ECS 165B: Database System Implementation
Lecture 6

UC Davis
April 9, 2010

Acknowledgements: portions based on slides by Raghu Ramakrishnan and Johannes Gehrke.

Class Agenda

- Last time:
  - Record Manager cookbook session
- Today:
  - An announcement!
  - Dynamic aspects of B+ Trees
- Reading
  - Chapter 10 in Ramakrishan and Gehrke

  (or Chapter 12 in Silberschatz, Korth, and Sudarshan)

Announcements

EXTENSION:

Project Part 1 deadline pushed back 1 week

now due Sunday 4/18 @11:59pm

TO ACCOMMODATE EXTENSION:

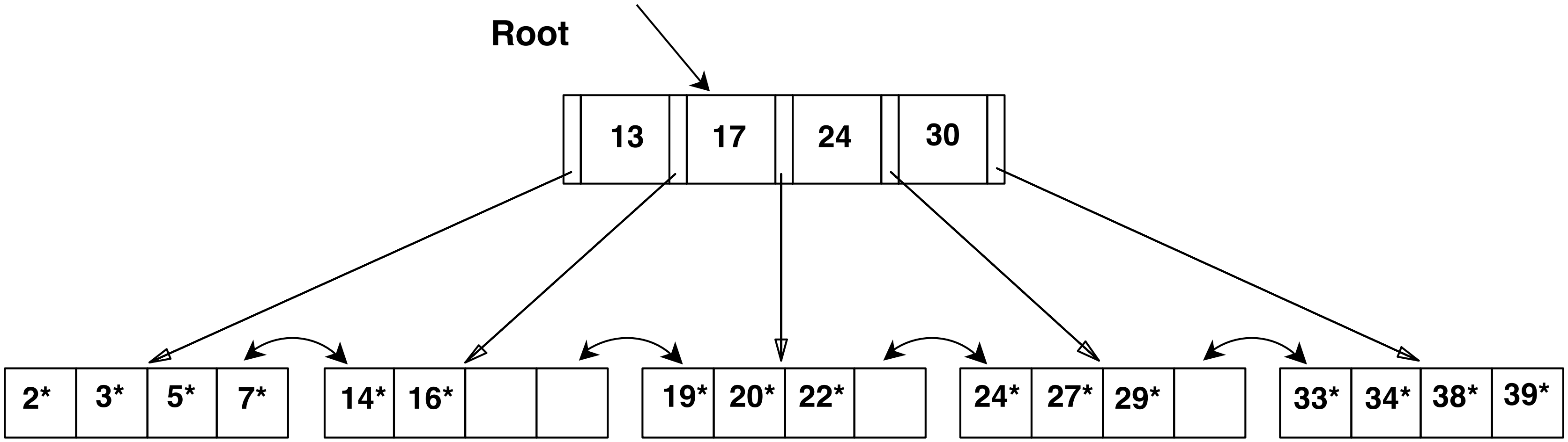
Deadlines for Parts 2-4 also pushed back 1 week

Project Part 5 cancelled

Dynamic Aspects of B+ Trees

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* ...

