Indexing

Some slide content courtesy of Raghu Ramakrishnan

Indexing

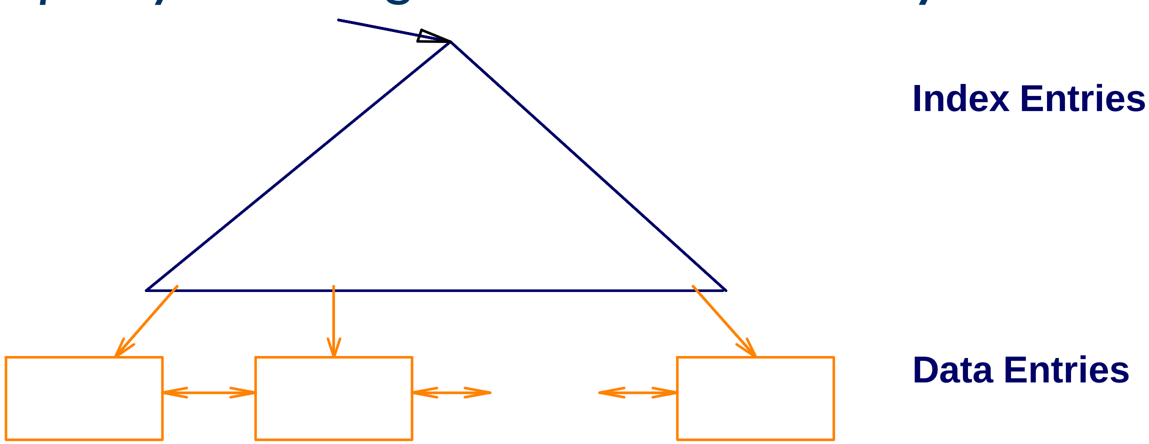
- An *index* on a file speeds up selections on the *search key attributes* for the index (trade space for speed).
  - Any subset of the fields of a relation can be the search key for an index on the relation.
  - *Search key* is not the same as *key* (minimal set of fields that uniquely identify a record in a relation).
- An index contains a collection of *data entries*, and supports efficient retrieval of all data entries **k*** with a given key value **k**.

B+ Tree: The DB World's Favorite Index

- An index is great, but must be able to adapt "gracefully" as the file changes
  - Insert, delete
  - Most files grow in size over time
- Tree-structured/Hash indexes can degrade to a linear search
  - B+ tree avoids this by "adapting" to updates and guaranteeing logarithmic search time

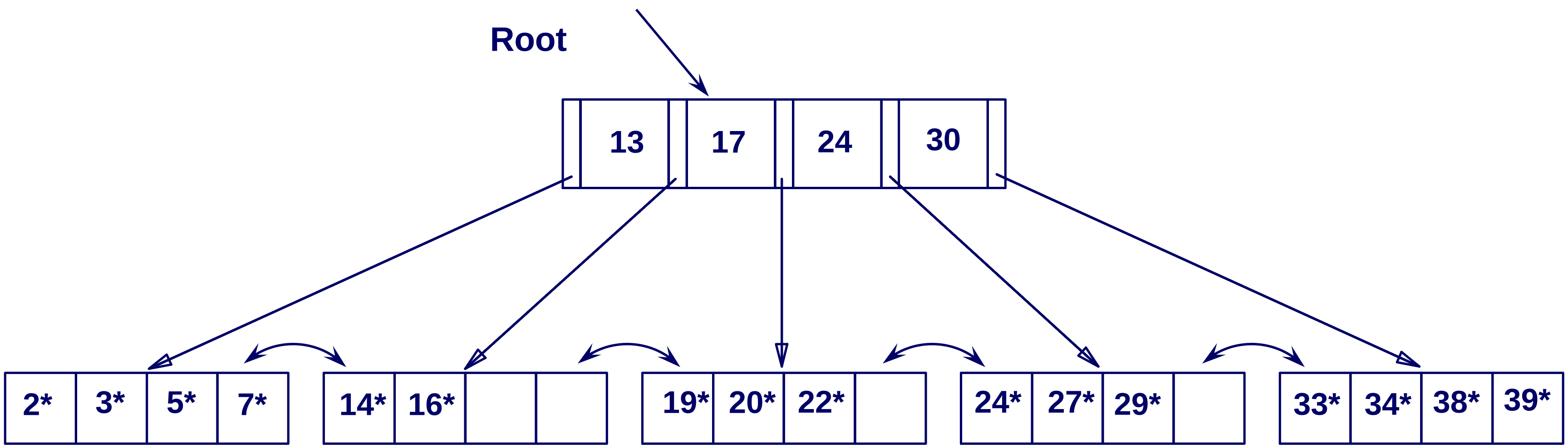
B+ Tree basics

- Insert/delete at $\log_F N$ cost
  - (F = fanout, N = # leaf pages)
  - Keep tree *height-balanced*
- Minimum 50% occupancy (except for root).
- Each node contains **d** <= *m* <= 2**d** entries.
  d is called the *order* of the tree.
- Supports *equality* and *range* searches efficiently.



B+ Tree search

- Given a search key, begin at root and branch based on key comparisons until you reach a leaf.
- Search for 5*, 15*, all data entries >= 24* ...

