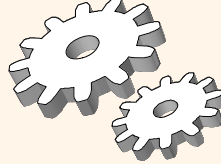
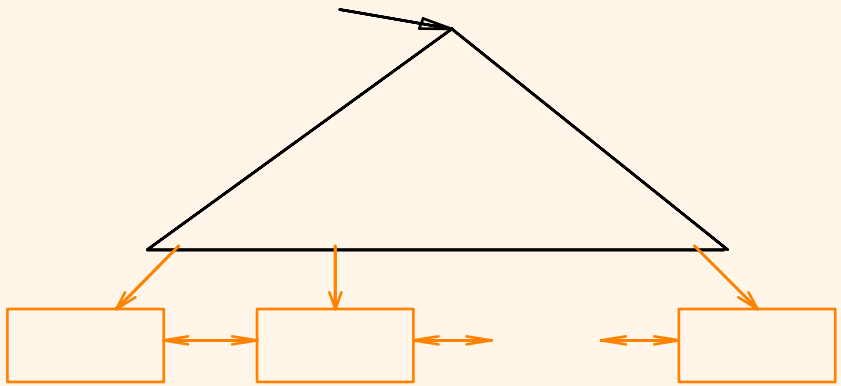


Tree-Structured Indexes

Yanlei Diao
UMass Amherst
Feb 28, 2007

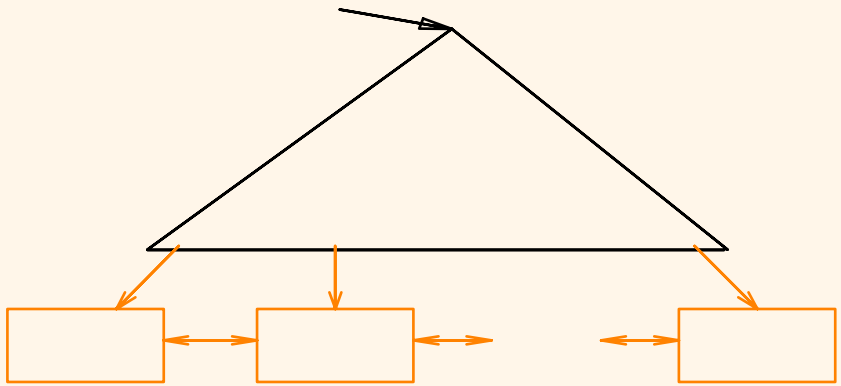
B+ Tree: Most Widely Used Index

- Height-balanced given arbitrary inserts/deletes.
  - *Fanout*: # child pointers of a non-leaf node
    F = avg. fanout
  - *Height*: N = # leaf pages
    $H = \text{Log}_F N$
    (Root: level 0, …, Leaf: level H)



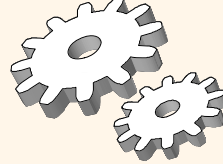
B+ Tree: Most Widely Used Index

- Minimum 50% occupancy (except for the root).
  - *Order* of the tree (n): max # of keys in a node.
    - Can be computed using the node size, key size, pointer size.
  - Each non-root node contains [⌈n/2⌉, n] entries, i.e., at least half full.
  - Root node can have [1, n] entries.



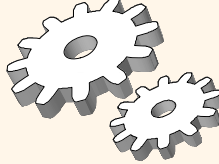
Index Entries
(Direct search)

Data Entries
("Sequence set")



