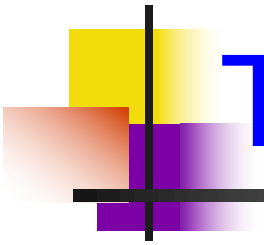


Tree-Structured Indexes

Chapter 10

Index Design Space

Organization Structure for k*

- **Hash-based**
  - + Equality search
- **Tree-based**
  - + Range, equality search
    - ➔B+Tree (dynamic)
    - ➔ISAM (static)

Data Entry (k*) Contents

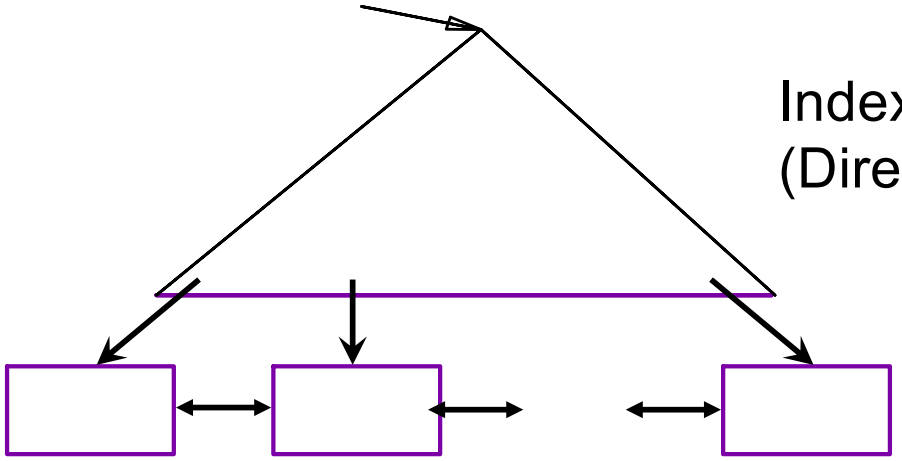
1. Actual Data record
   - index = file!
2. <k, rid>
   - actual records in a different file
3. <k, list of rids>

Motivation

- Range and equality searches very common
- Can scan heap file, but this is expensive
- **Goal:** *Create a dynamic index structure that allows for efficient evaluation and equality and range queries*

B+ Tree

- Height-balanced (dynamic) tree structure
- Minimum 50% occupancy (except for root).
- Each node contains **d** <= *m* <= 2**d** entries. The parameter **d** is called the order of the tree.
- Supports equality and range-searches efficiently.



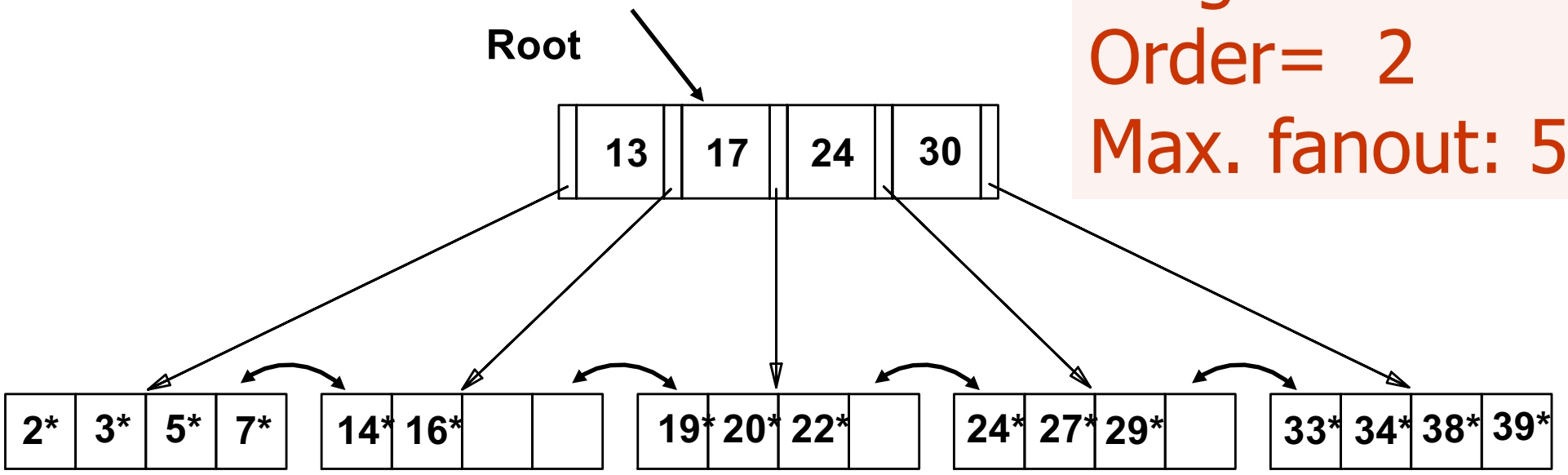
Index Entries
(Direct search)