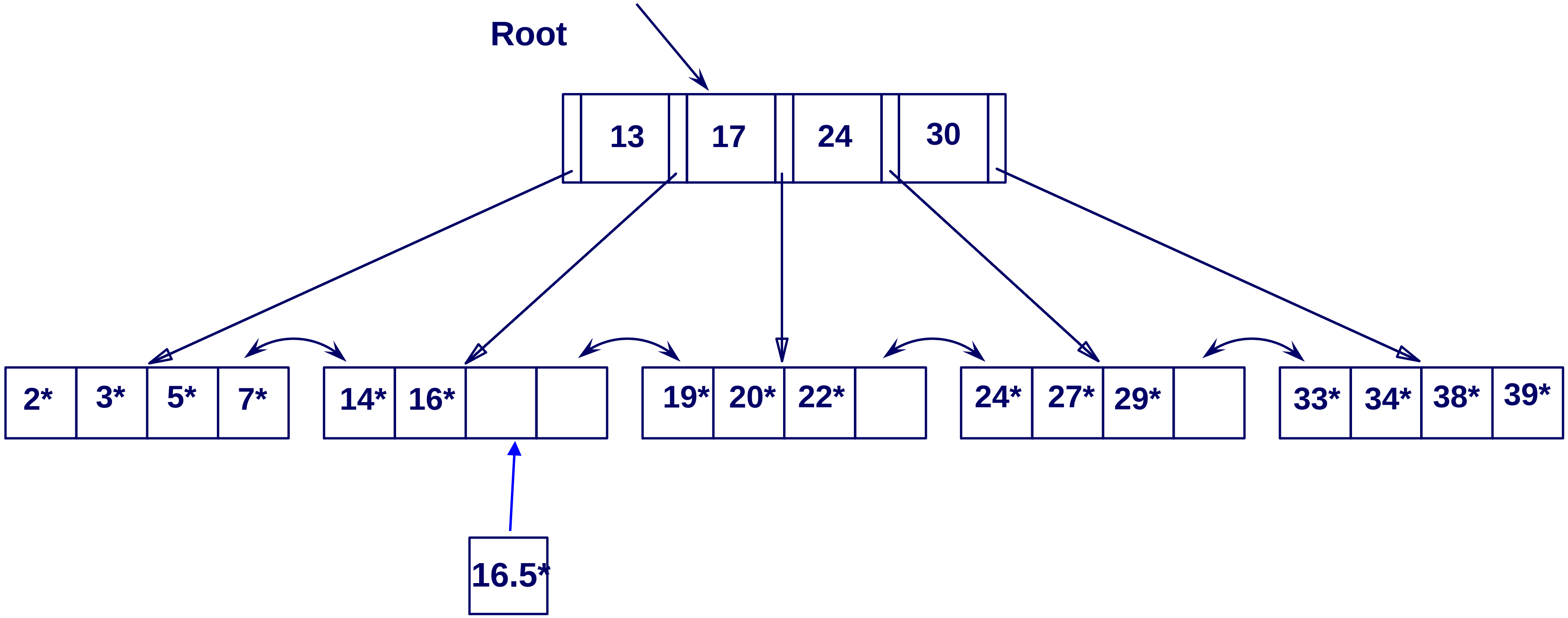


➢ *Based on the search for 15*, we <u>know</u> it is not in the tree!*

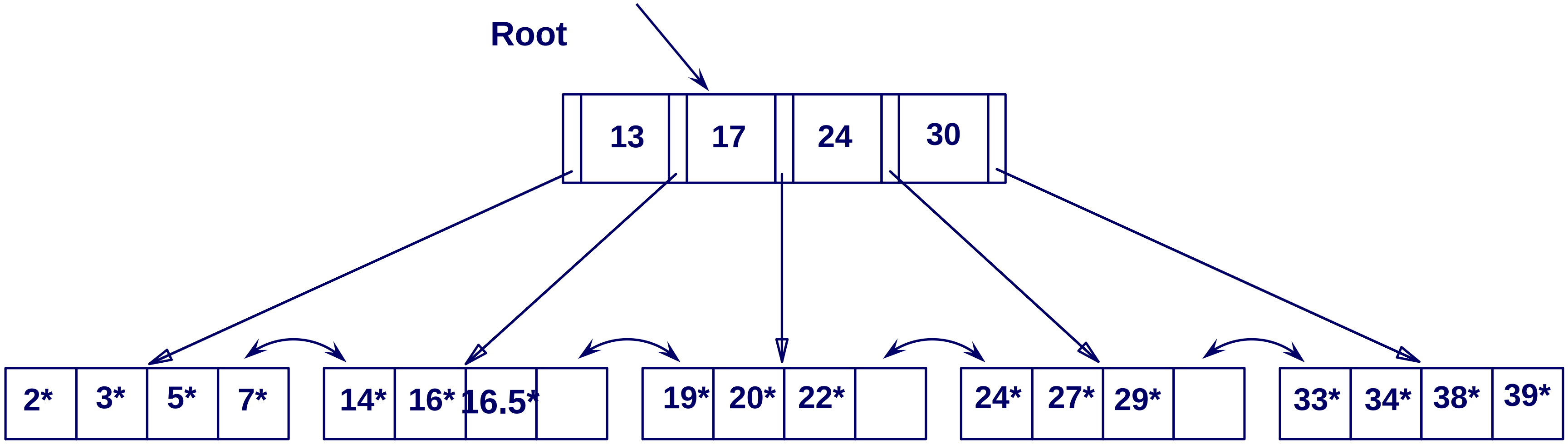
Inserting into a B+ Tree

- 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.*