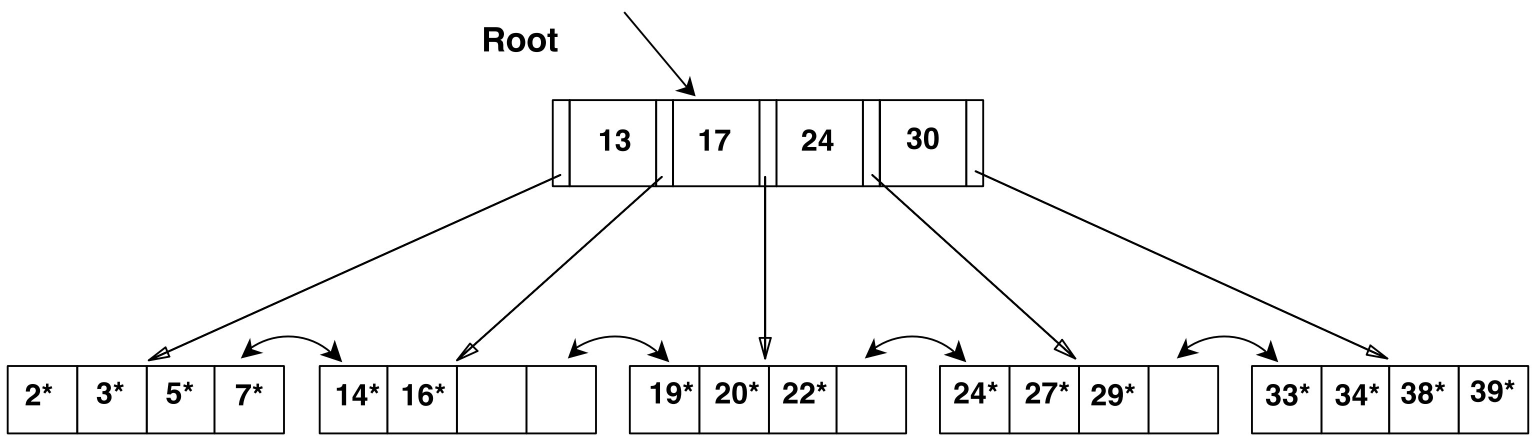


* *Based on the search for 15*, we <u>know</u> it is not in the tree!*

Inserting a Data Entry into a B+ Tree

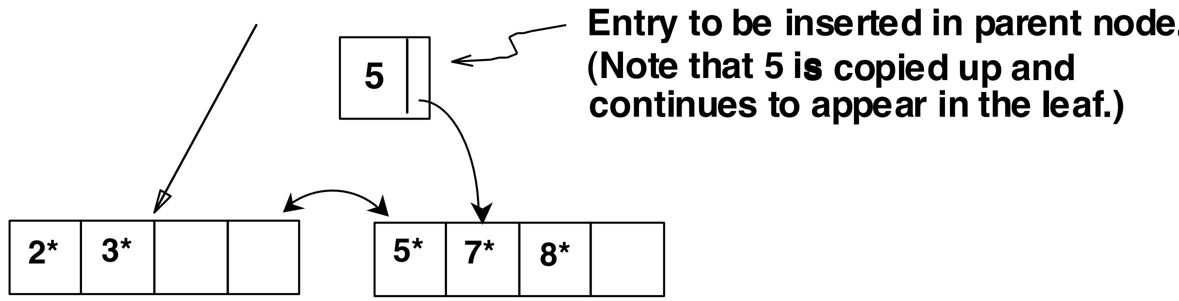
- Find correct leaf *L.*
- Put data entry onto *L*.
  - If *L* has enough space, *done*!
  - Else, must <u>*split*</u> *L (into L and a new node L2)*
    - Redistribute entries evenly, **<u>copy up</u>** middle key.
    - Insert index entry pointing to *L2* into parent of *L*.
- This can happen recursively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

