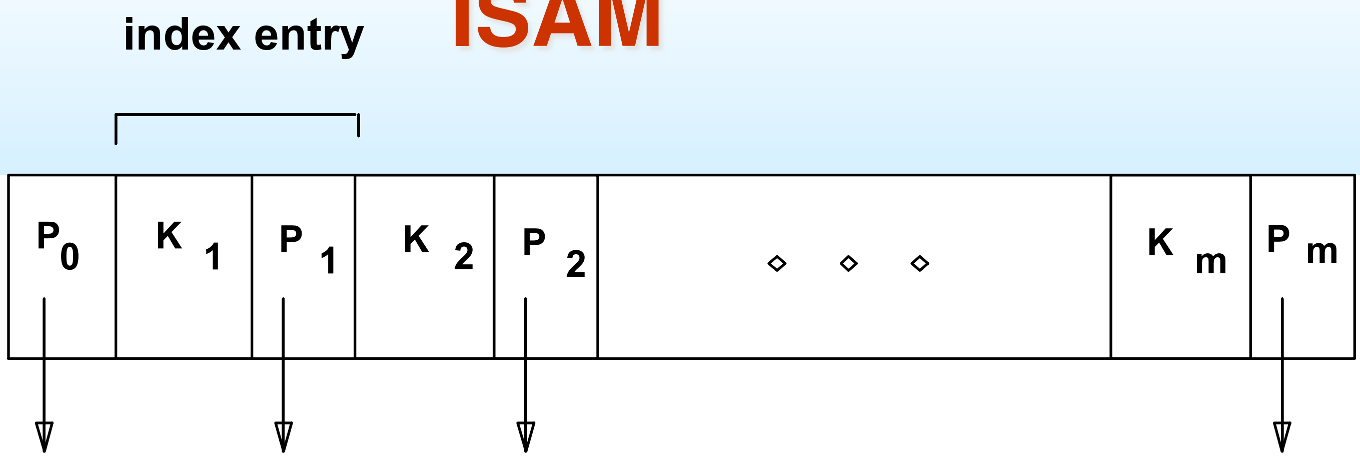
- *``Find all students with gpa > 3.0''*
  - If data is in sorted file, do binary search to find first such student, then scan to find others.
  - Cost of binary search on disk is still quite high. Why?
- Simple idea: Create an `index' file.



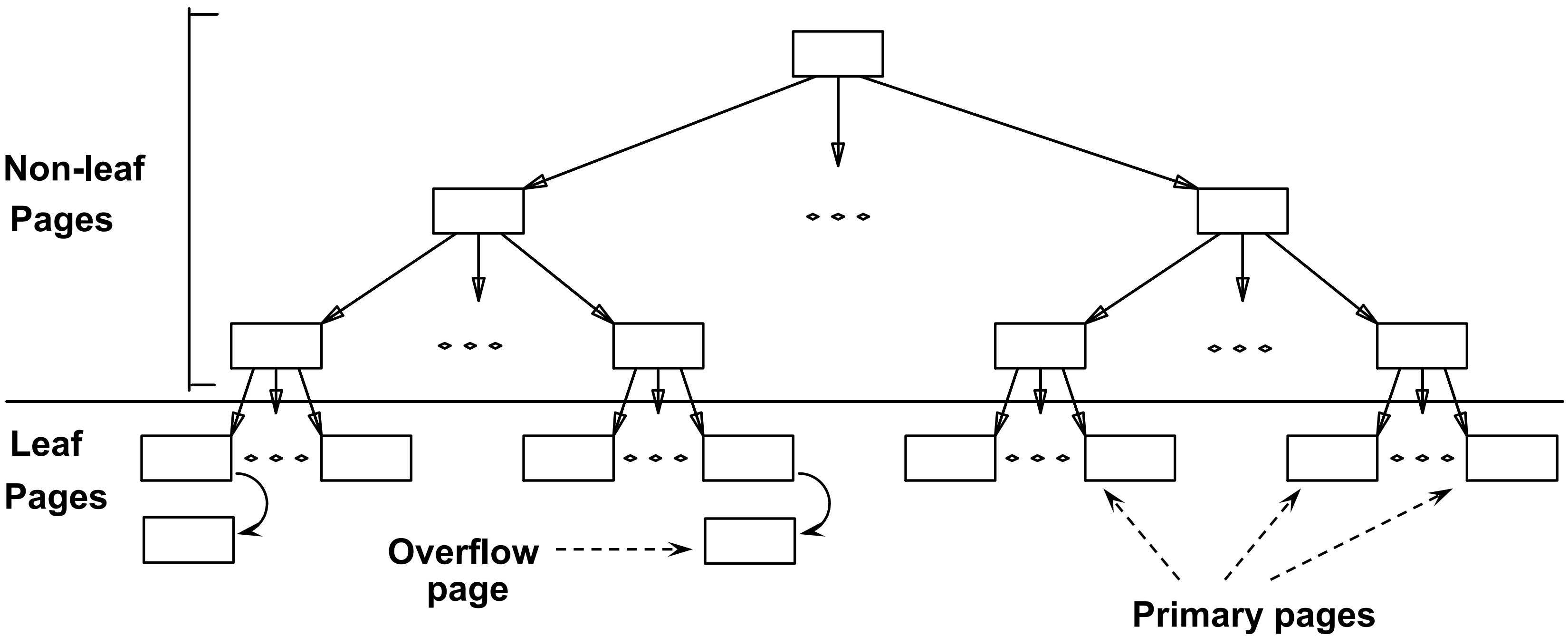
☛ *Can do binary search on (smaller) index file!*

☛*But what if index doesn't fit easily in memory?*

ISAM

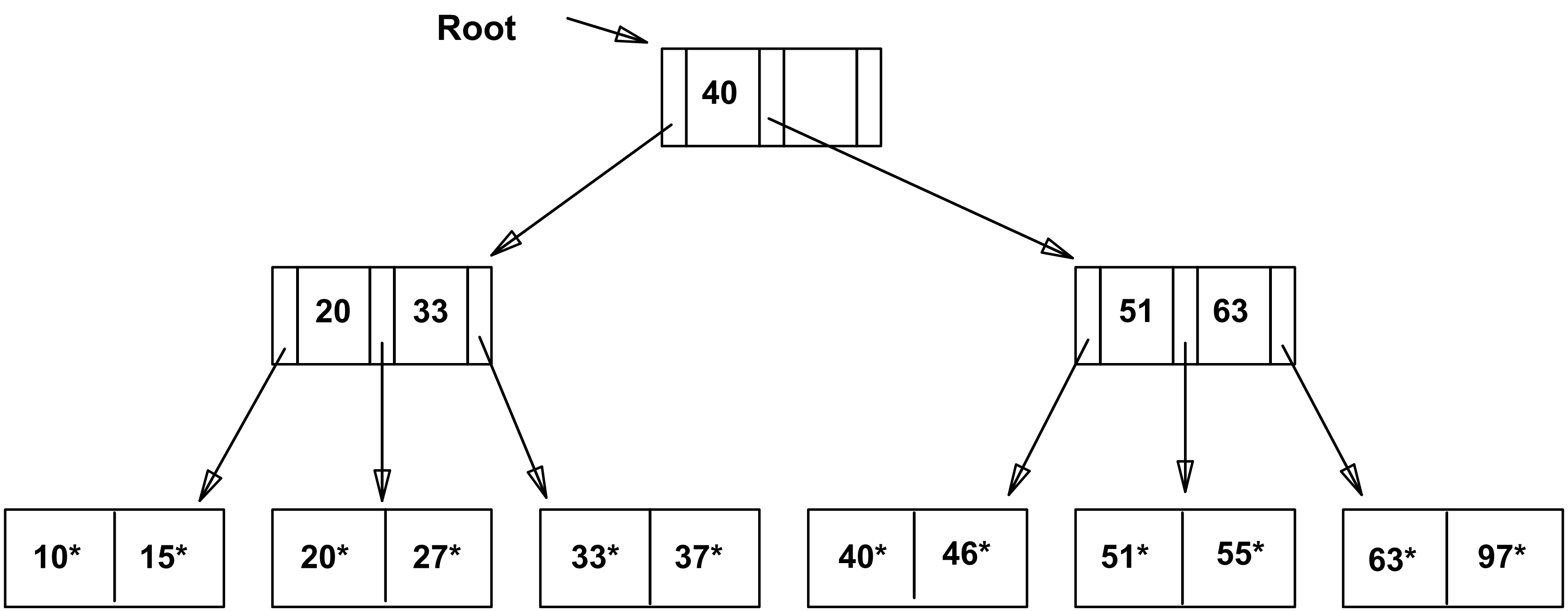


We can apply the idea repeatedly!