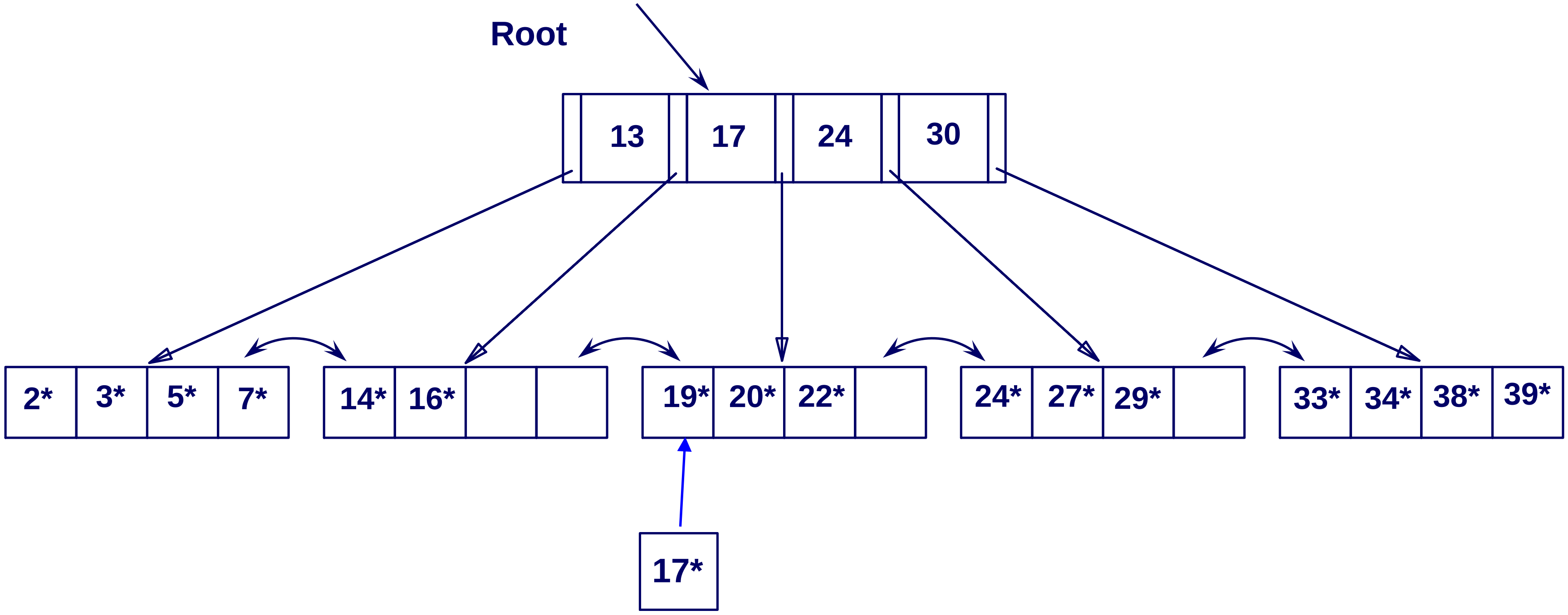
Inserting 16.5* Example



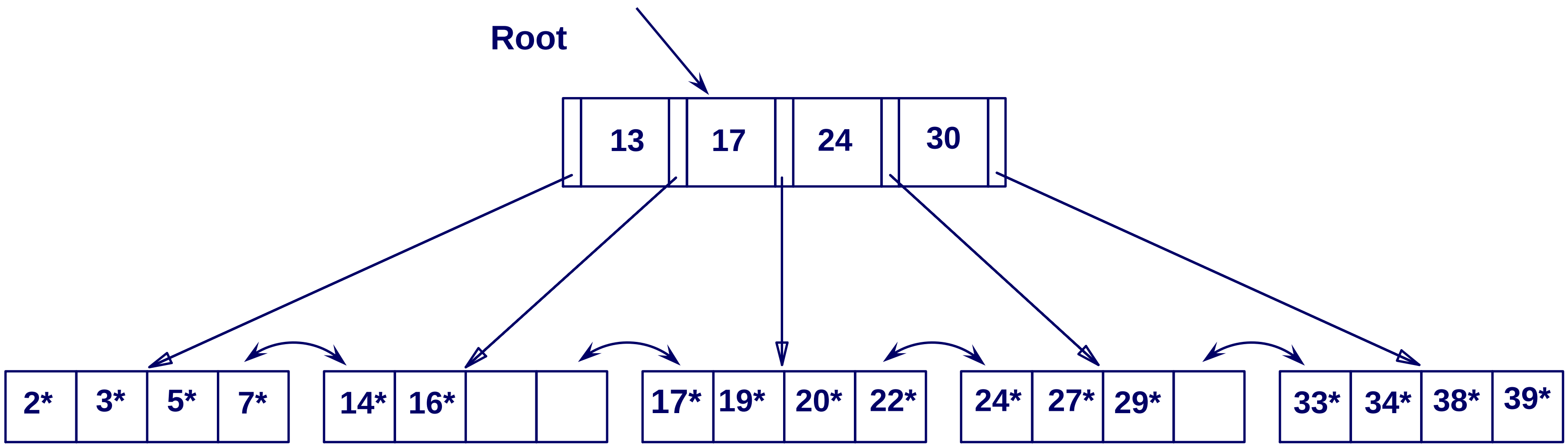
Inserting 16.5* Example



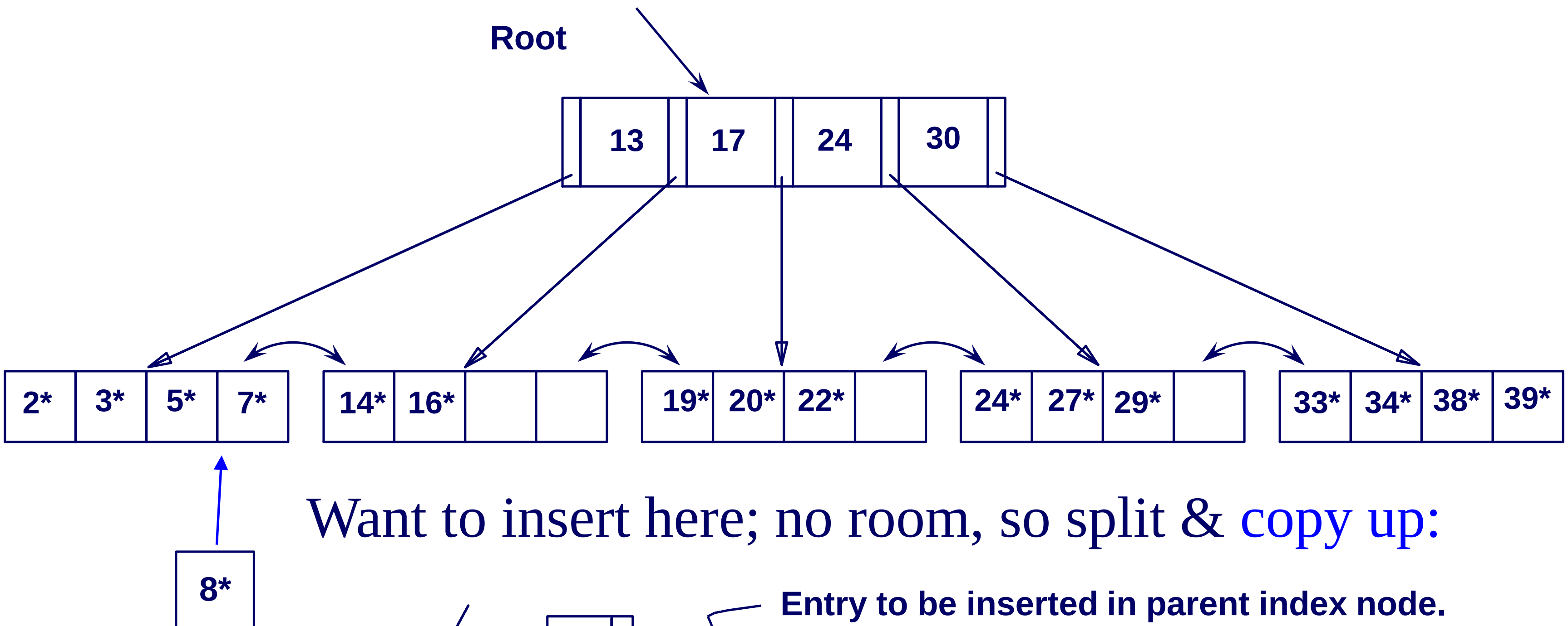
Inserting 17* Example



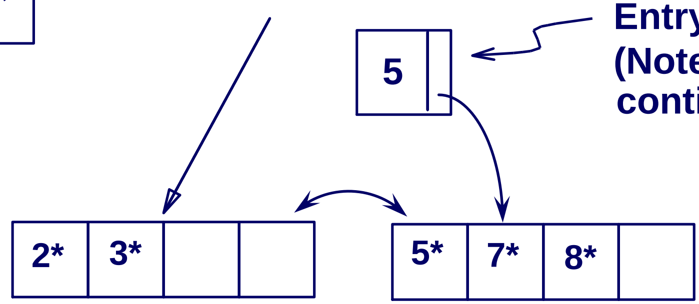