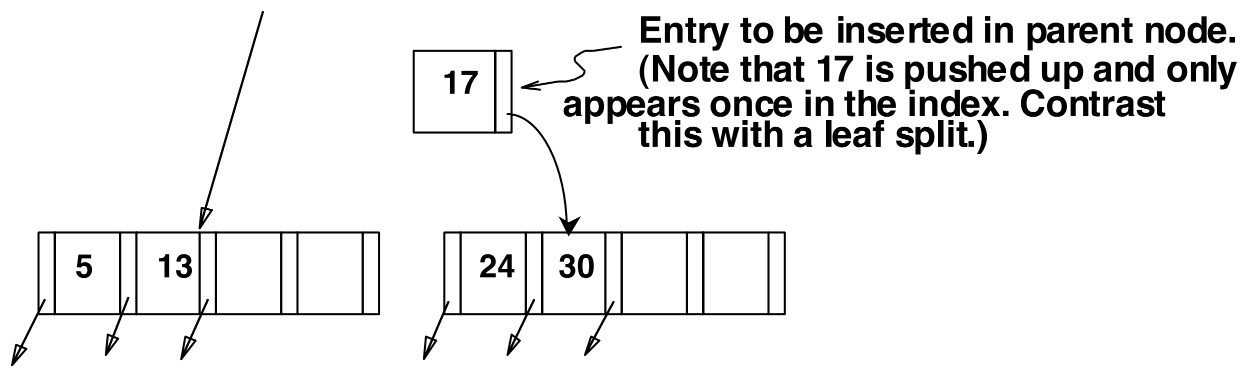
Example B+ Tree



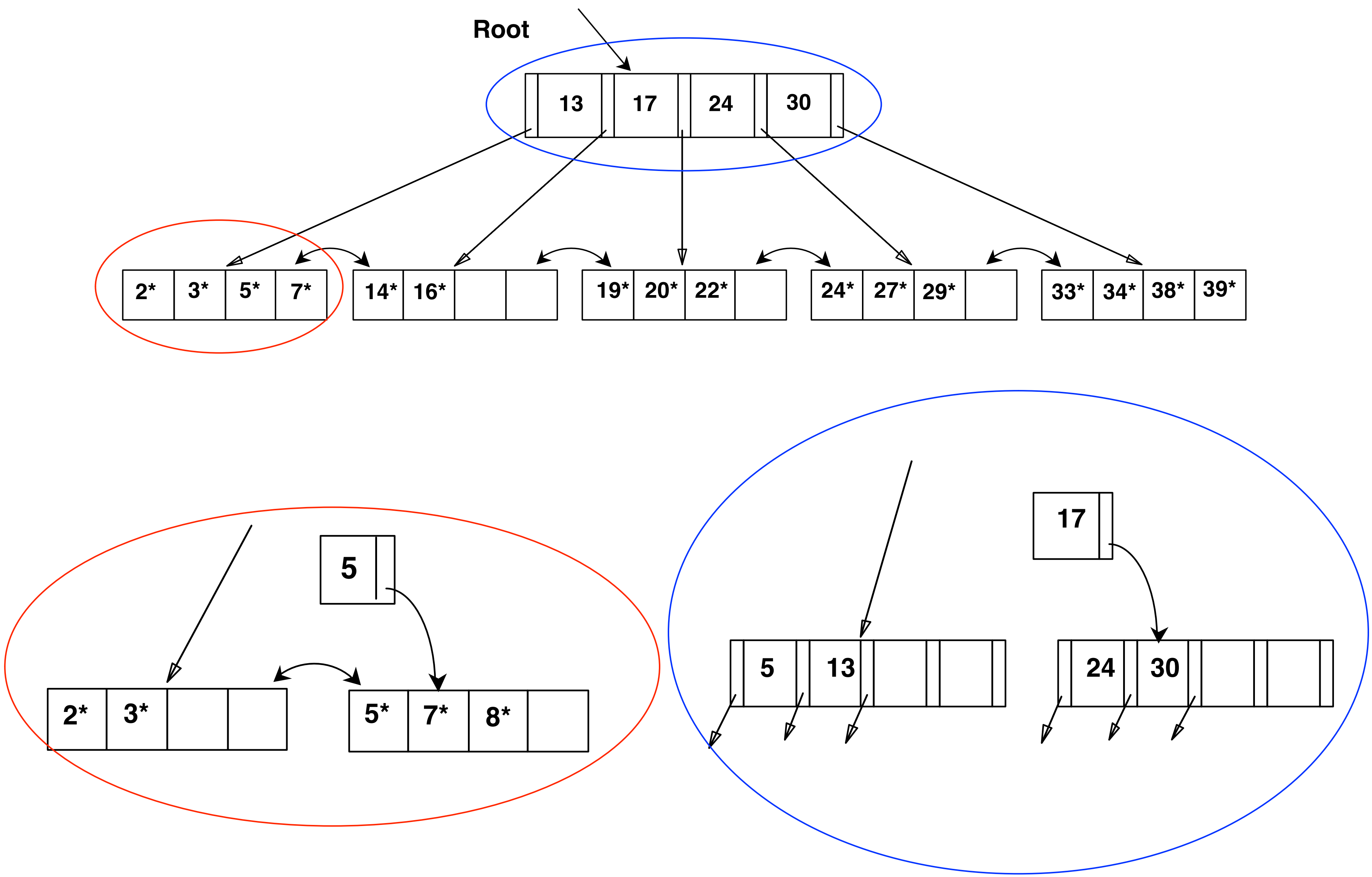
Inserting 8* into Example B+ Tree

- 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.