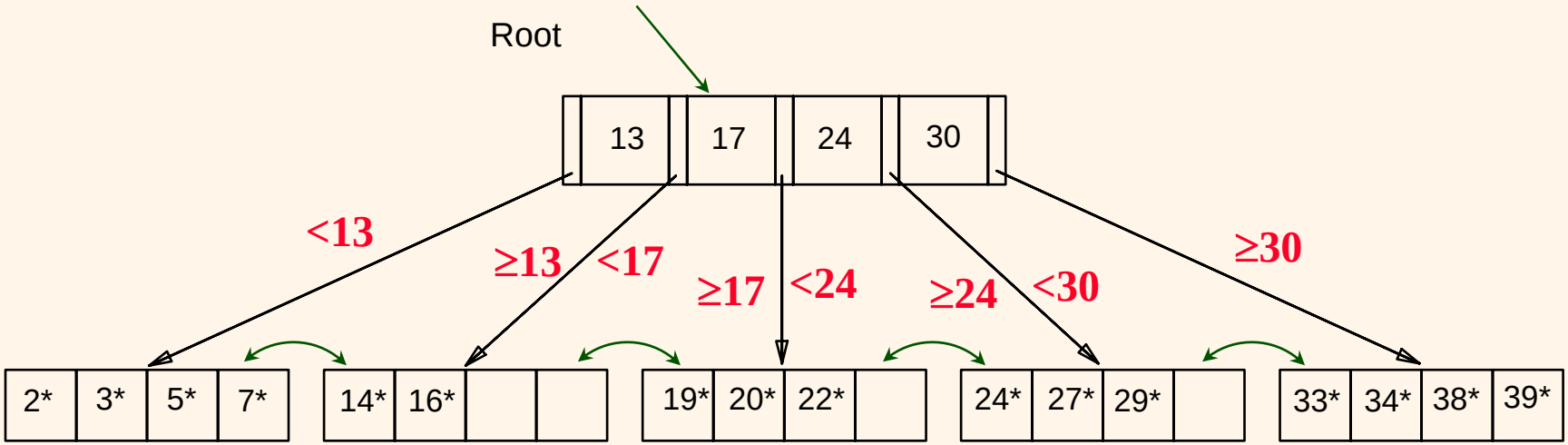
B+ Trees in Practice

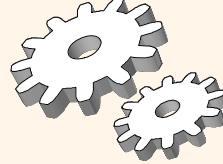
- Typical order is 200. Typical fill-factor (occupancy) is 67%.
  - Average fanout = 133
  - Level 0=1 page; Level 1=133 pages; Level 2=133^2 pages...
- Typical capacities:
  - Height 3: 133^3 = 2,352,637 records
  - Height 4: 133^4 = 312,900,700 records



Searches in a B+ Tree

- Search begins at root, and key comparisons direct it to a leaf.
- Search for 5*, 15*, all data entries >= 24* ...