Data
Entries

Index Entries
Entries in the index
(i.e. non-leaf) pages:
(search key value, pageid)

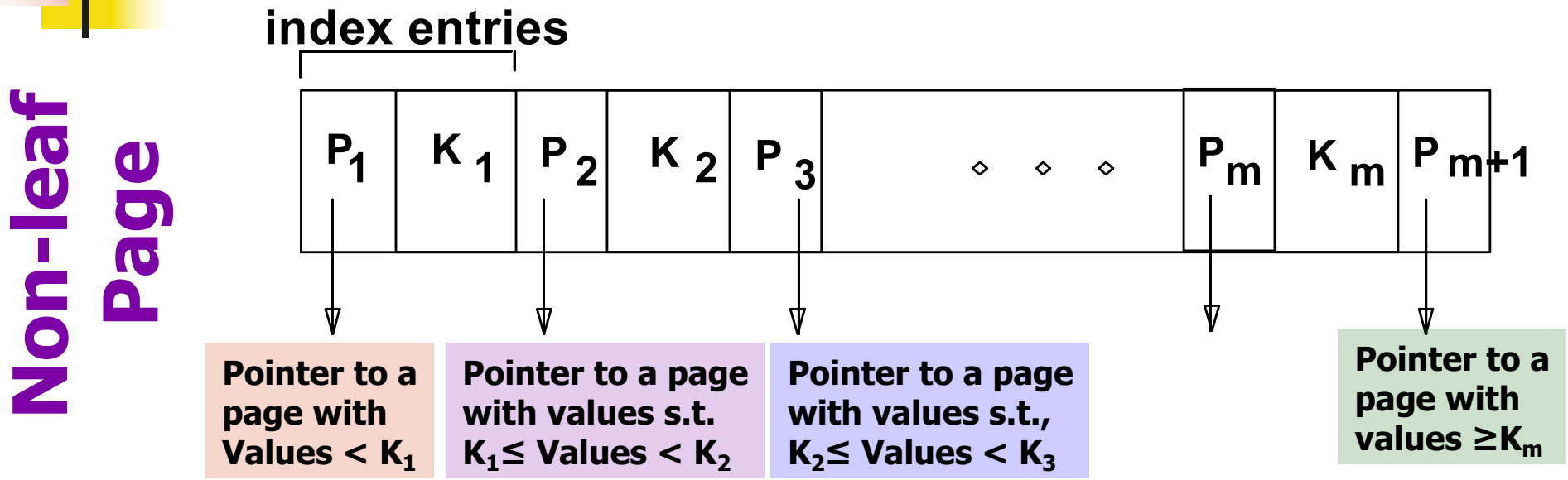
Example B+ Tree

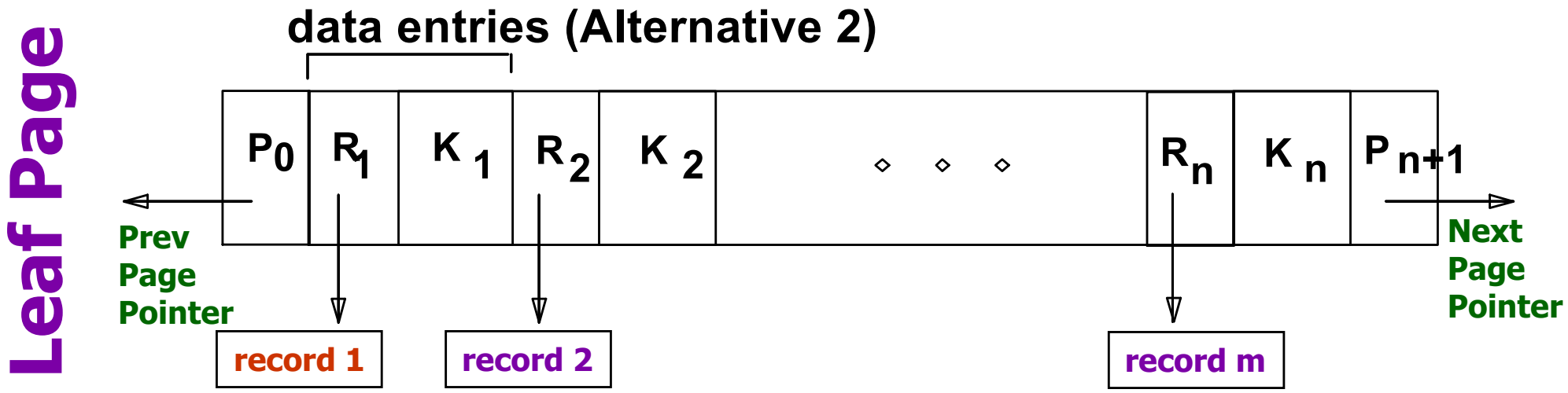
- Search: Starting from root, examine index entries in non-leaf nodes, and traverse down the tree until a leaf node is reached
  - Non-leaf nodes can be searched using a binary or a linear search.
- Search for 5*, 15*, all data entries >=24*