Example ISAM Tree

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



ISAM has a STATIC Index Structure

File creation:

1. Allocate leaf (data) pages sequentially
2. Sort records by search key
3. Allocate and fill index pages

(now the structure is ready for use)

4. Allocate and overflow pages as needed

ISAM File Layout

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

ISAM (continued)

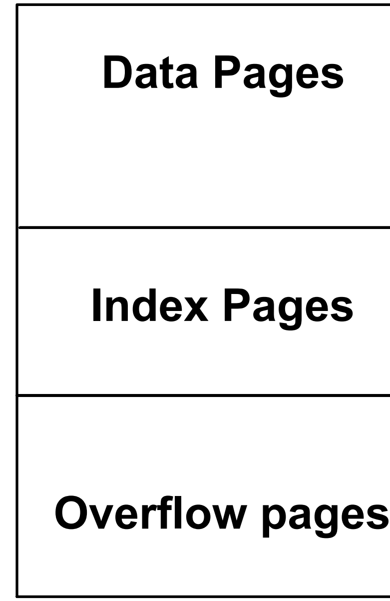
Search: Start at root; use key comparisons to navigate to leaf.

$$\text{Cost} = \log_F N$$

F = # entries/pg (i.e., fanout)

N = # leaf pgs

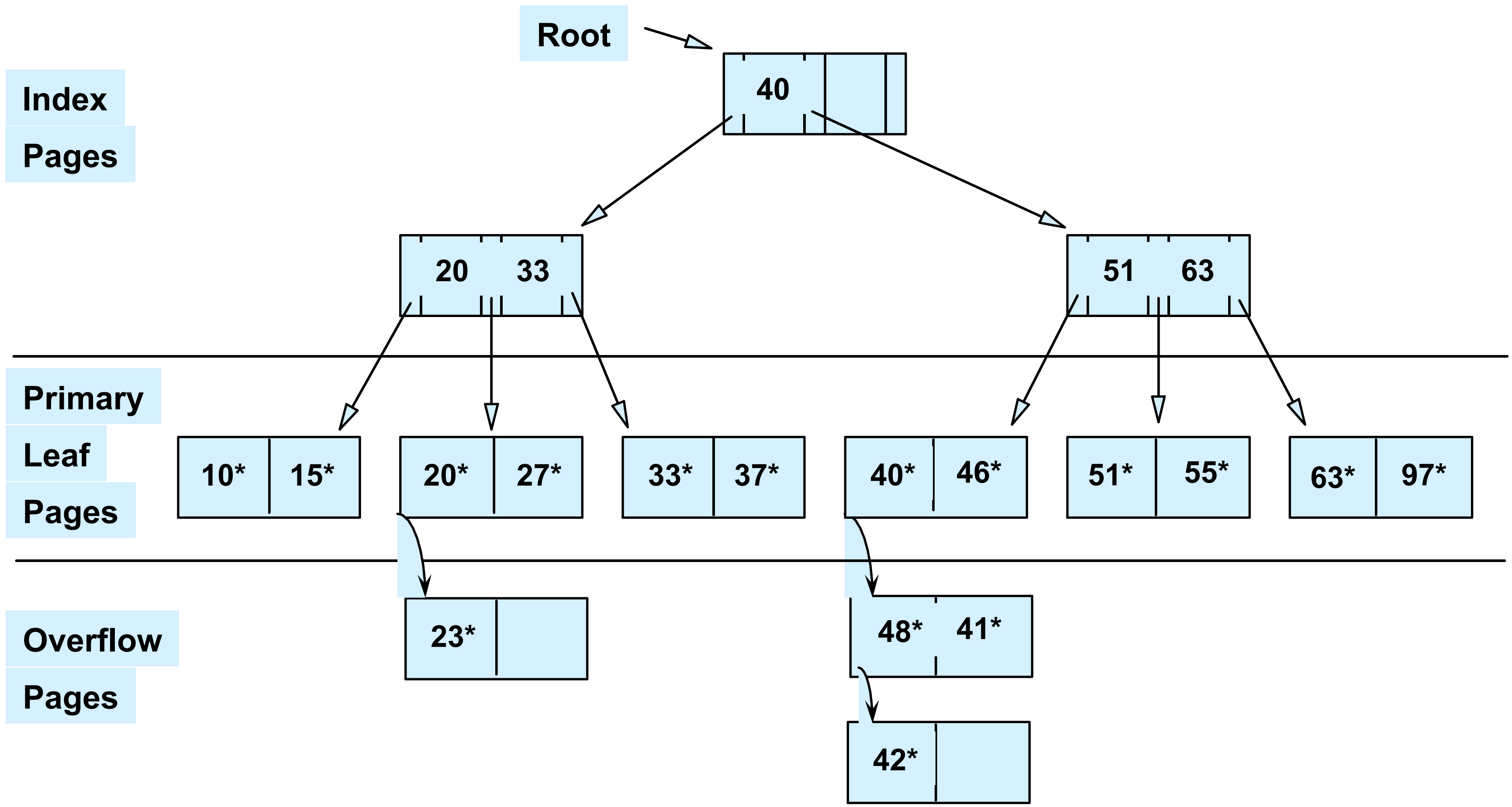
- no need for `next-leaf-page' pointers. (Why?)



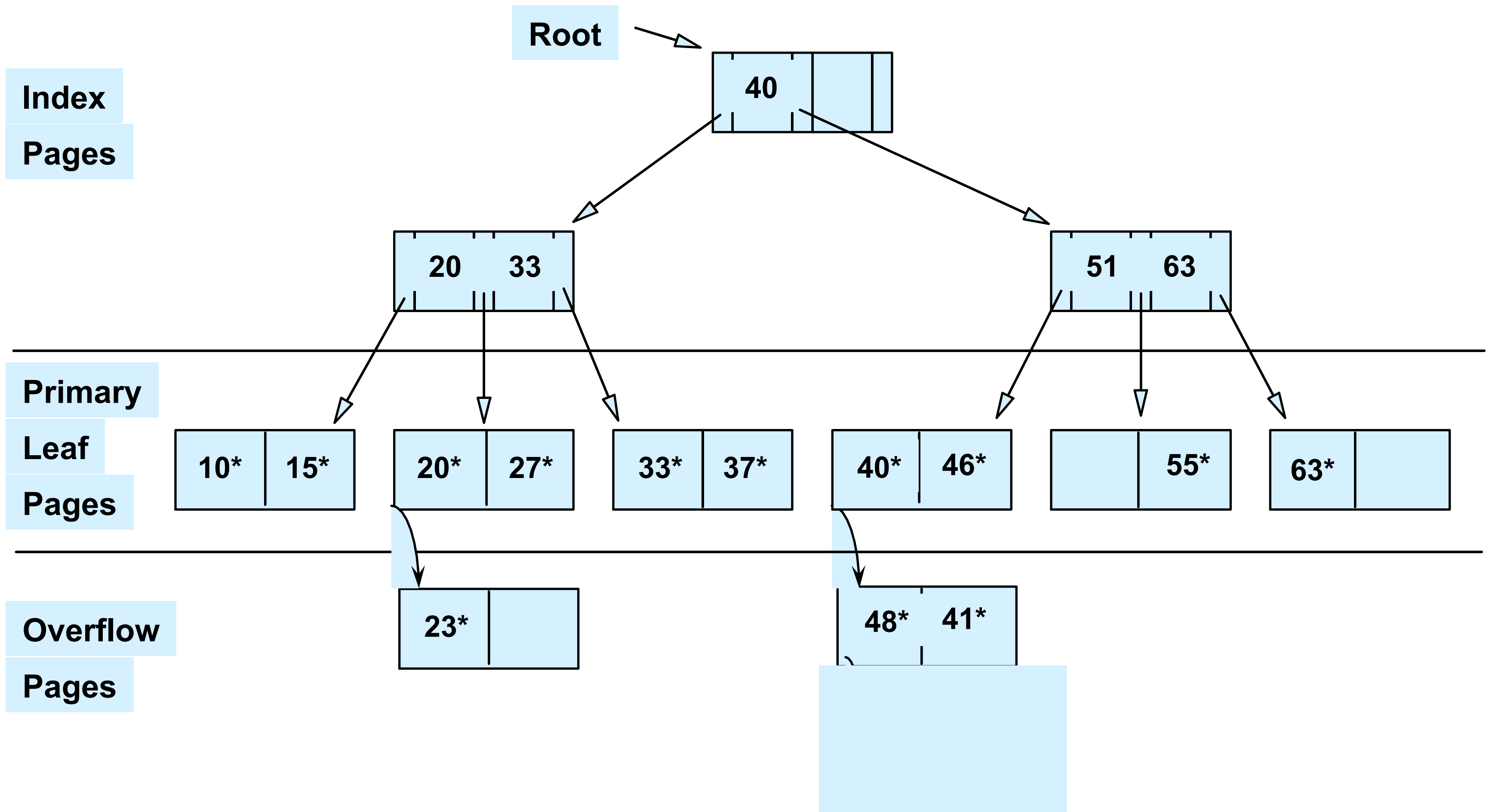
Insert: Find leaf that data entry belongs to, and put it there. Overflow page if necessary.