Inserting 17* Example



Inserting 8* Example: Copy up



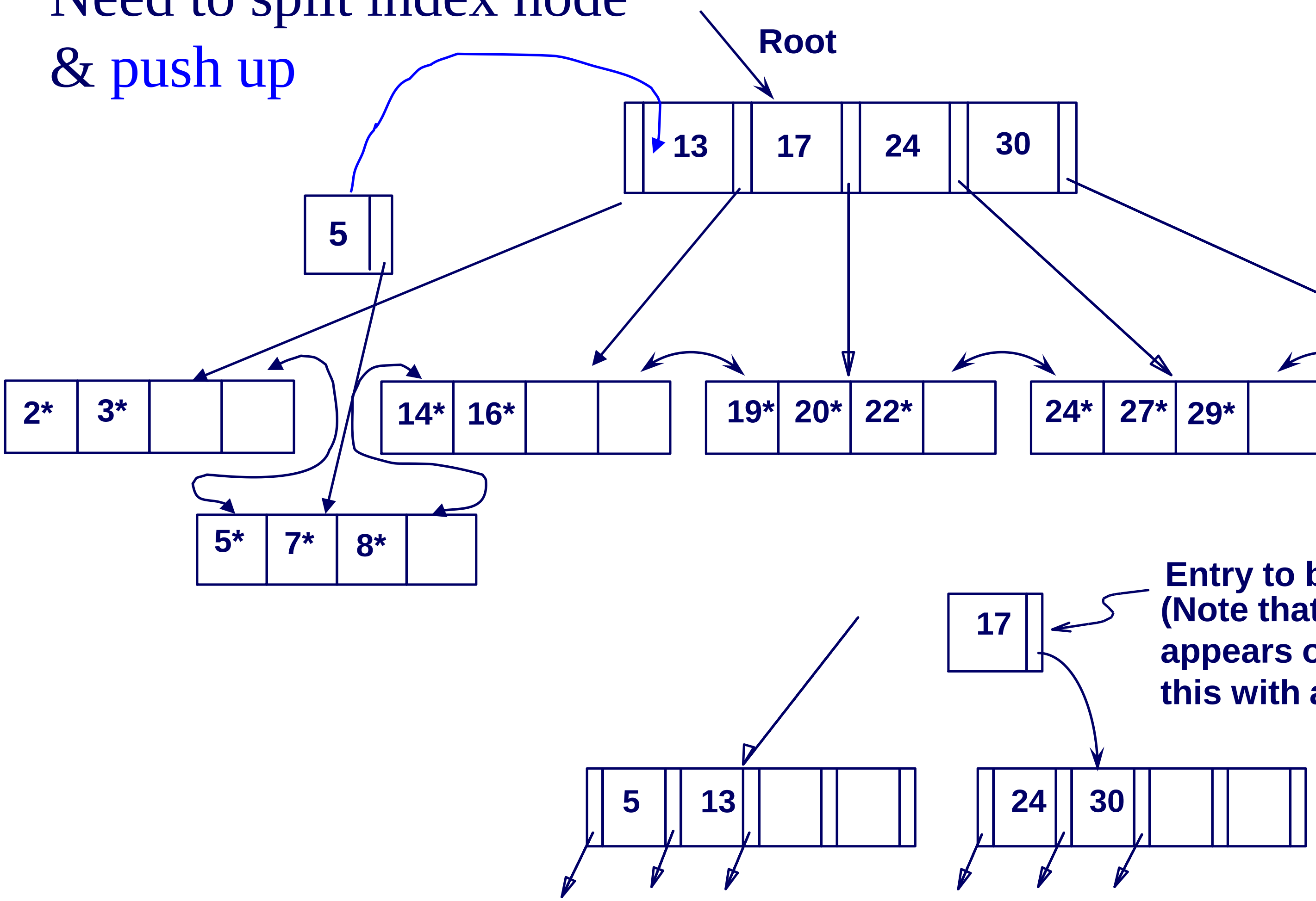
Want to insert here; no room, so split & copy up:



Entry to be inserted in parent index node.
(Note that 5 is copied up and
continues to appear in the leaf.)