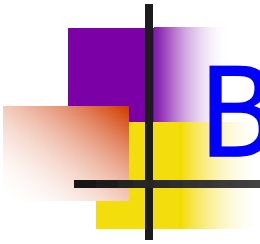


Height = 1
Order= 2
Max. fanout: 5

B+-tree Page Format







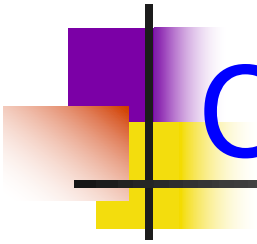
B+ Trees

- What is the height of a B+ tree?
  - Fanout F (average number of children for non-leaf node)
  - N total leaf pages

$\log_F N$

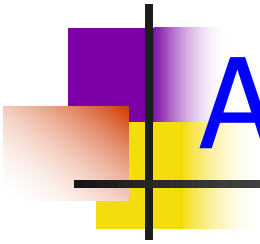
B+ Trees in Practice

- Typical order 100. Typical fill-factor 67%.
  - Maximum fanout: 201
  - Average fanout = 133
- Typical capacity:
  - Height = 1: 133 pages of data entries (leaf pages)
  - Height = 2: 133^2 pages of data entries
  - Height = 3: 133^3 (> 2 million) pages of data entries
  - Height = 4: 133^4 (> 300 million) pages of data entries
- Can often keep top levels of index in buffer pool
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes



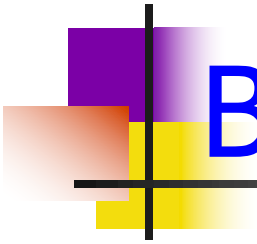
Check your understanding

- You are given a file of 10 million records
- Suppose you can store 10 data entries per leaf page
- You build a B+ Tree with order 75, 67% average fill-factor
- What is the avg fanout?
  - Fanout = 100
- What is the height of your B+ Tree?



A Note on Order

- Some literature uses
  - *order* to be the *maximum* number of entries
- In this class, order means *minimum* number of entries (book uses this)
- *Order* (**d**) concept replaced by physical space criterion in practice (e.g., *at least half-full*).
  - Index (i.e. non-leaf) pages can typically hold many more entries than leaf pages.



B+ Tree Operations

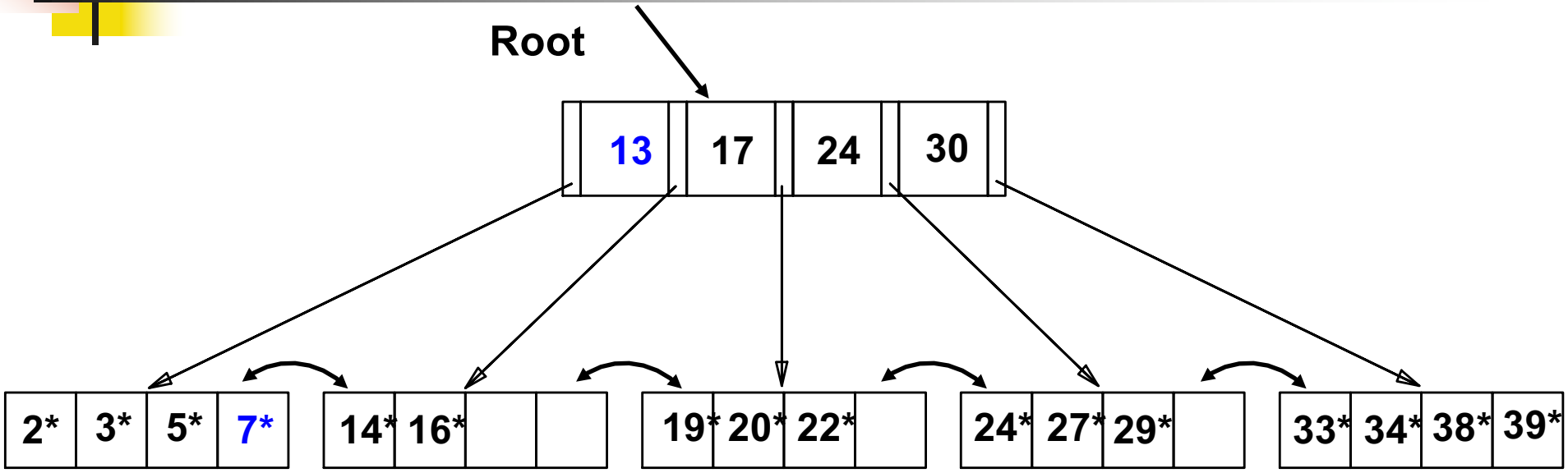
- Search
  - Equality
  - Range
- **Insert data entry**
- Delete data entry
- Bulk load