Delete: Find; remove from leaf; if empty de-allocate.

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
  - ????

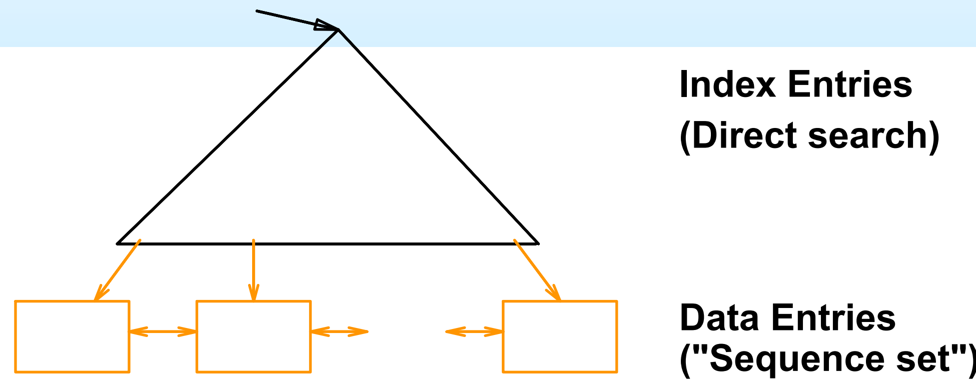
- Cons
  - ????

B+ Tree: The Most Widely Used Index

Insert/delete at log $_F$ N cost;

keep tree *height-balanced*.

N = # leaf pages



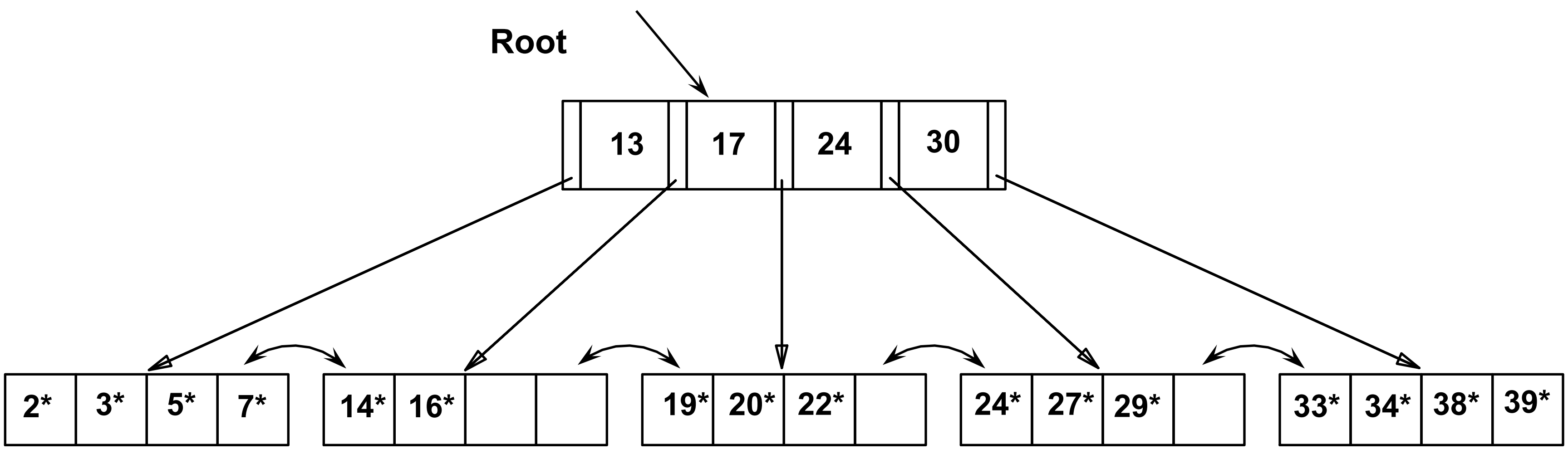
- Each node (except for root) contains *m entries:* d <= *m* <= 2d entries.
- "d" is called the *order* of the tree.

  (maintain 50% min occupancy)

- 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 page, and key comparisons direct it to a leaf (as in ISAM).
- Search for 5*, 15*, all data entries >= 24* ...



☛ *Based on the search for 15*, we <u>know</u> it is not in the tree!*

A Note on Terminology

- The "+" in B^+Tree indicates a special kind of "B Tree" in which all the data entries reside in leaf pages.
  - In a vanilla "B Tree", data entries are sprinkled throughout the tree.
- B^+Trees are simpler to implement than B Trees.
  - And since we have a large fanout, the upper levels comprise only a tiny fraction of the total storage space in the tree.
- To confuse matters, most database people (like me) call B^+Trees "B Trees"!!! (sorry!)

B+Tree Pages

Question: How big should the B+Tree pages (i.e., nodes) be?

- Hint 1: we want them to be fairly large (to get high fanout).
- Hint 2: they are typically stored in files on disk.
- Hint 3: they are typically read from disk into buffer pool frames.
- Hint 4: when updated, we eventually write them from the buffer pool back to disk.
- Hint 5: we call them "pages".