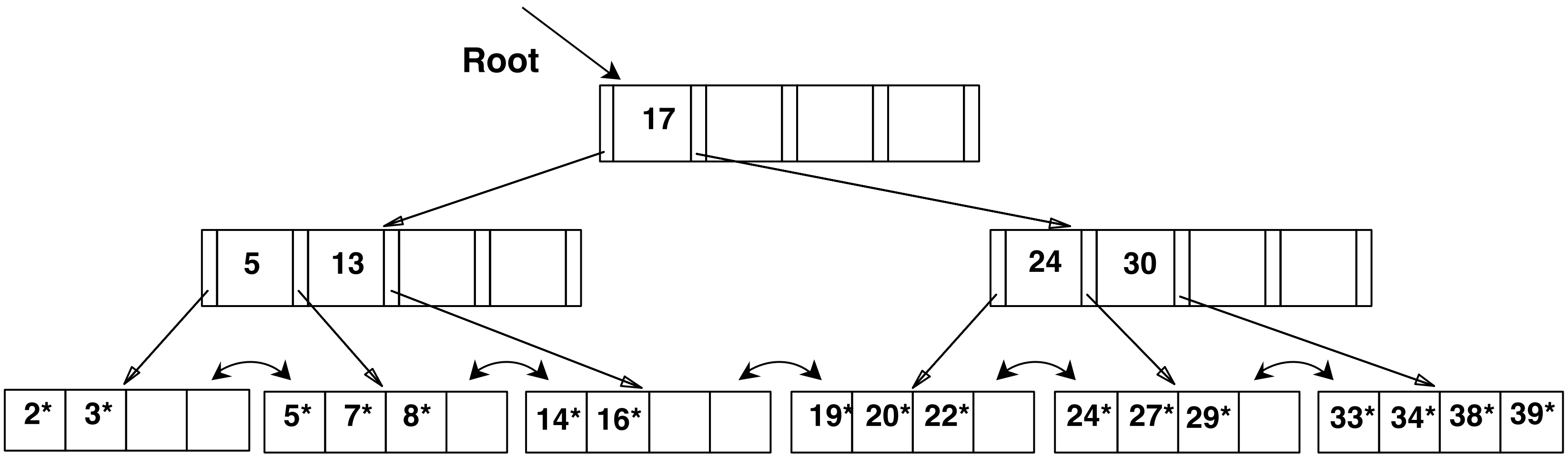




Inserting 8* Into Example B+ Tree



Example B+ Tree After Inserting 8*

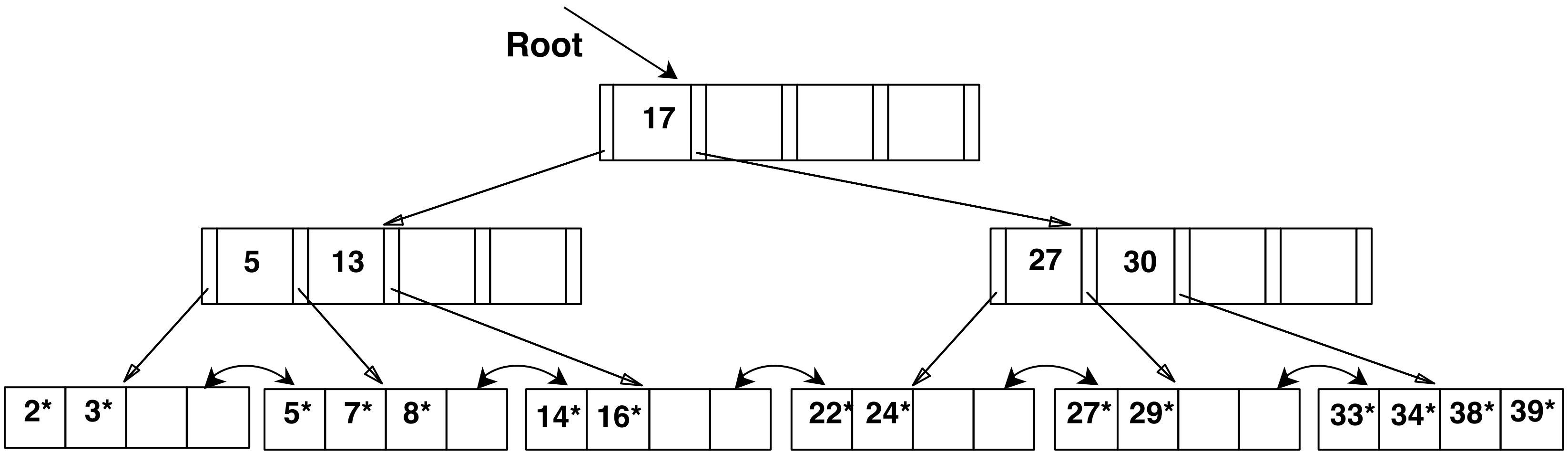


- Notice that root was split, leading to increase in height.
- In this example, we can avoid split by re-distributing entries; however, this is usually not done in practice.

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.

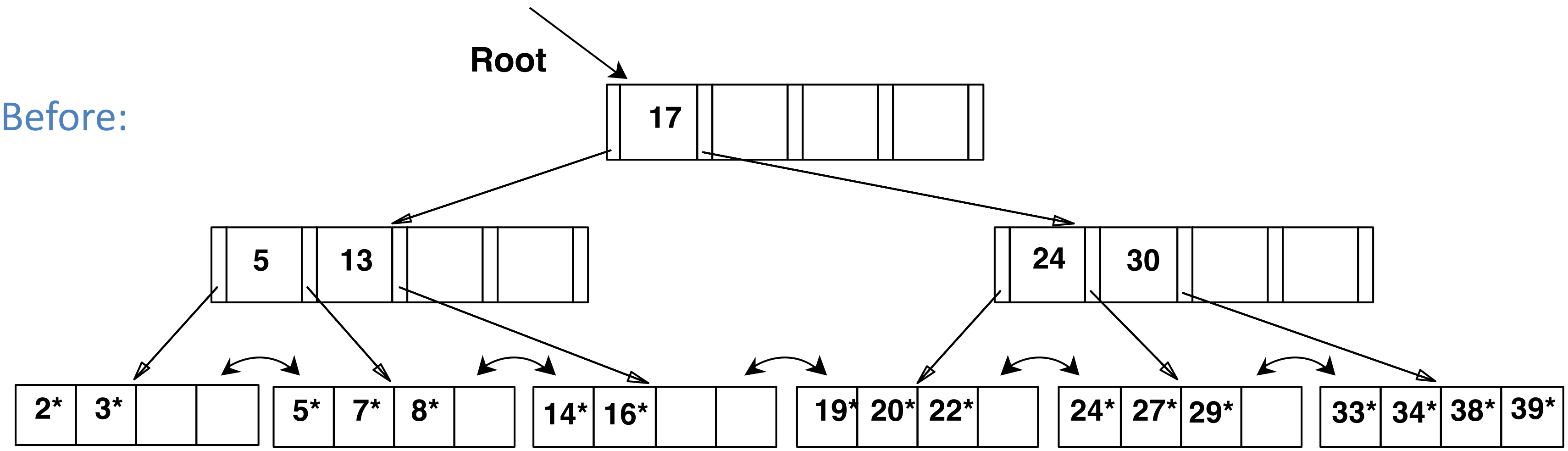
Example Tree After (Inserting 8, Then) Deleting 19* and 20* ...*



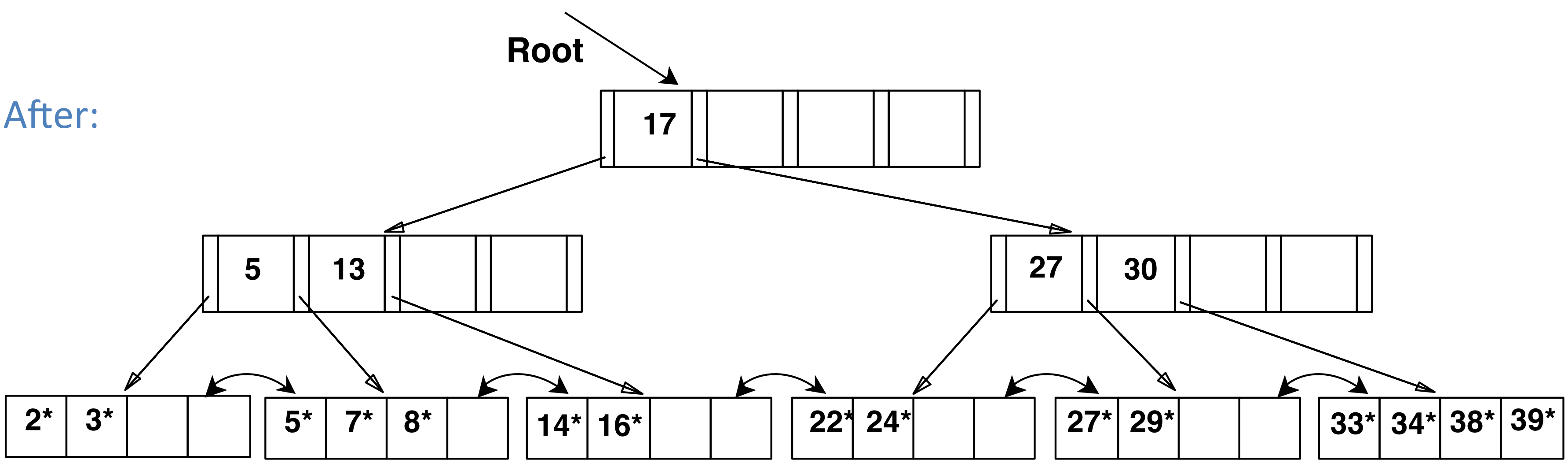
- Deleting 19* is easy.
- Deleting 20* is done with re-distribution. Notice how middle key is *copied up*.