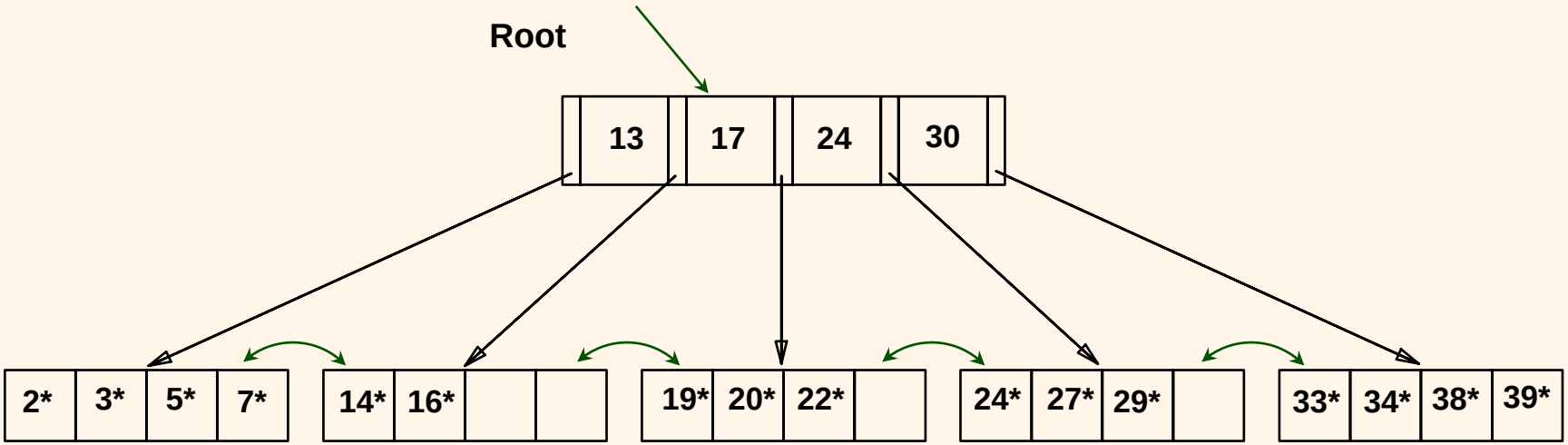


Inserting a Data Entry into a B+ Tree

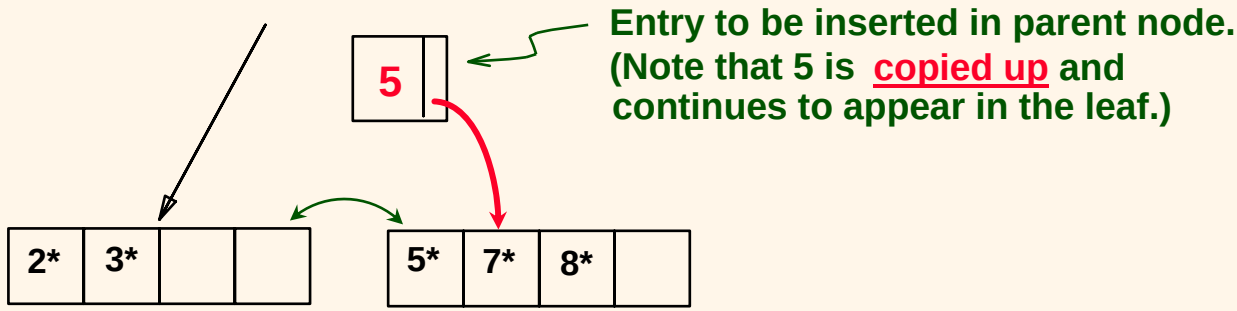
- Find correct leaf *L* via a top-down search.
- Put data entry onto *L*.
  - If *L* has enough space, *done*!
  - Else, must *split* *L (into L and a new node L2)*
    - Redistribute entries evenly, ***copy up*** middle key *k*, insert (*k*, pointer to *L2)* into parent of *L*.
  - Splitting can happen recursively to non-leaf nodes
    - Redistribute entries evenly, but ***push up*** middle key. (Contrast with leaf splits.)
- Splits "grow" the tree!
  - First *wider*, then *one level taller* when the root splits.