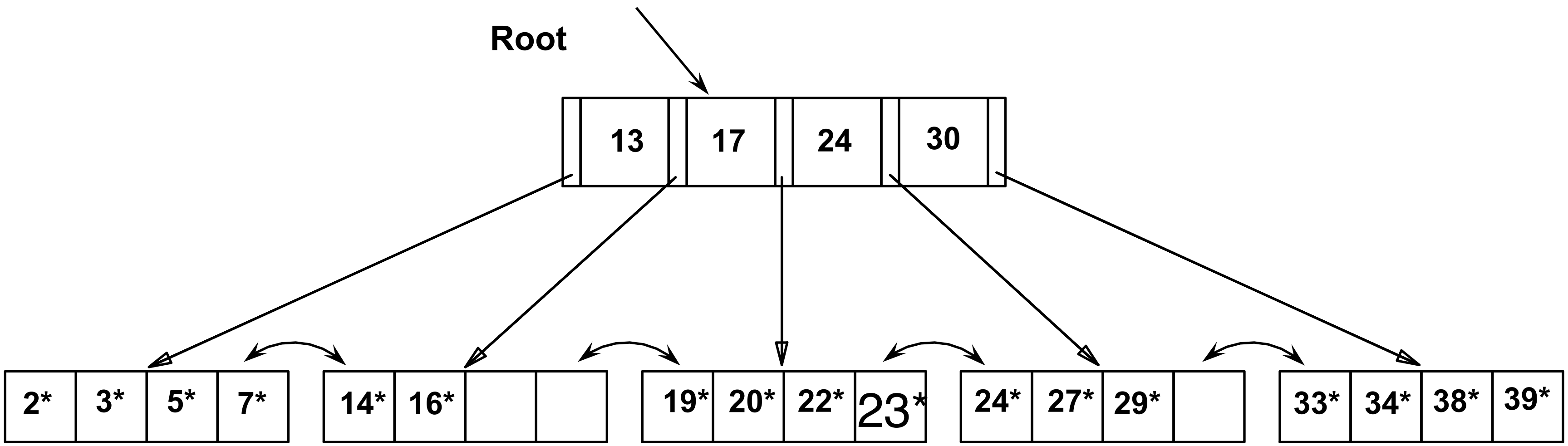
B+ Trees in Practice

- Remember = Index nodes are disk pages
  - e.g., fixed length unit of communication with disk
- Typical order: 100. Typical fill-factor: 67%.
  - average fanout = 133
- Typical capacities:
  - Height 3: 133^3 = 2,352,637 entries
  - Height 4: 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

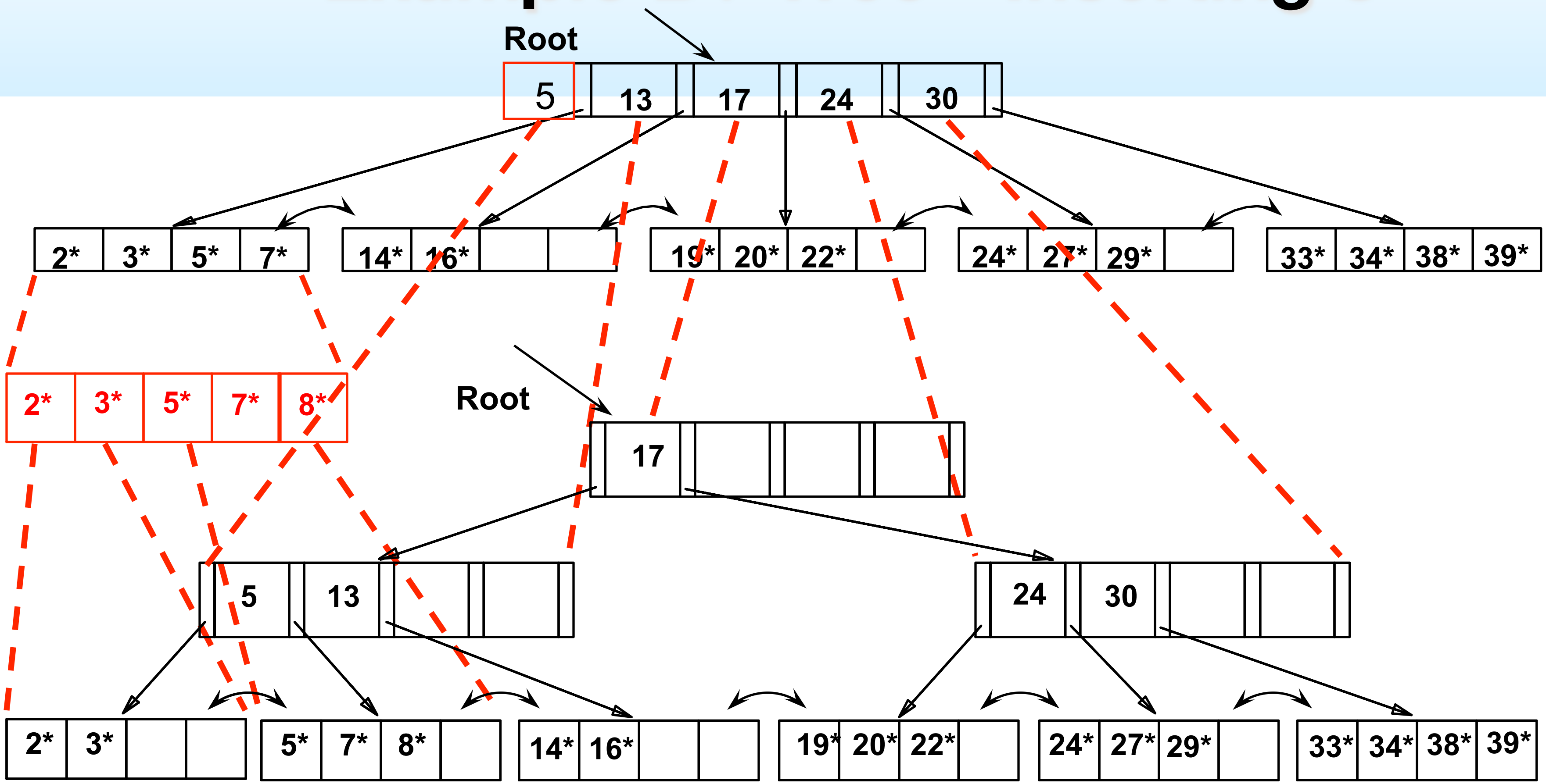
Inserting a Data Entry into a B+ Tree

- 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 23*



Example B+ Tree - Inserting 8*



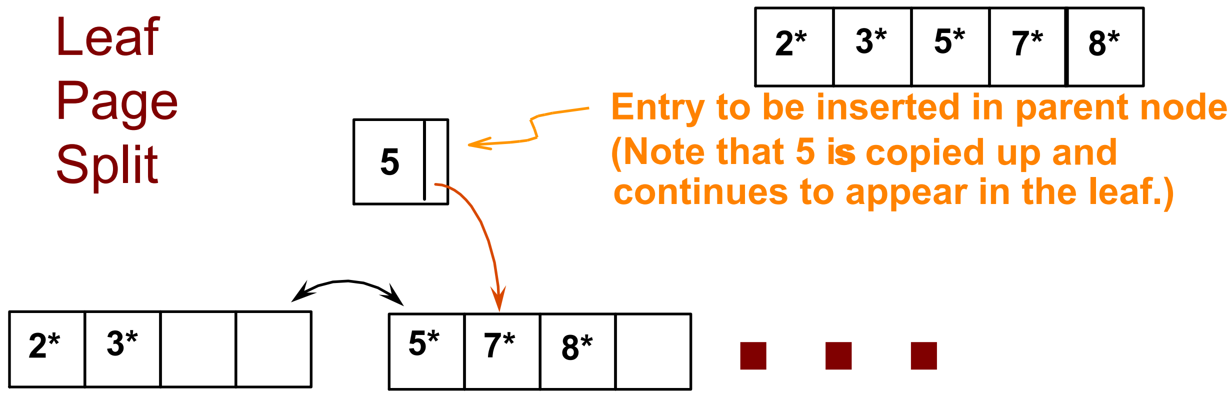
❖ Notice that root was split, leading to increase in height.