Example Tree Before/After Deleting 19* and 20*

Before:

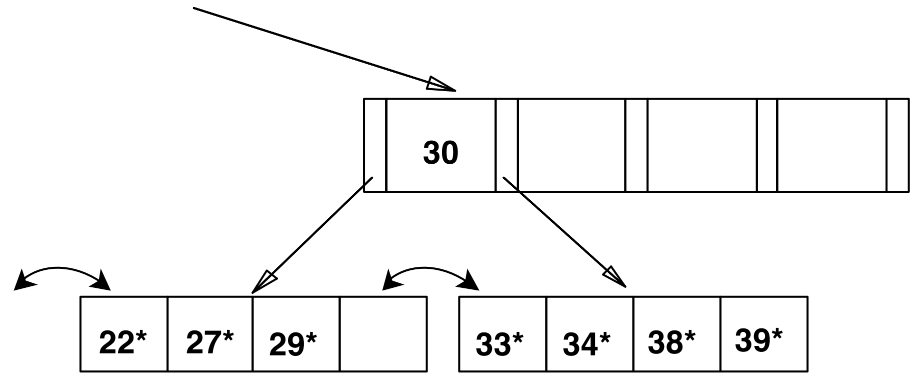


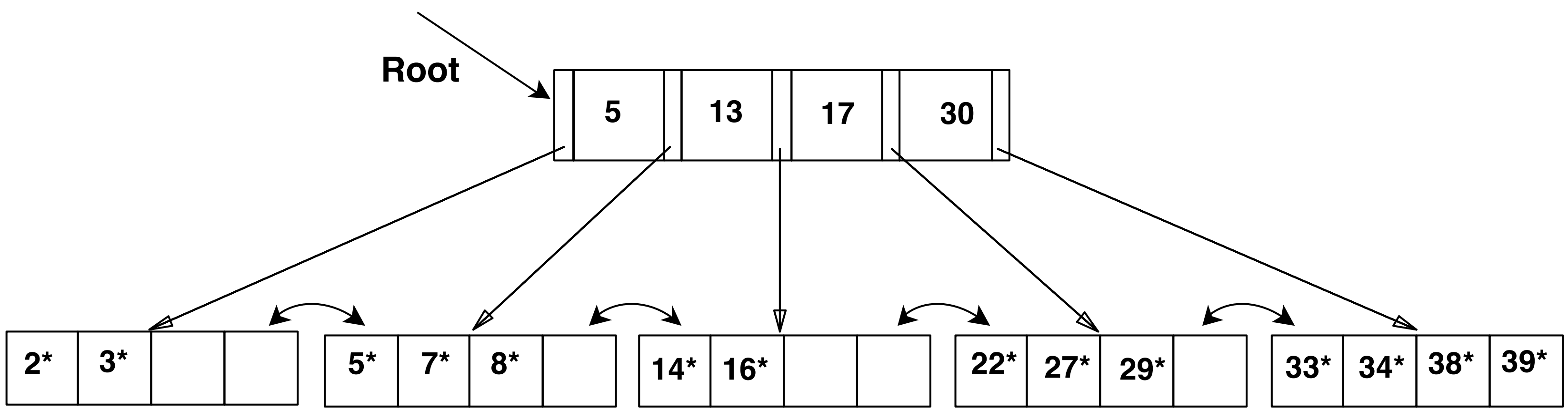
After:



... And Then Deleting 24*

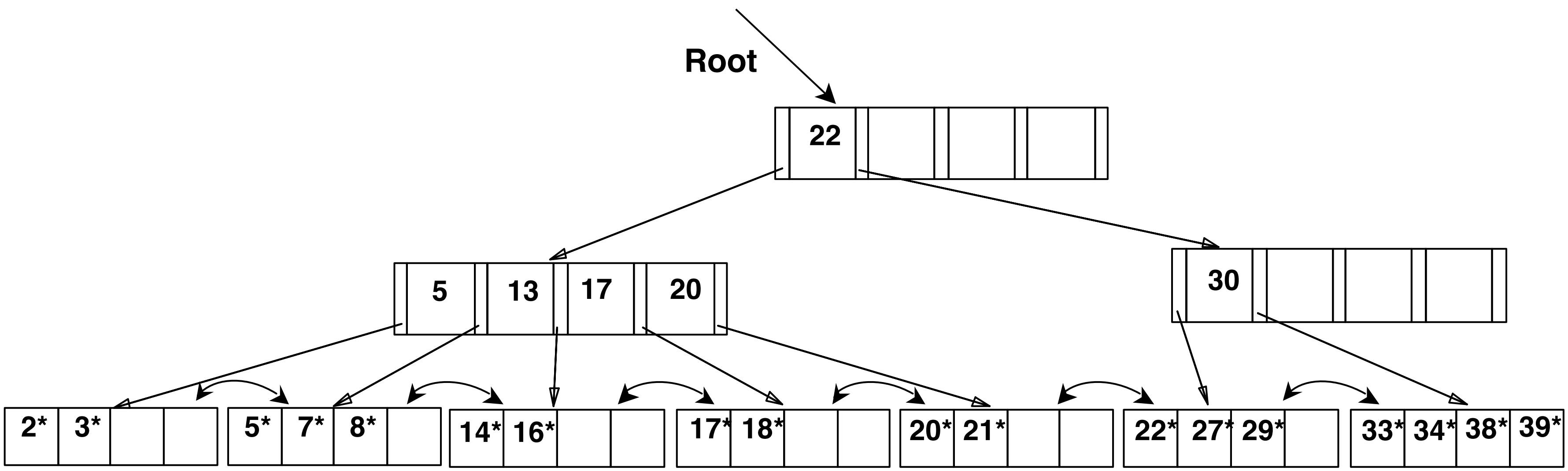
- Must merge.
- Observe `*toss*' of index entry (on right), and `*pull down*' of index entry (below).





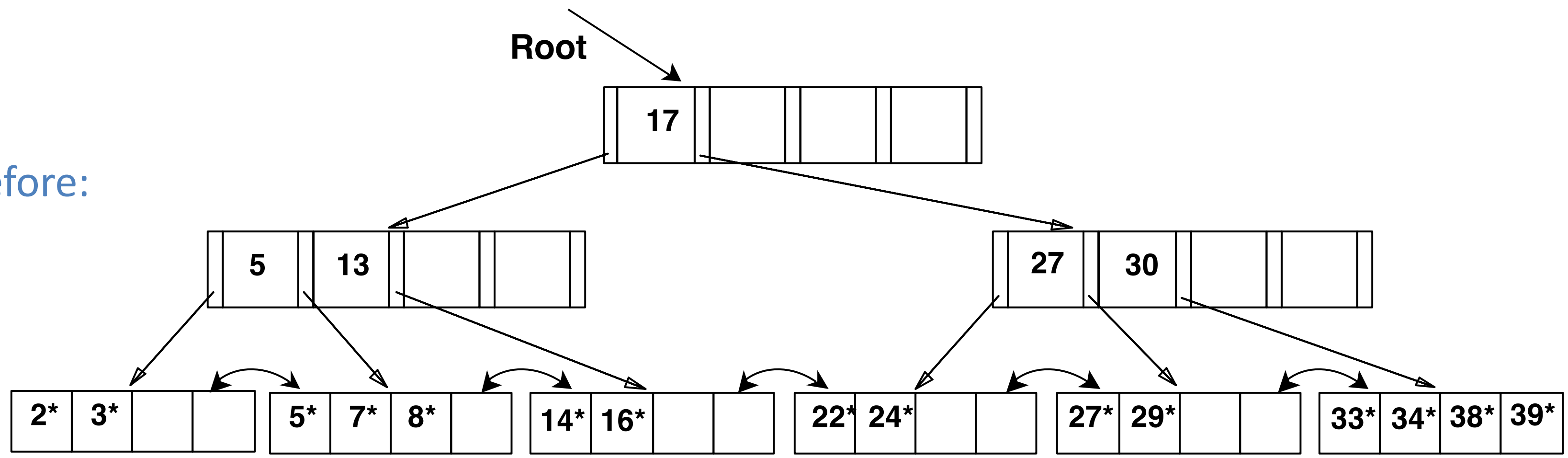
Example of Non-leaf Re-distribution

- 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.