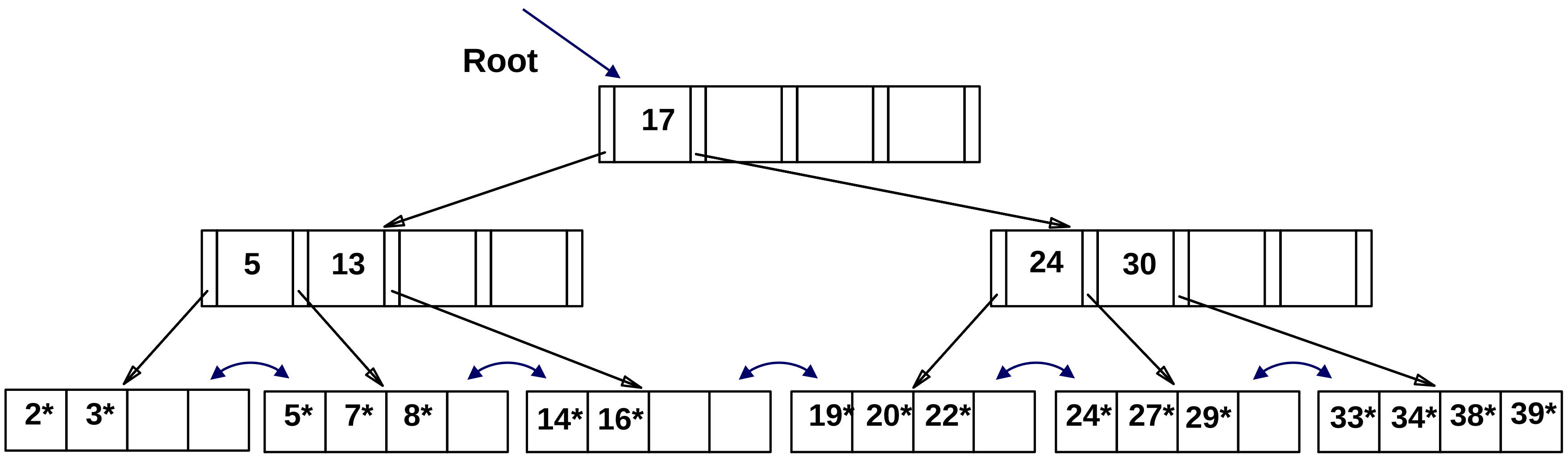
Inserting 8* Example, cont: Push up

Need to split index node
& push up



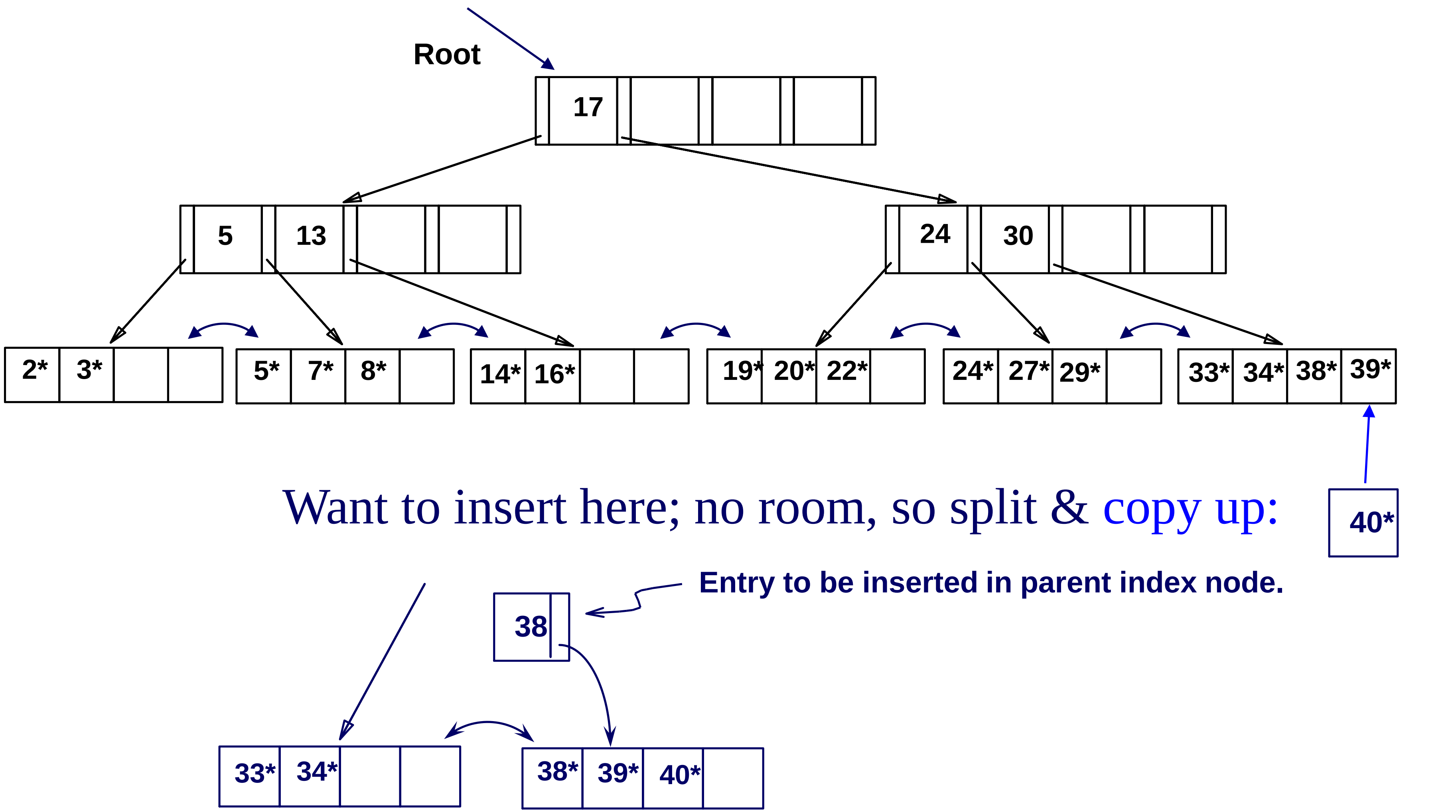
Entry to be inserted in parent node. (Note that 17 is pushed up and only appears once in the index. Contrast this with a leaf split.)

Example B+ Tree After Inserting 8*

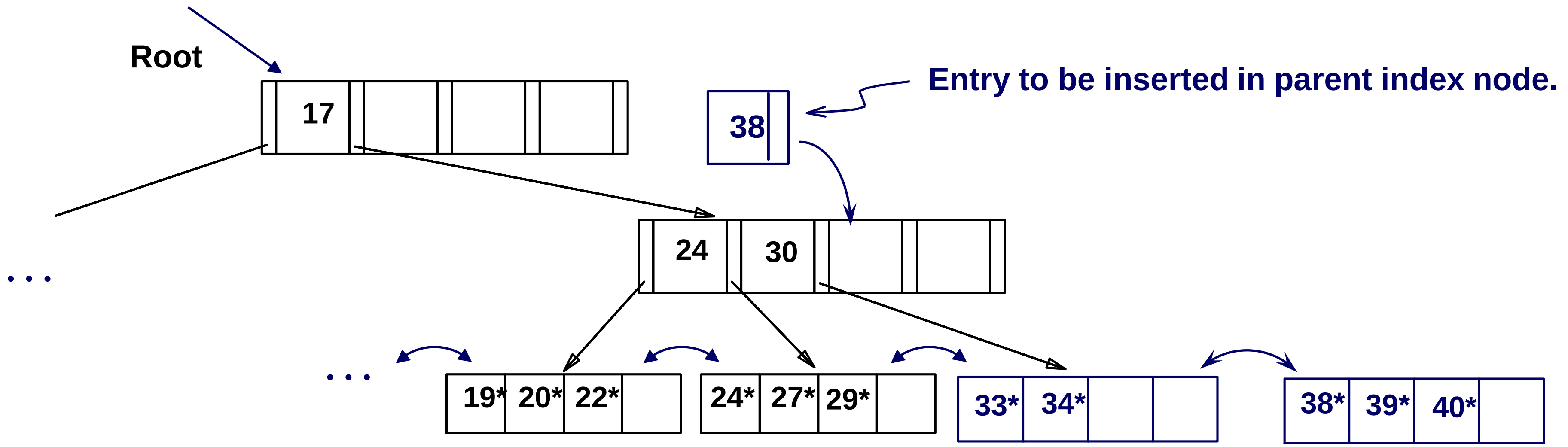


- Notice that root was split, leading to increase in height.
- In this example, we can avoid split by re-distributing entries; however, this is usually not done in practice.