Previous Example

Inserting 8*

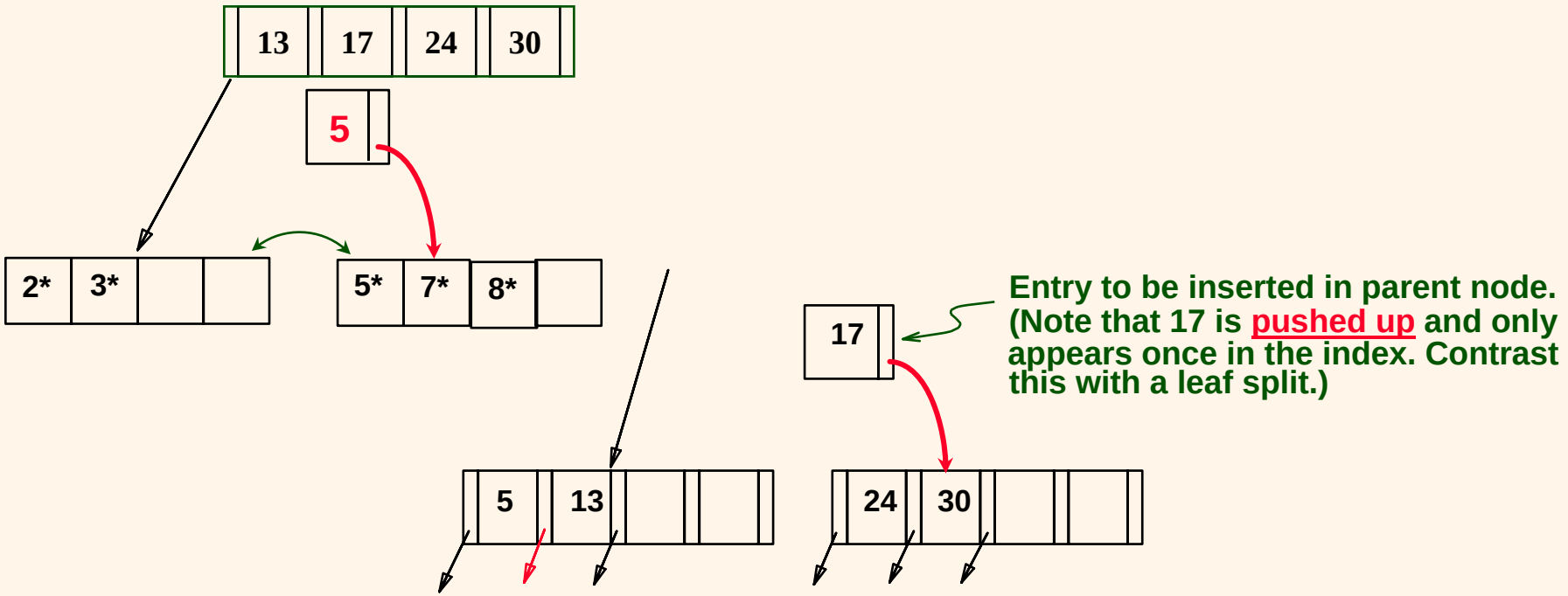


Inserting 8 into Example B+ Tree*



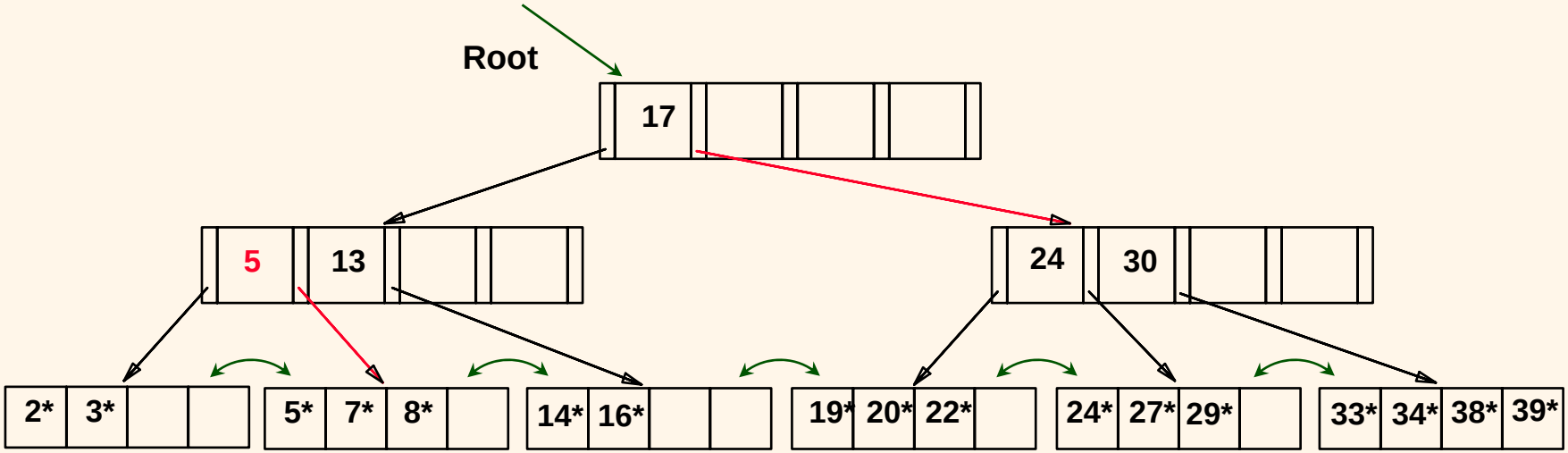
- Minimum occupancy is guaranteed in node splits.
- Copy up: key value of an inserted entry must appear in a leaf node!

Inserting 8 into Example B+ Tree*

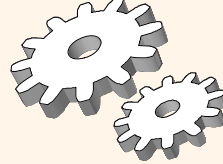


- Note difference between *copy-up* and *push-up*. Reasons?
- Push up: Any key value can appear at most once in non-leaf nodes of the tree!

*Example B+ Tree After Inserting 8**



- Root was split, leading to increase in height!