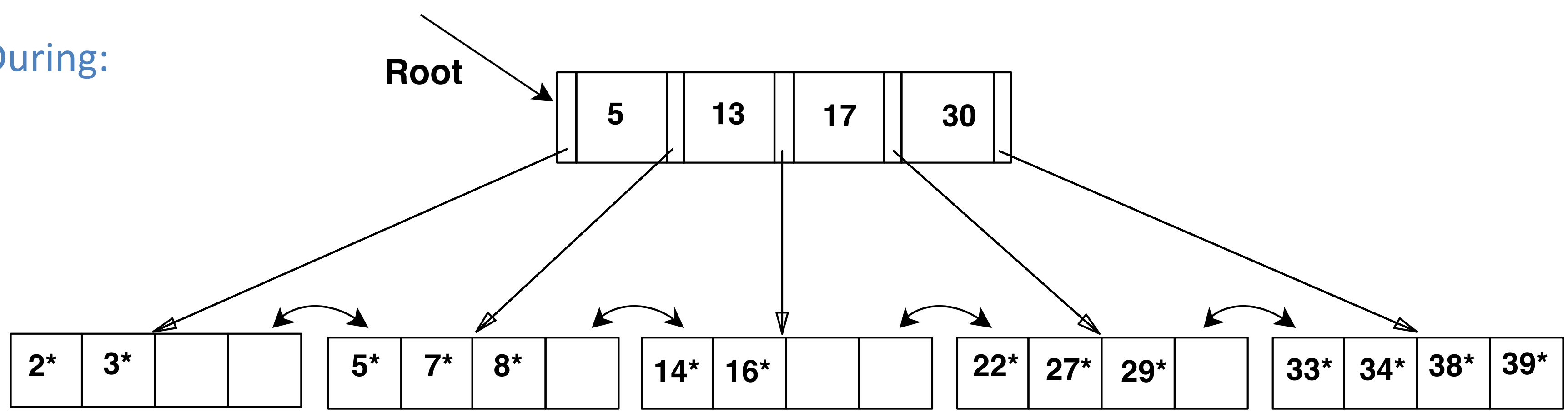


Before/After Deleting 24*

Before:

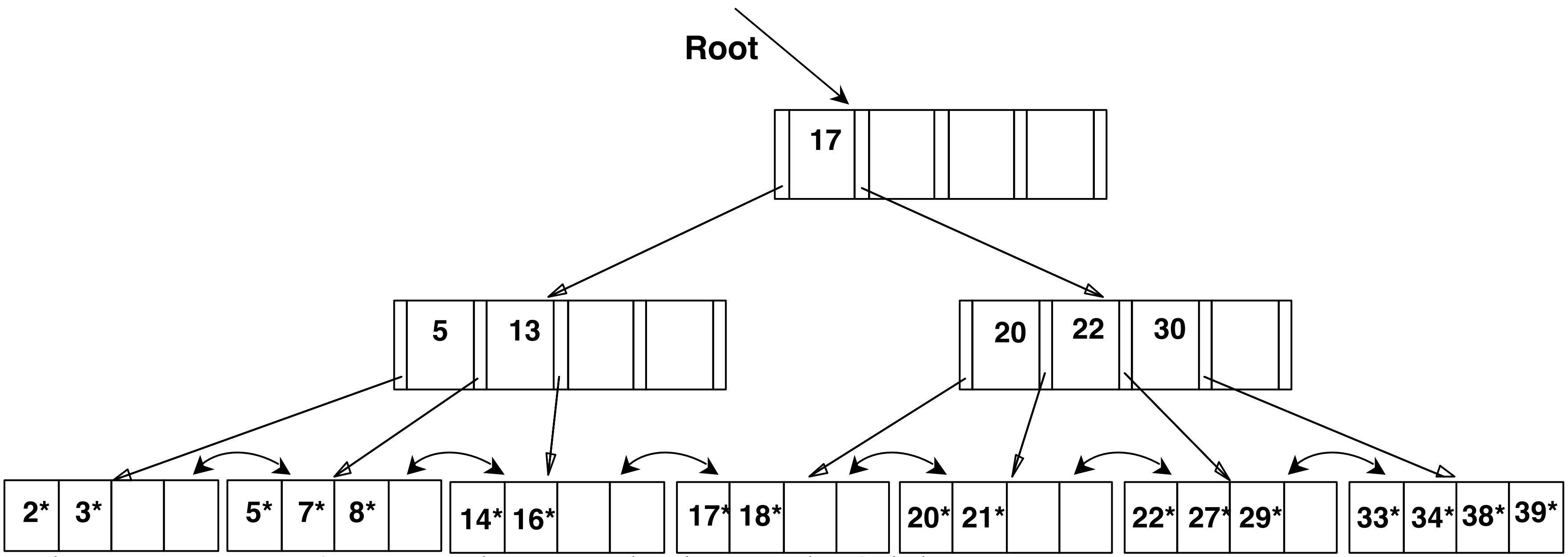


During:



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.



B+ Tree Deletion in DavisDB

- Standard deletion algorithm is tricky to implement (many corner cases)

- We'll use a simplified version of scheme: *lazy deletion*

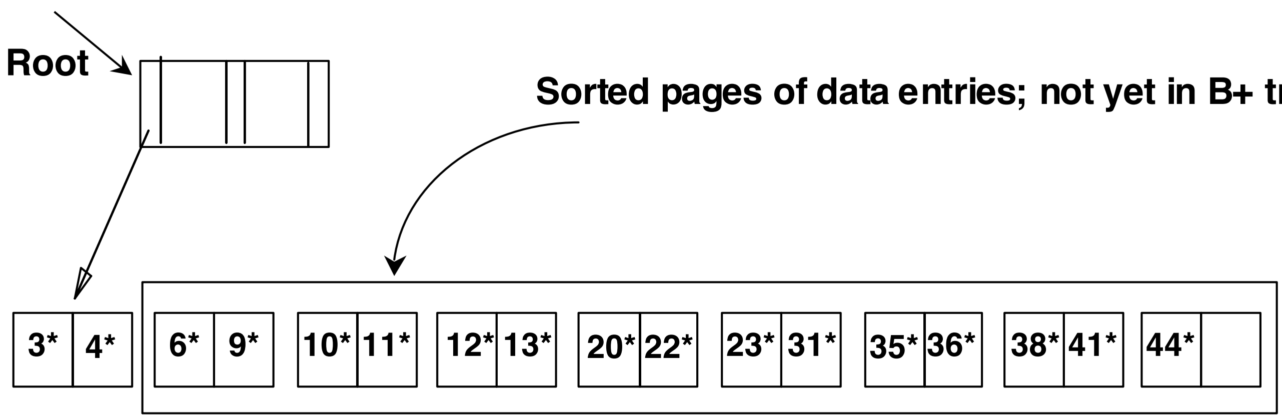
  – When entry is deleted, no redistribution or node merge even if leaf page < half full; underfull page remains in tree

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.