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

Leaf vs. Index Page Split
(from previous example of inserting “8”)

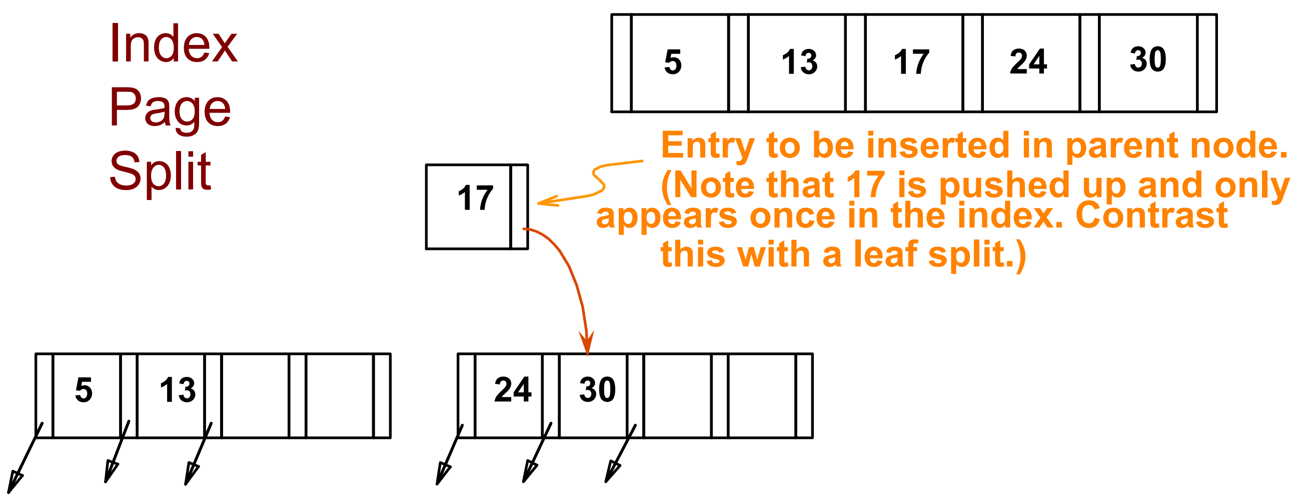
- Observe how minimum occupancy is guaranteed in both leaf and index pg splits.

Leaf Page Split



- Note difference between *copy-up* and *push-up*; be sure you understand the reasons for this.

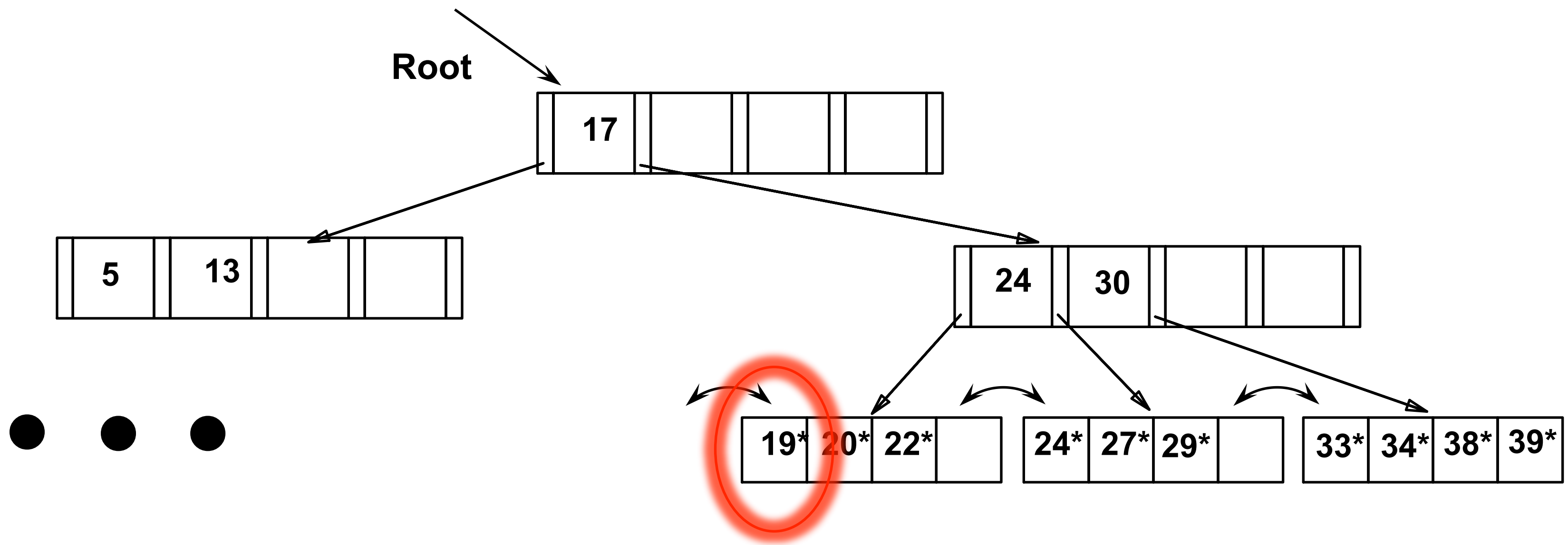
Index Page Split



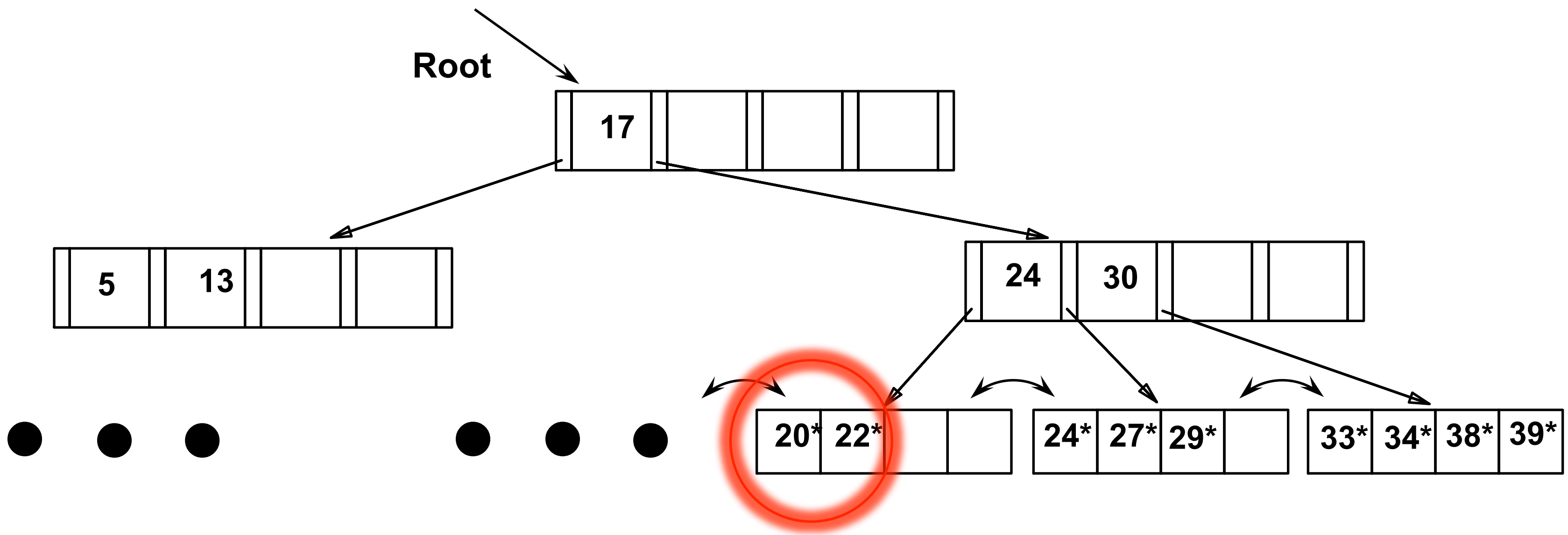
Deleting a Data Entry from a B+ Tree

- 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.