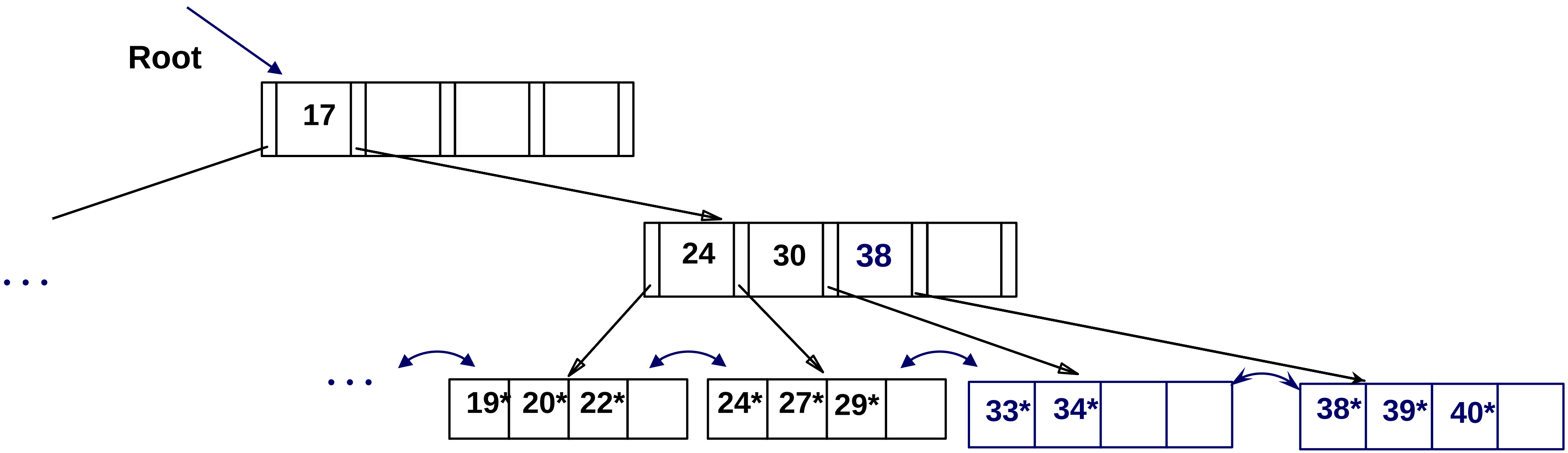
Inserting 40* Example: Copy Up



Inserting 40* Example



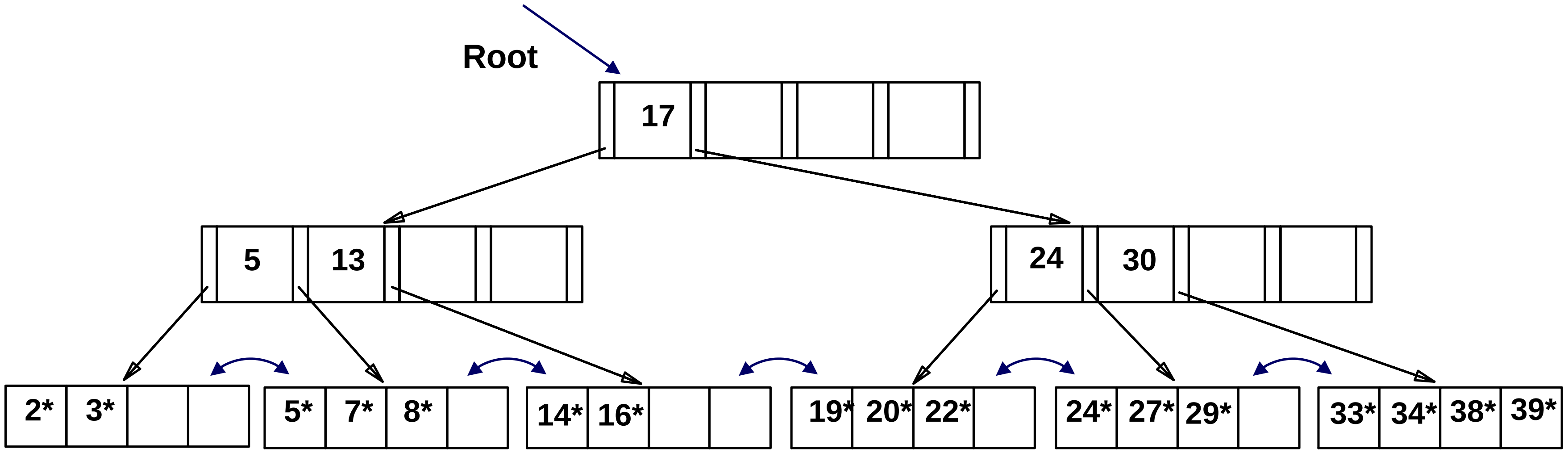
Inserting 40* Example



Deleting Data from a B+ Tree

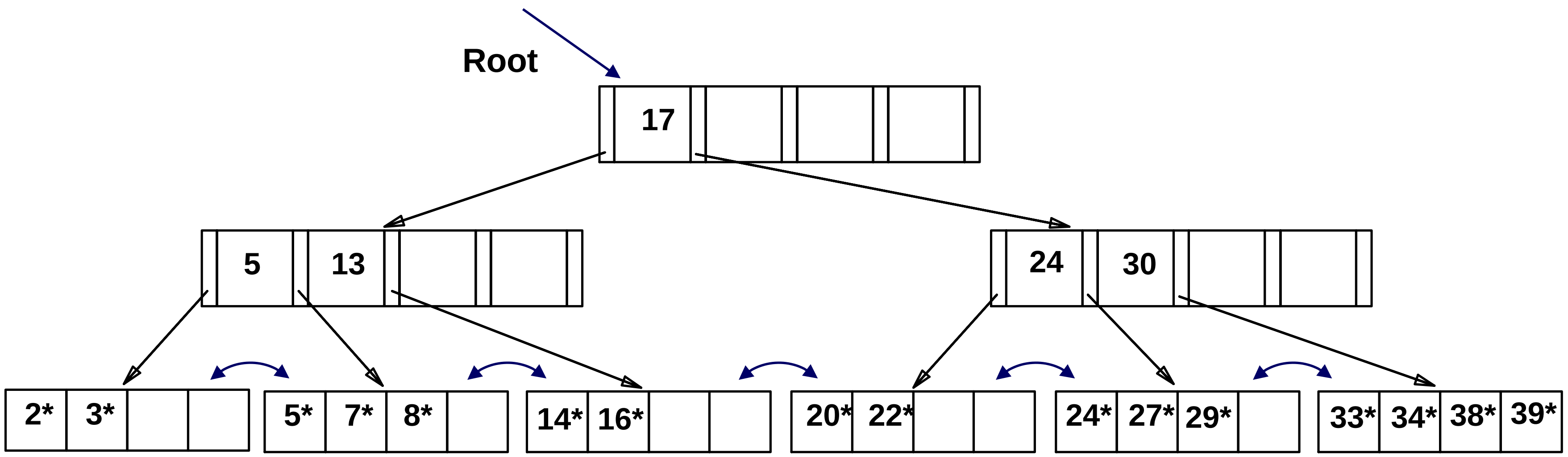
- Start at root, find leaf L where entry belongs.
- Remove the entry.
  - If L is at least half-full, done!
  - If L has only d-1 entries,
    - Try to re-distribute, borrowing from sibling (adjacent node with same parent as L).
    - If re-distribution fails, merge L and sibling.
- If merge occurred,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