B+-Tree: Inserting a Data Entry

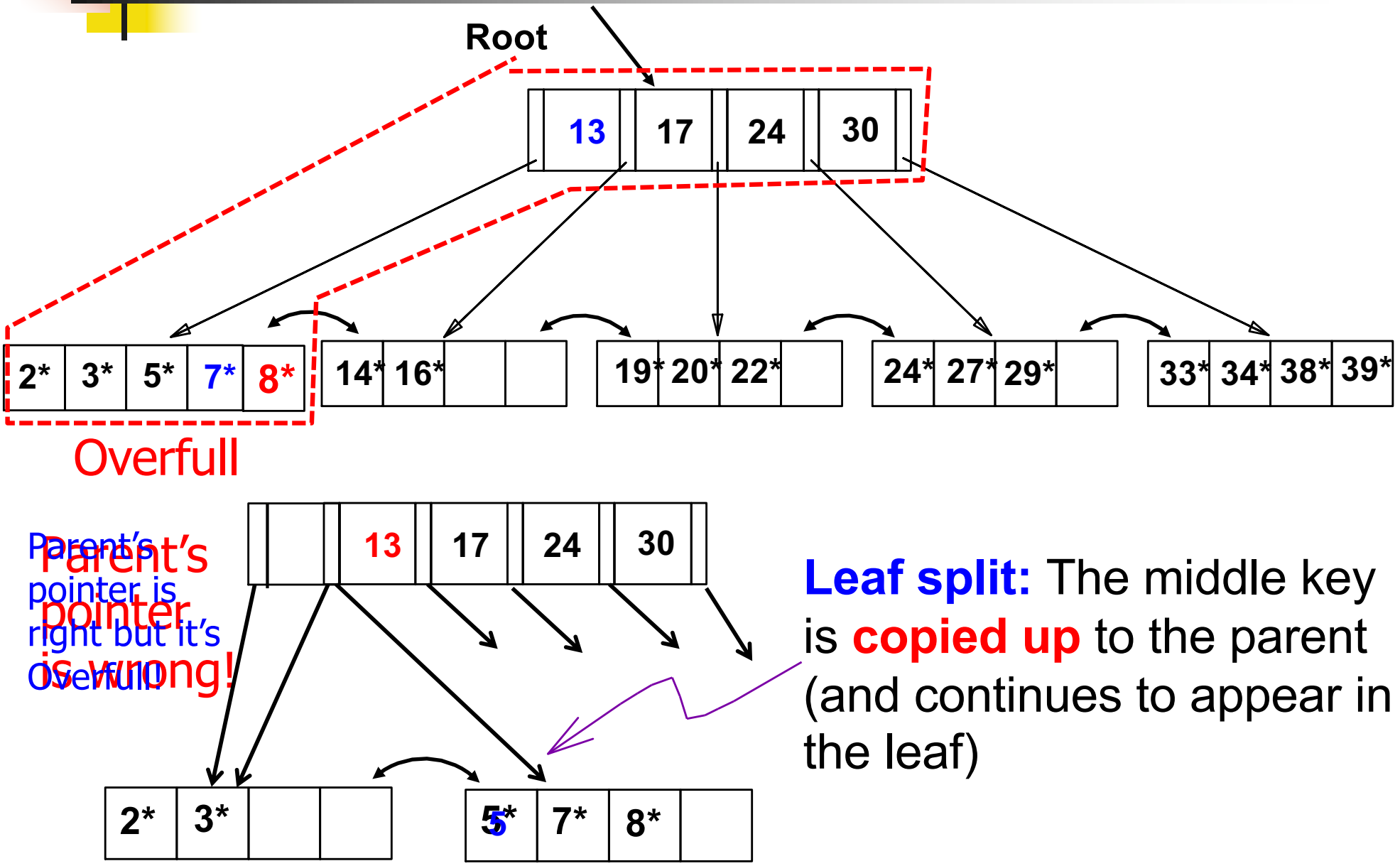
- Maintain invariants:
  - Search-tree property.
  - All nodes must be at least ½ full (except root node), i.e., has between d and 2d entries.
  - Root node is allowed to have a single entry