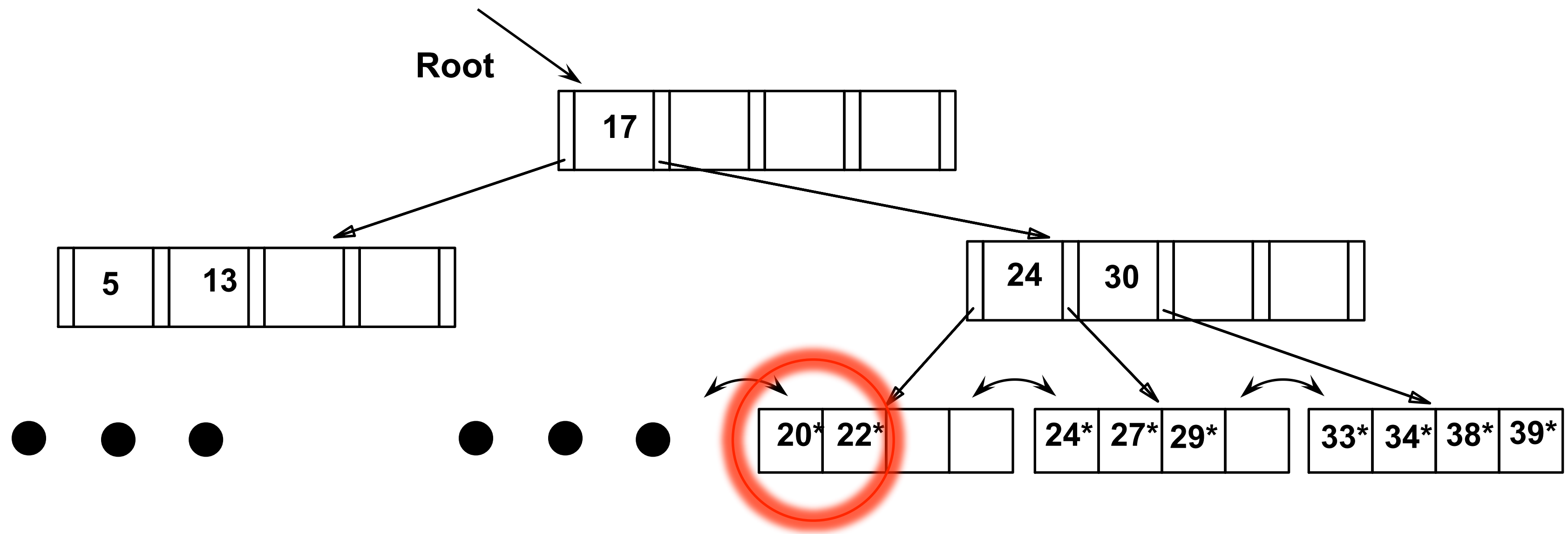
Example Tree - Delete 19*



Example Tree - Delete 19*

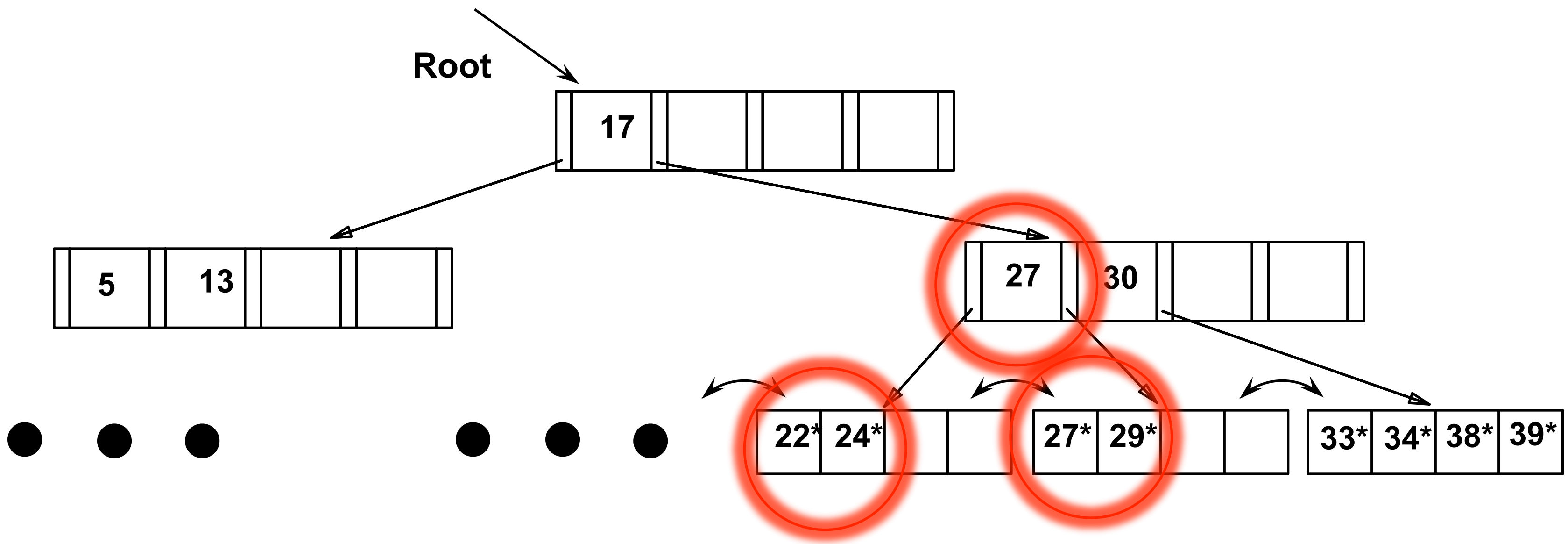


Example Tree – Now, Delete 20*

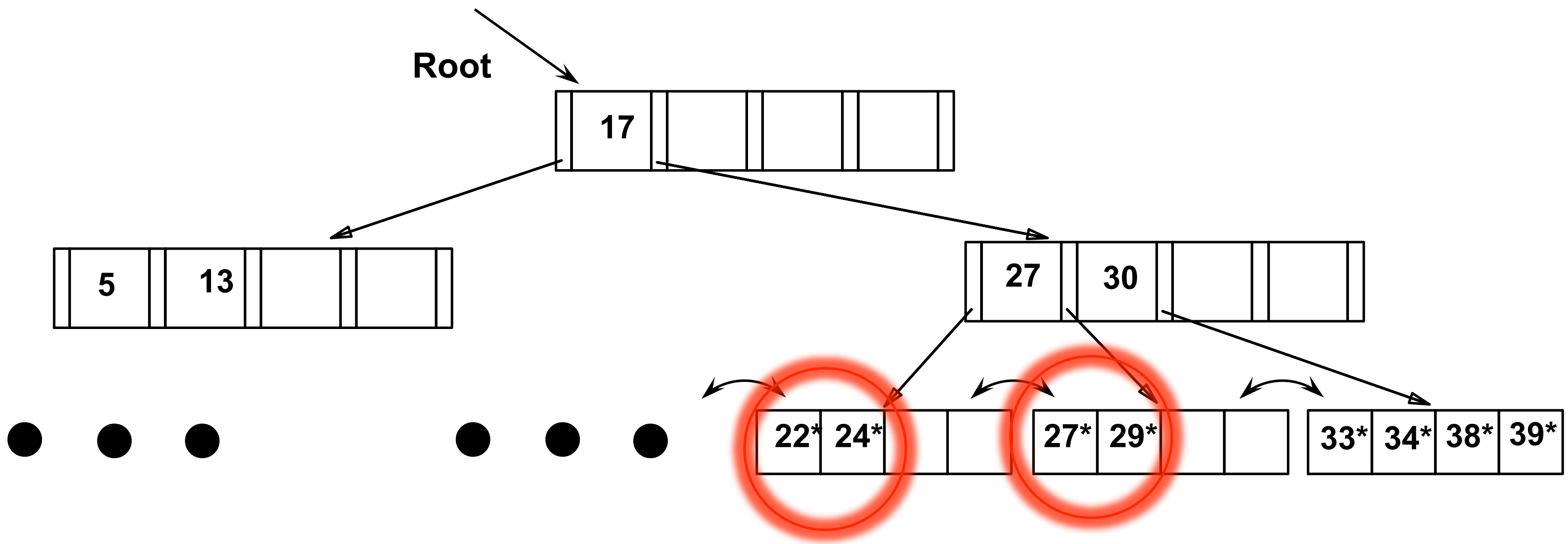


Redistribute

Example Tree – Delete 20*



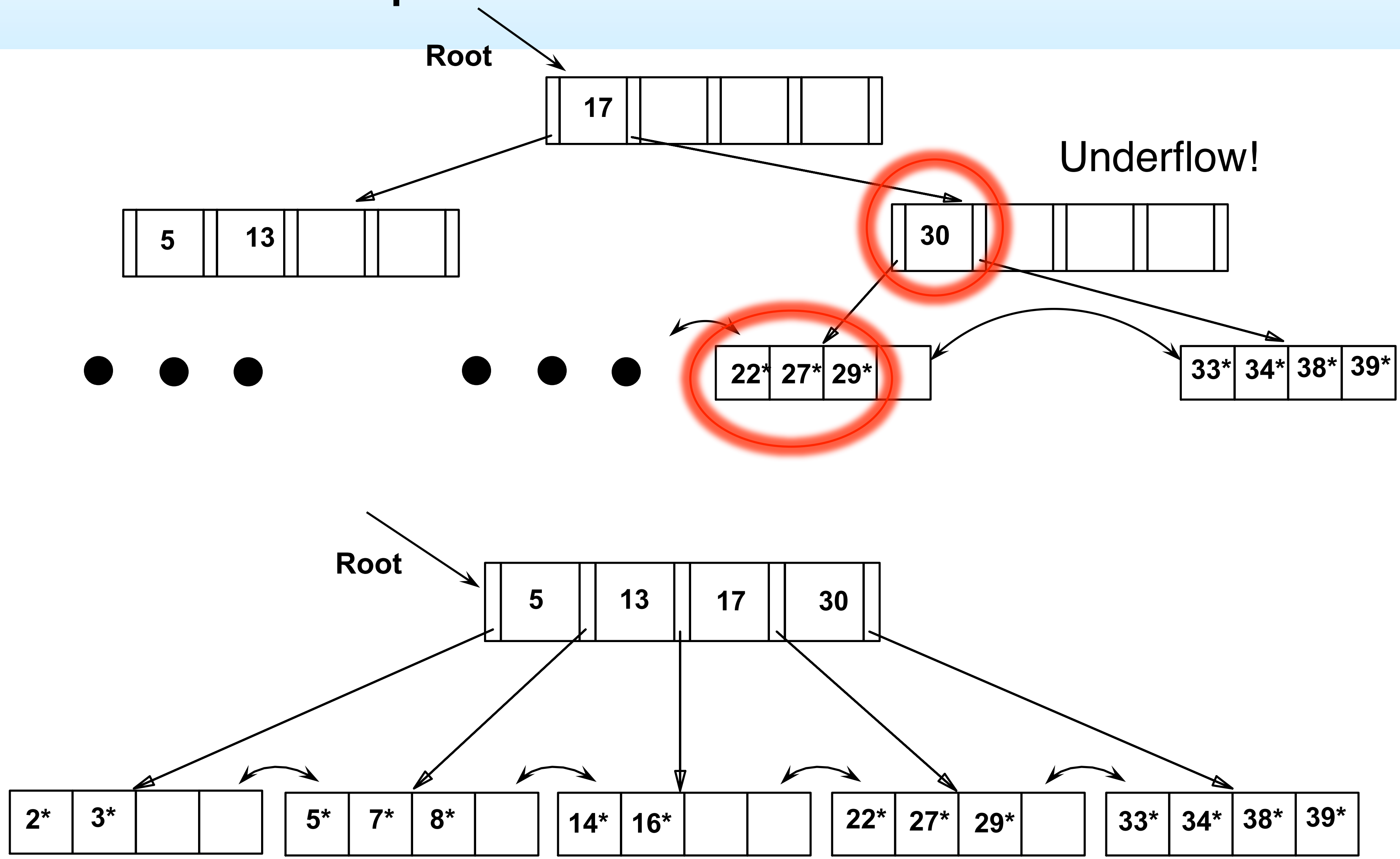
Example Tree – Then Delete 24*



Underflow!