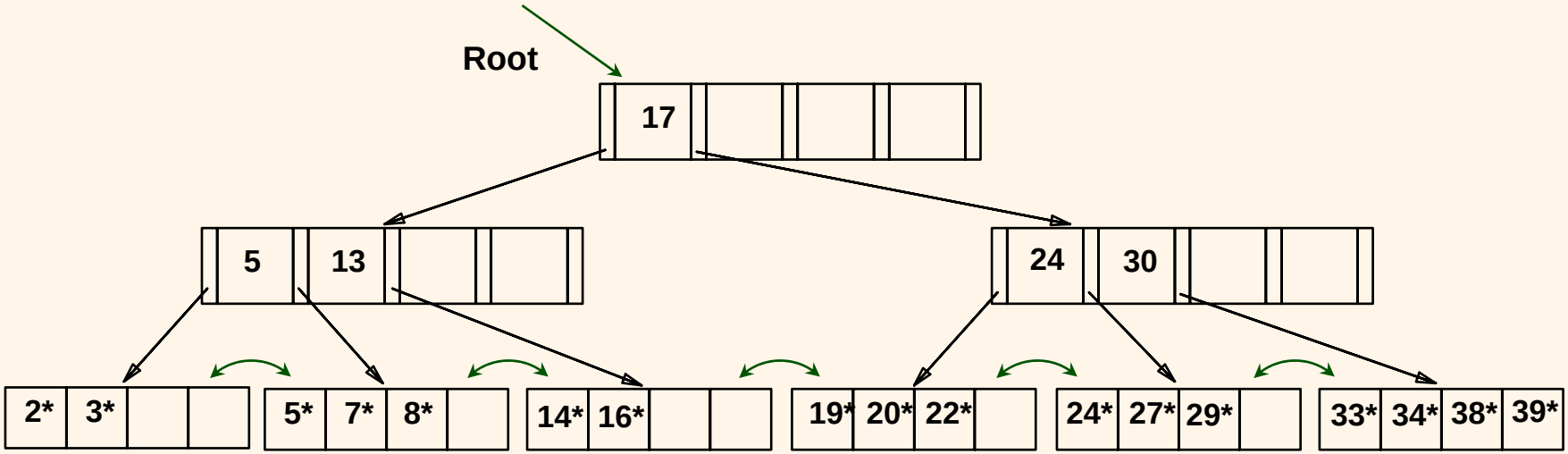
Deleting a Data Entry from a B+ Tree

- Start at root, find leaf *L* where entry belongs.
- Remove the entry.
  - If L is at least half-full, *done!*
  - If L has only $\lceil n/2 \rceil - 1$ entries,
    - Try to *re-distribute*, borrowing from *sibling* (*adjacent node with same parent as L*).
    - If re-distribution fails, *merge* *L* and sibling. Must delete index entry (pointing to *L* or sibling) from parent of *L*.
- Merge could propagate to root, decreasing height.