Starting with this B+ Tree…



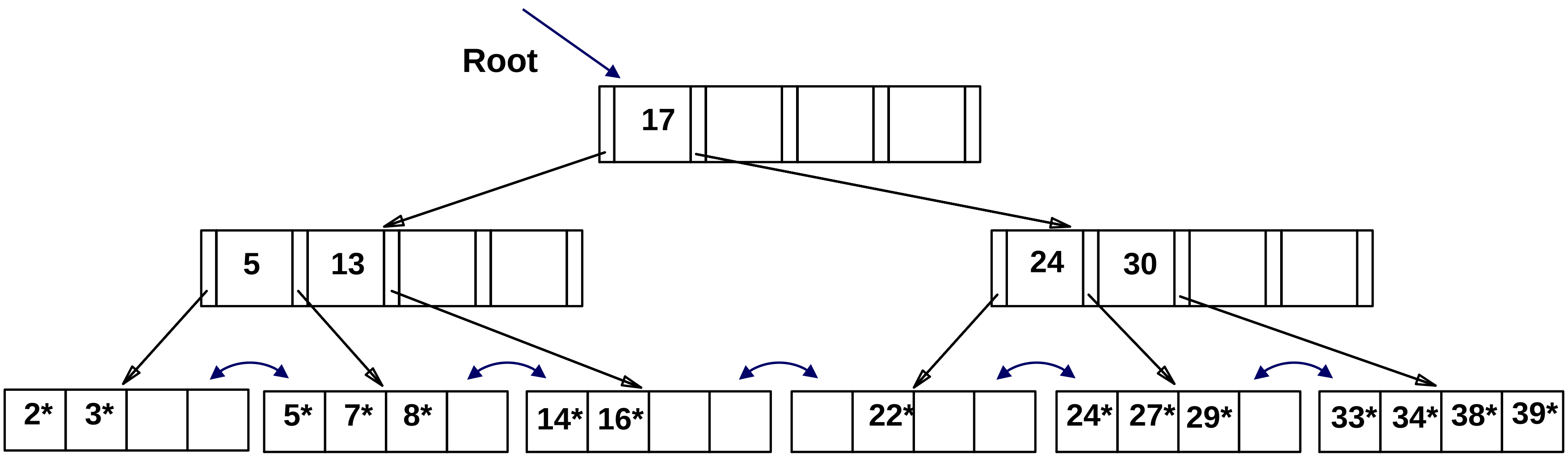
- Let’s delete 19.

Deleting 19*



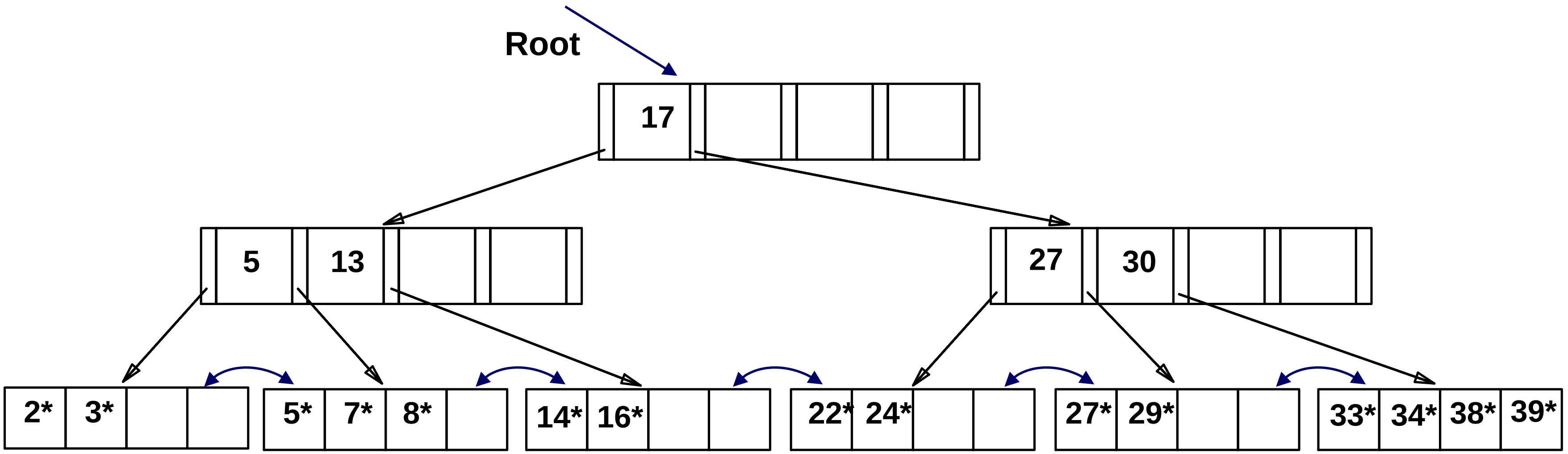
- We've deleted 19.
- Do we need to change the index?

Now let's delete 20*



- Problem: too few keys
- Use redistribution from a sibling

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

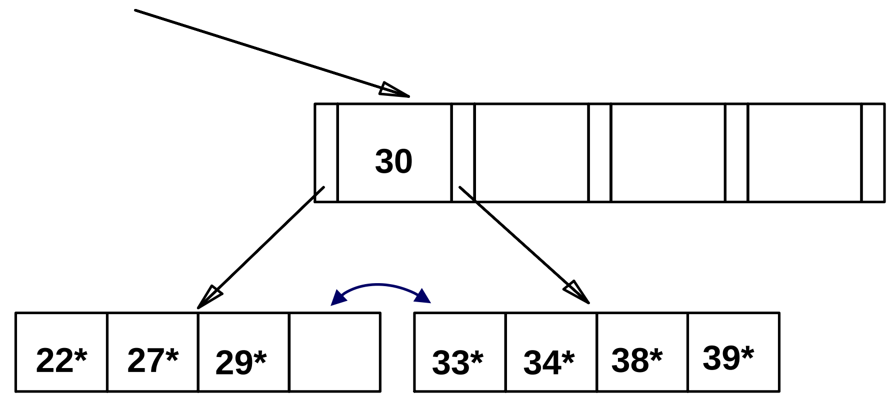


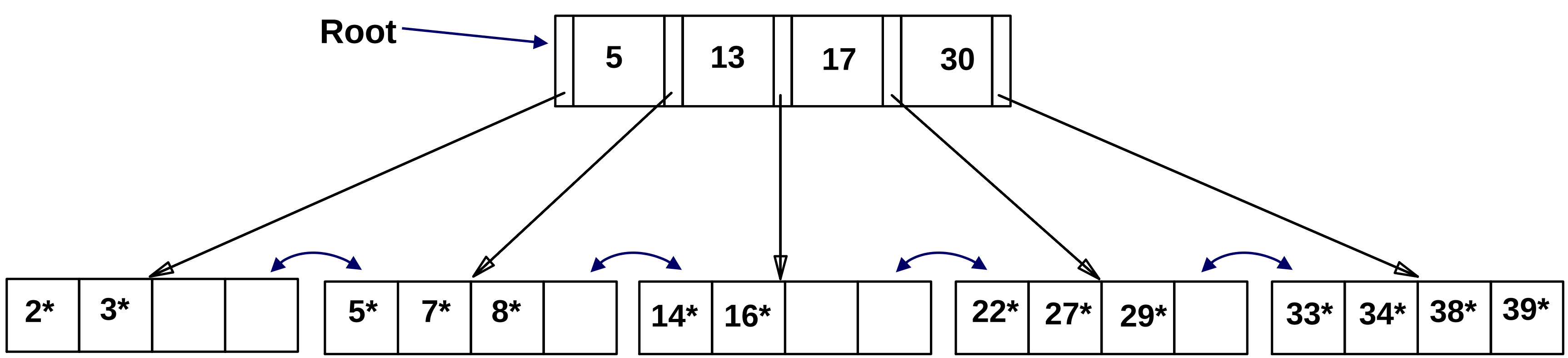
- Notice how middle key (27) is *copied up*.