Can't redistribute, must Merge…

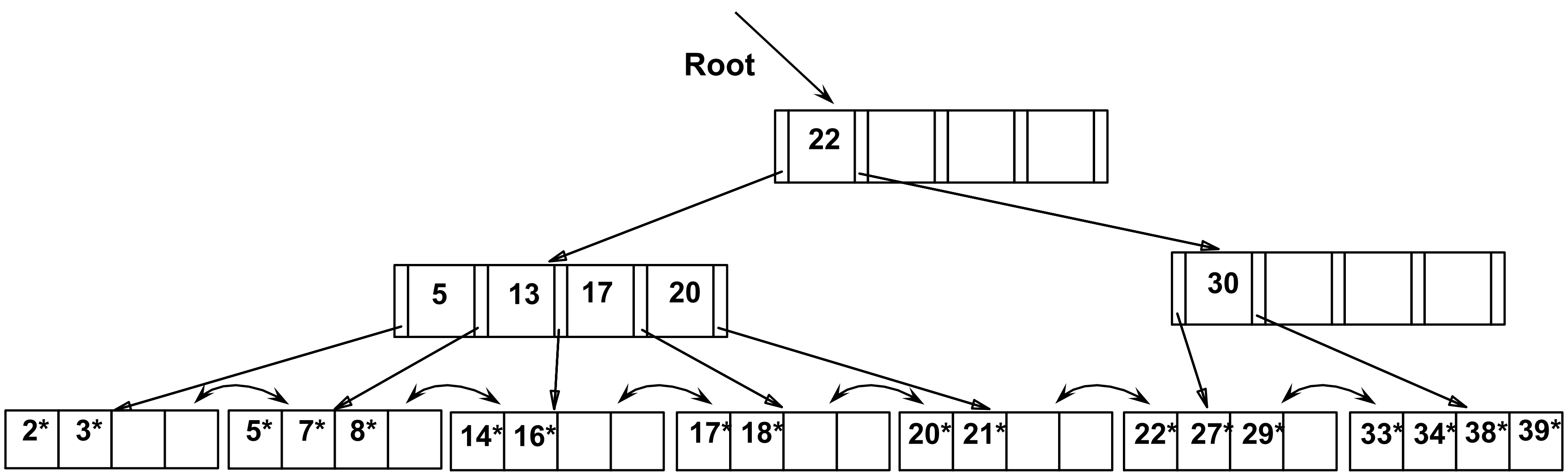
Example Tree – Delete 24*



Example of Non-leaf Re-distribution

- Tree is shown below *during deletion* of 24*. (What could be a possible initial tree?)

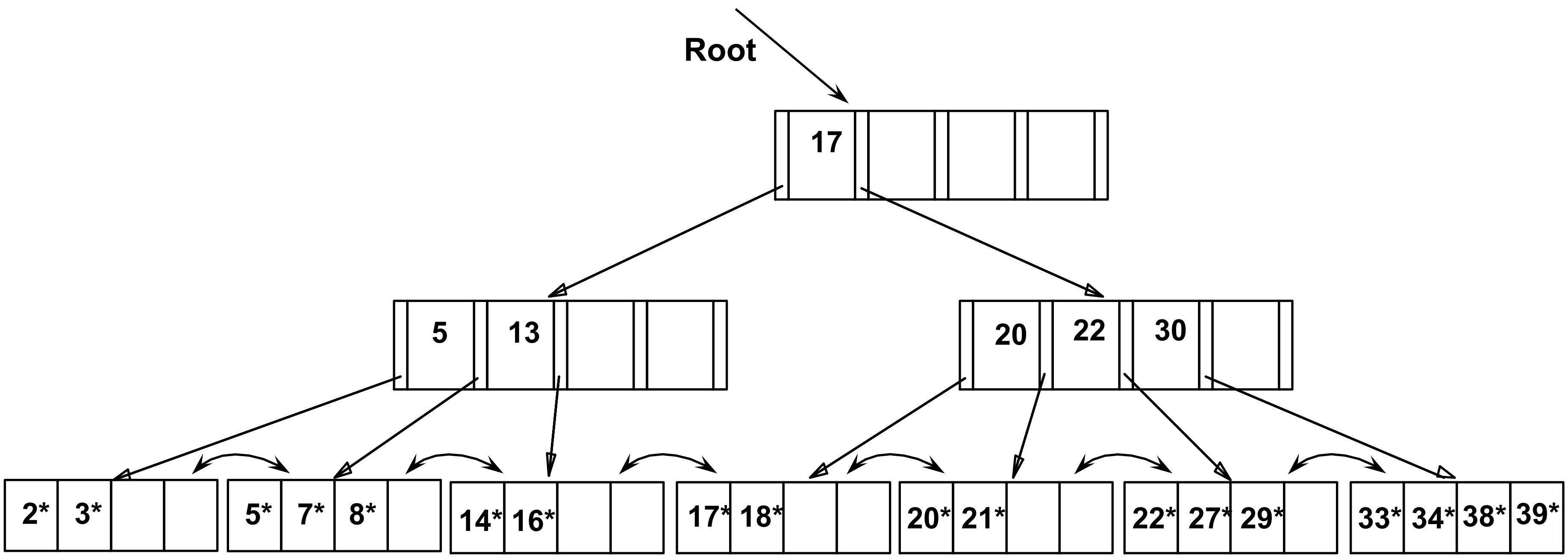
- In contrast to previous example, can re-distribute entry from left child of root to right child.