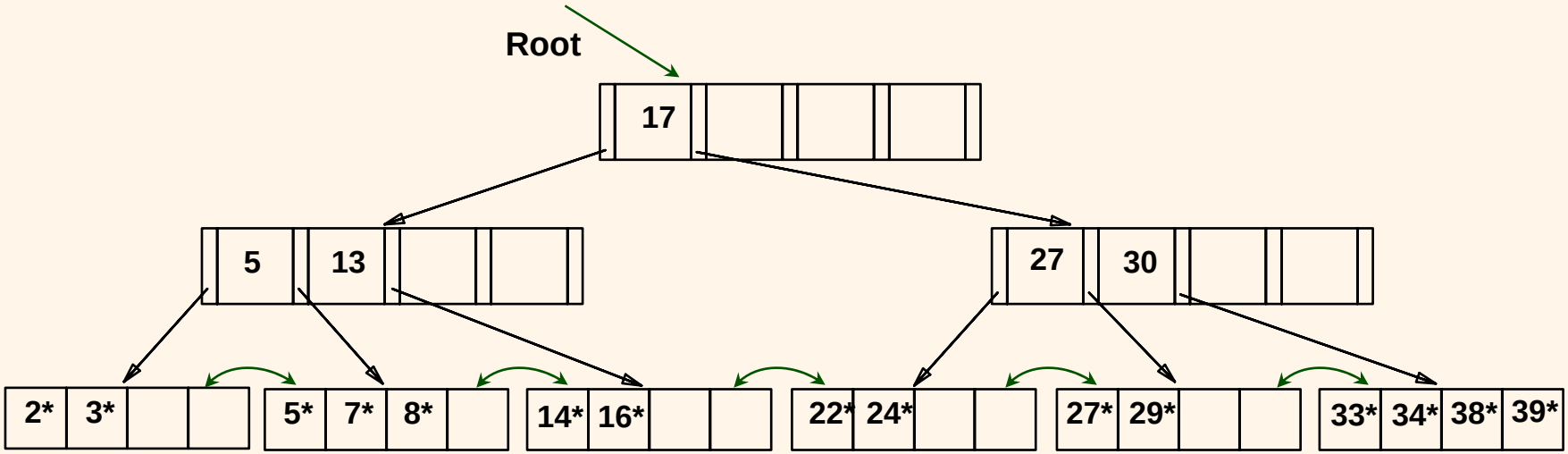
Current B+ Tree

Delete 19*

Delete 20*



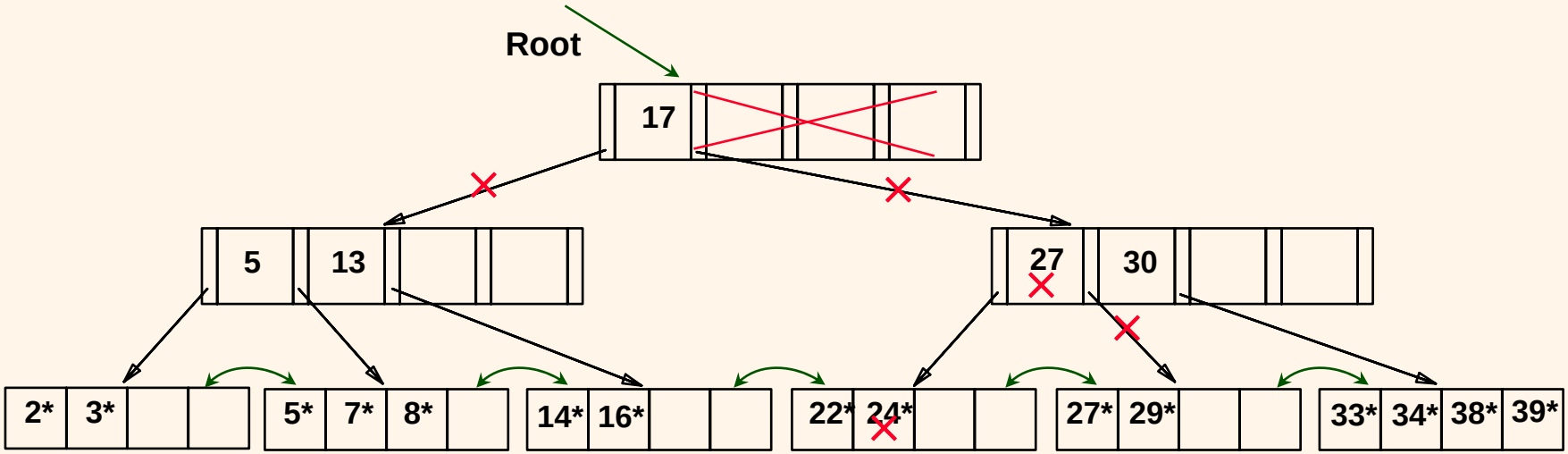
Example Tree After Deleting 19 and 20* ...*



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

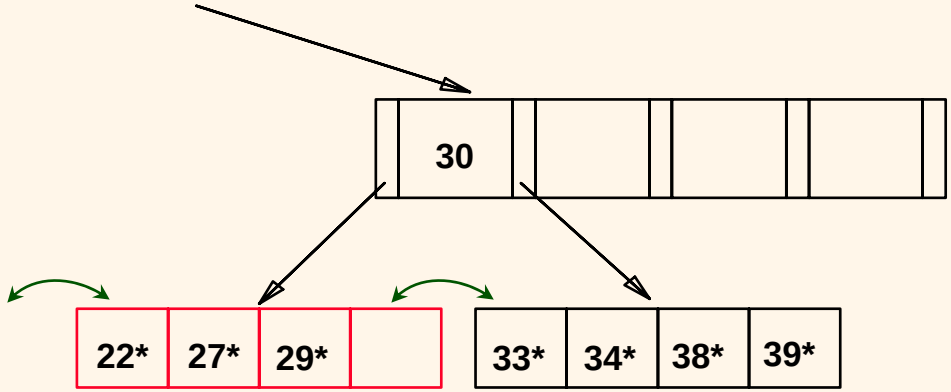
New B+ Tree ...