  Now let's delete 24*

... And Then Deleting 24*

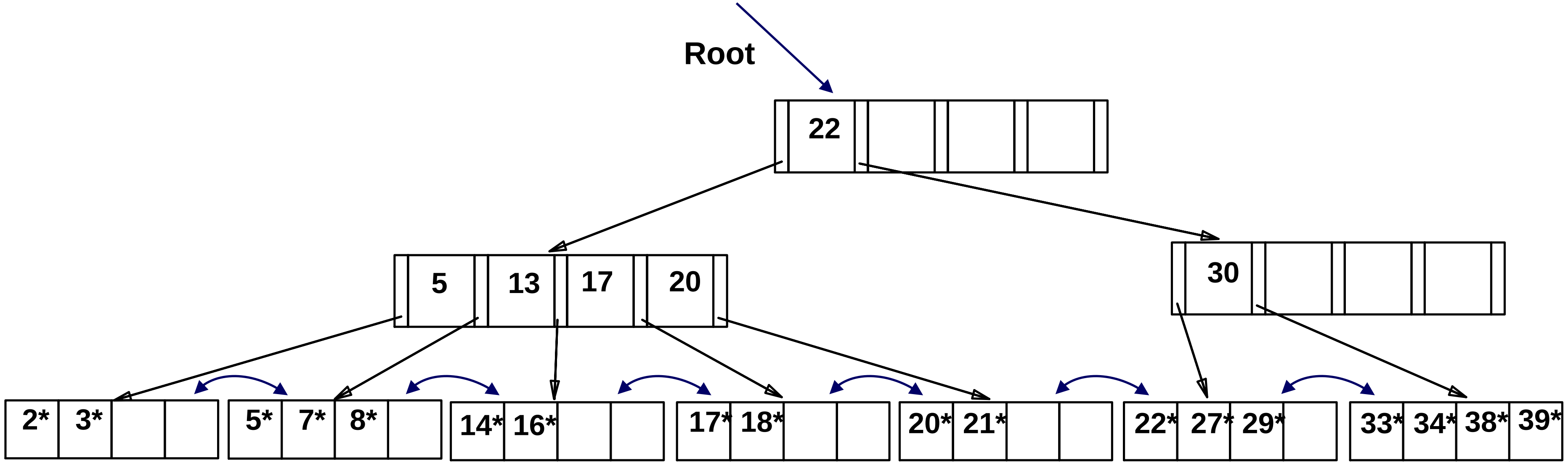
- Must merge sibling leaves.
- Index parent is too small.
- Observe `*toss*' of index entry (on right), and `*pull down*' of index entry from root (below).





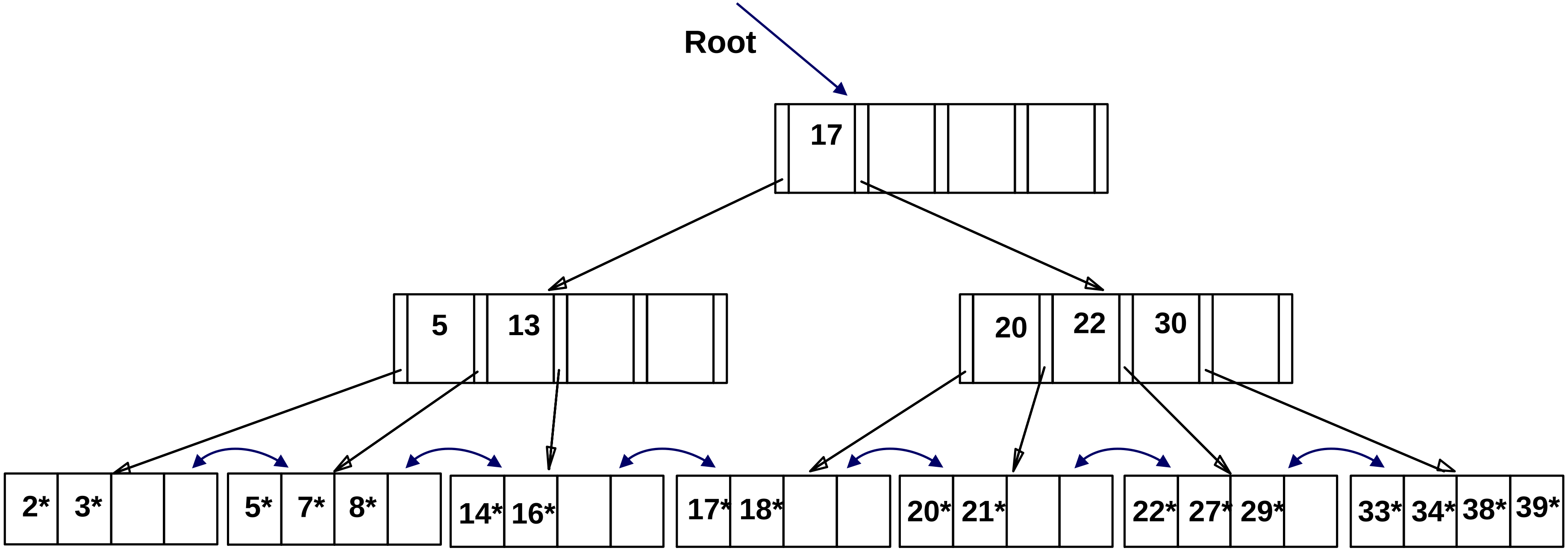
Example of Non-leaf Re-distribution

- 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.



B+ Tree Summary

B+ tree and other indices ideal for range searches, good for equality searches.

- 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.
- Note: Order (d) concept replaced by physical space criterion in practice ("at least half-full").
  - Records may be variable sized
  - Index pages typically hold more entries than leaves

Summary

- We've discussed the three approaches to file organization: heap, sorted, and hash.
- We've also discussed three core concepts to efficiently support access to data.
  - Indexing: in particular, B+-trees
  - Hashing
- Coming soon: how to use these ideas to implement relational operators.