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.



A Note on `Order'

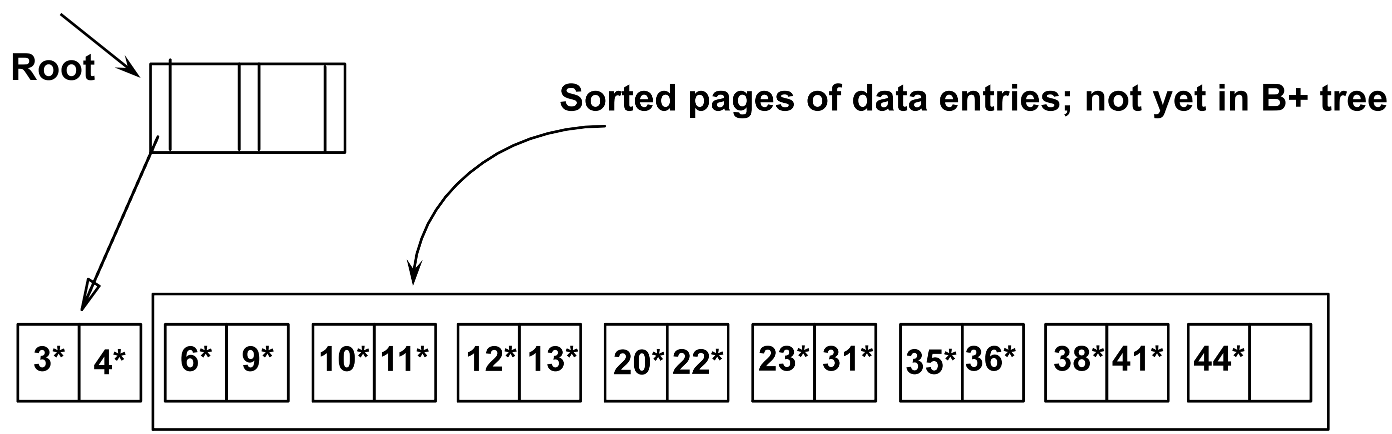
- *Order* (d) concept replaced by physical space criterion in practice (`*at least half-full*').
  - Index pages can typically hold many more entries than leaf pages.
  - Variable sized records and search keys mean different nodes will contain different numbers of entries.
  - Even with fixed length fields, multiple records with the same search key value (*duplicates*) can lead to variable-sized data entries (if we use Alternative (3)).
- Many real systems are even sloppier than this --- only reclaim space when a page is *completely* empty.

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.

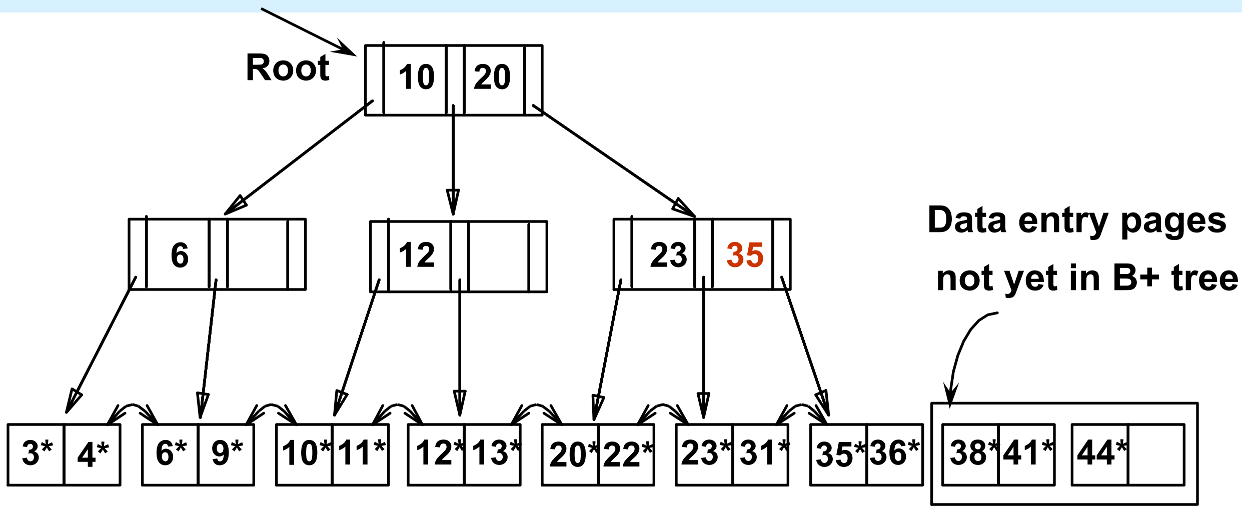
Bulk Loading of a B+ Tree

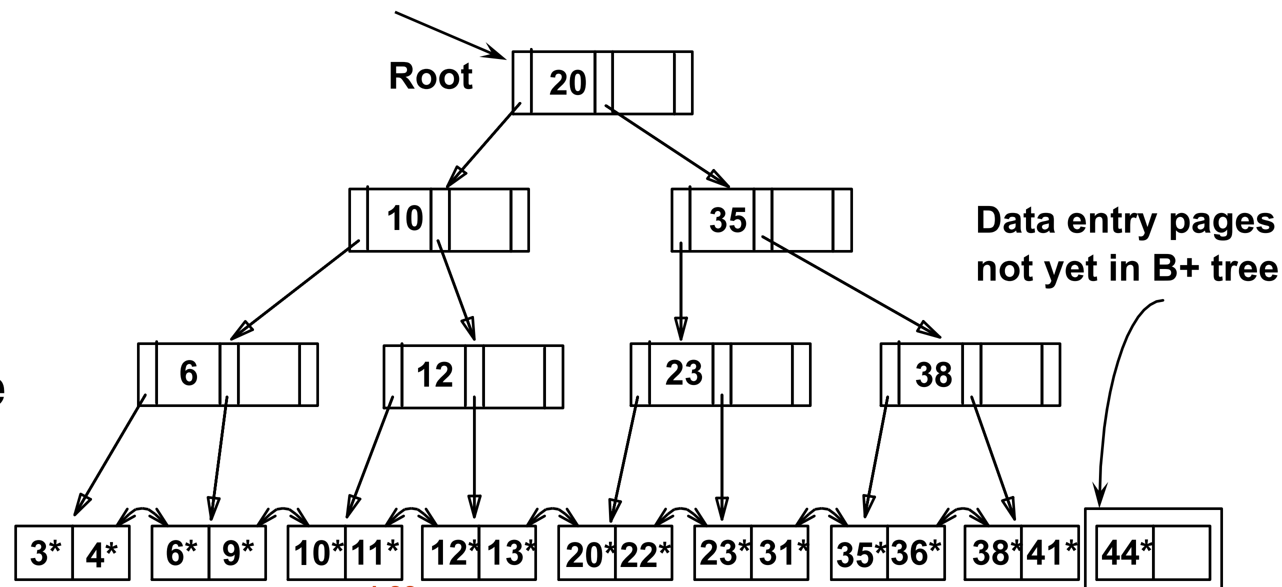
- If we have a large collection of records, and we want to create a B+ tree on some field, doing so by repeatedly inserting records is very slow.
  - Also leads to minimal leaf utilization --- why?
- *Bulk Loading* can be done much more efficiently.
- *Initialization*: Sort all data entries, insert pointer 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.

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; adjusts to growth gracefully.
- If data entries are records, splits can change rids!

- Other topics:
  - 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.