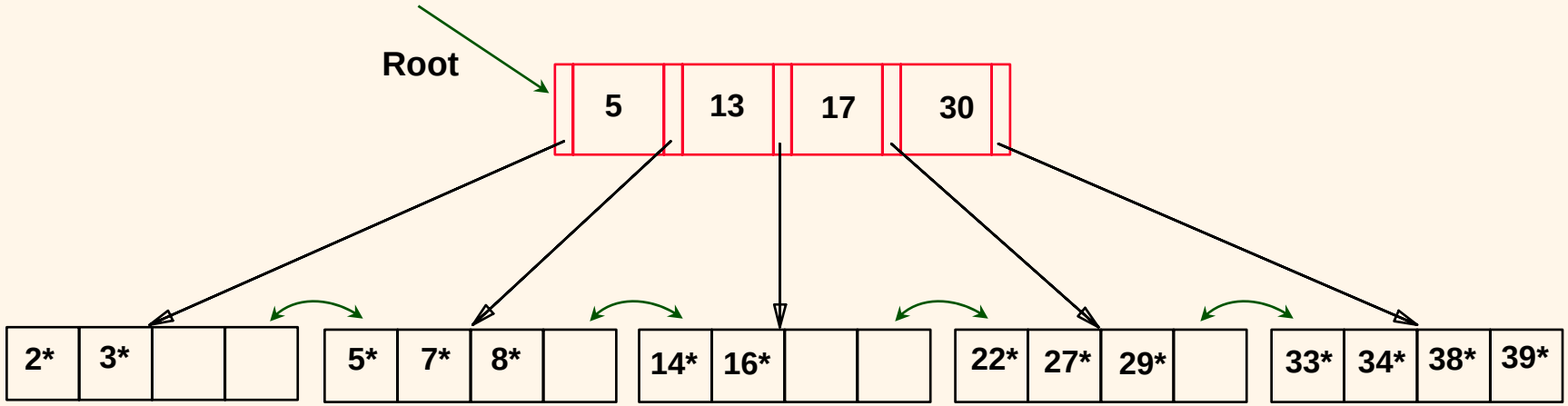
Delete 24*



... And Then Deleting 24*

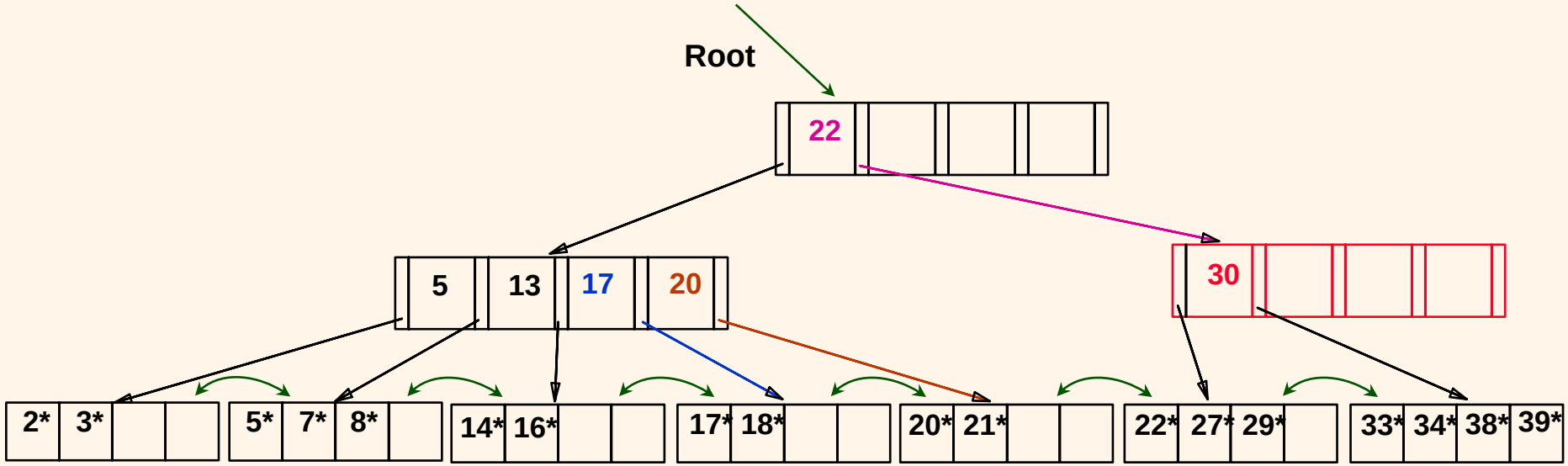
- Must merge nodes.
- *Toss* index entry (right)
- *Pull down* of index entry (below).