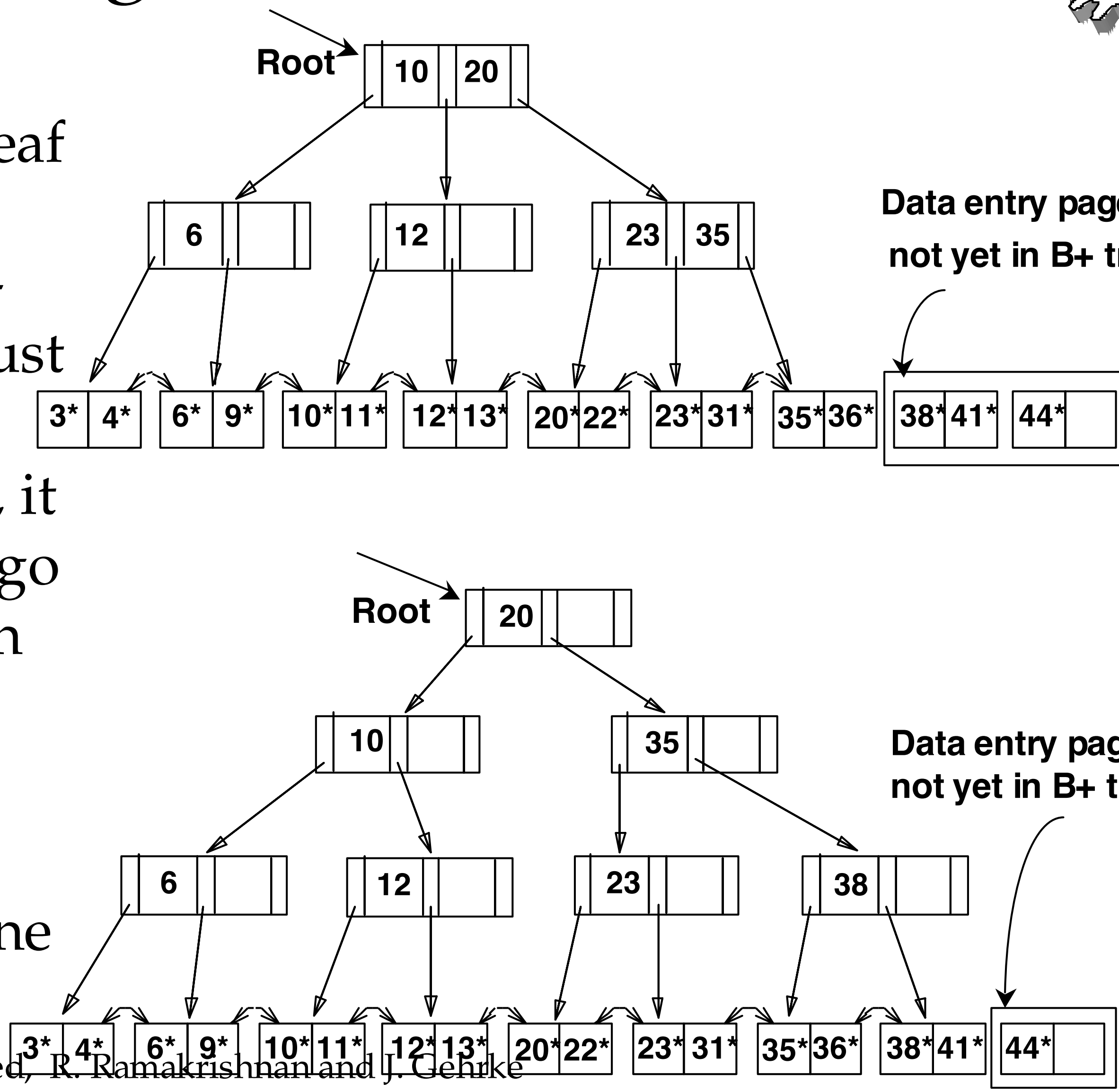
Bulk Loading of a B+ Tree

- 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.

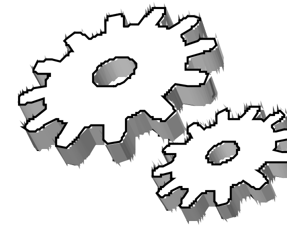


Bulk Loading (Contd.)

- 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: <u>*Bulk Loading*</u>
  - 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 differnt 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)).

Duplicate Keys

- Several data entries may have same key value; what if, e.g., there are too many to fit on a single leaf page?
- Solution 1 (rare): Use overflow leaf pages, as in ISAM
- Solution 2 (common): Use splitting as usual, allowing duplicate key values in index nodes
  - Range search: find **leftmost** data entry with given key value; scan
  - When record is deleted, have to scan all records with that key value (can be slow)
- Solution 3: expand key to include record id (rules out duplicates)
  - Fast deletion; but index takes more space

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.