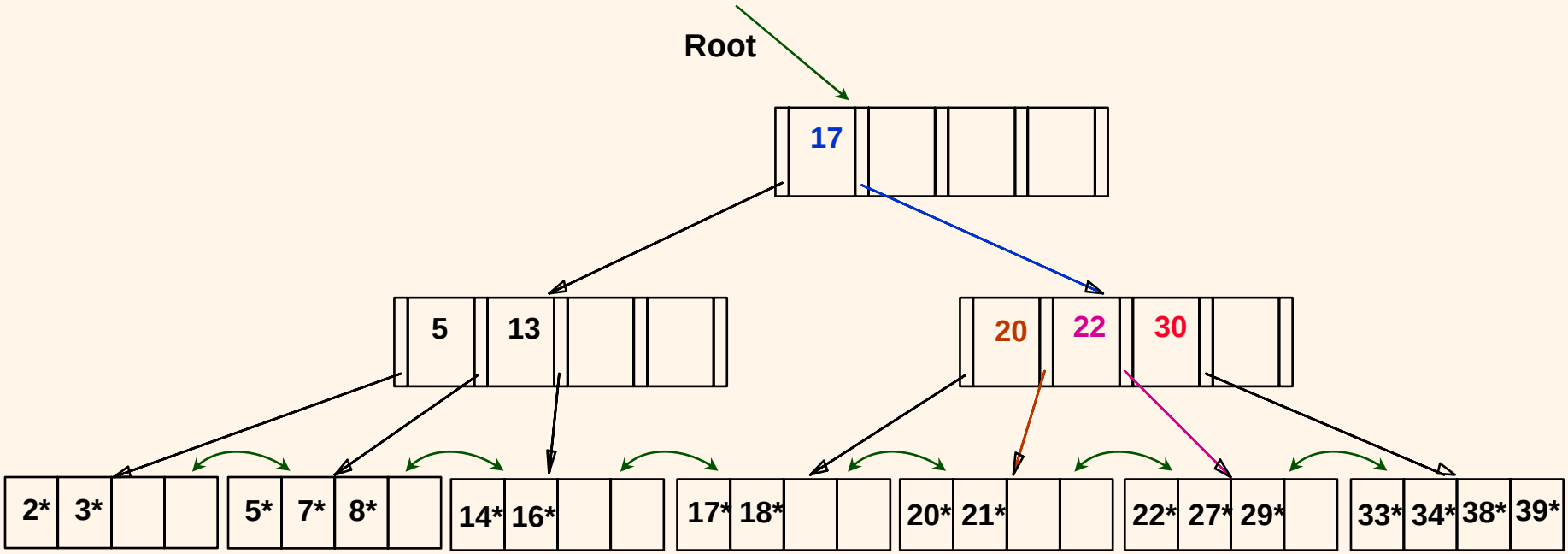
Example of Non-leaf Re-distribution

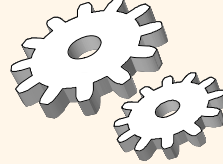
- Tree is shown below *during deletion* of 24*. (What could be a possible initial tree?)
- In contrast to previous example, can re-distribute entry from left child of root to right child.



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.





Summary

- Tree-structured indexes are ideal for range-searches, also good for equality searches.
- 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.
  - 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!