- Strategy:
  - Split nodes when they become full and a node is added:
    - An overfull node with (2d + 1) entries split into two nodes consisting of d and (d+1) entries, restoring the invariant

Inserting 8* into B+ Tree



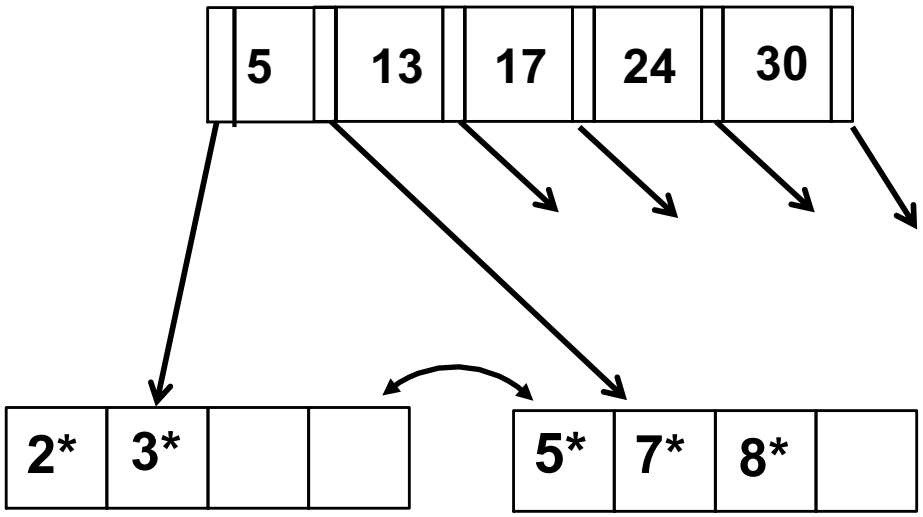
Inserting 8* into B+ Tree



Leaf split: The middle key is **copied up** to the parent (and continues to appear in the leaf)

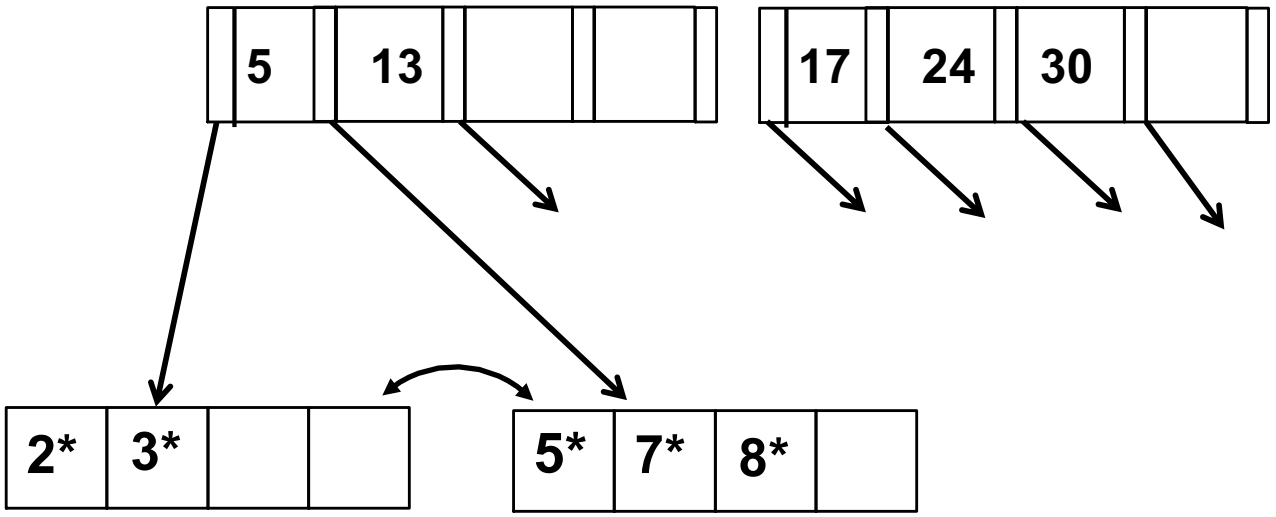
Splitting overfull non-leaf nodes

Overfull



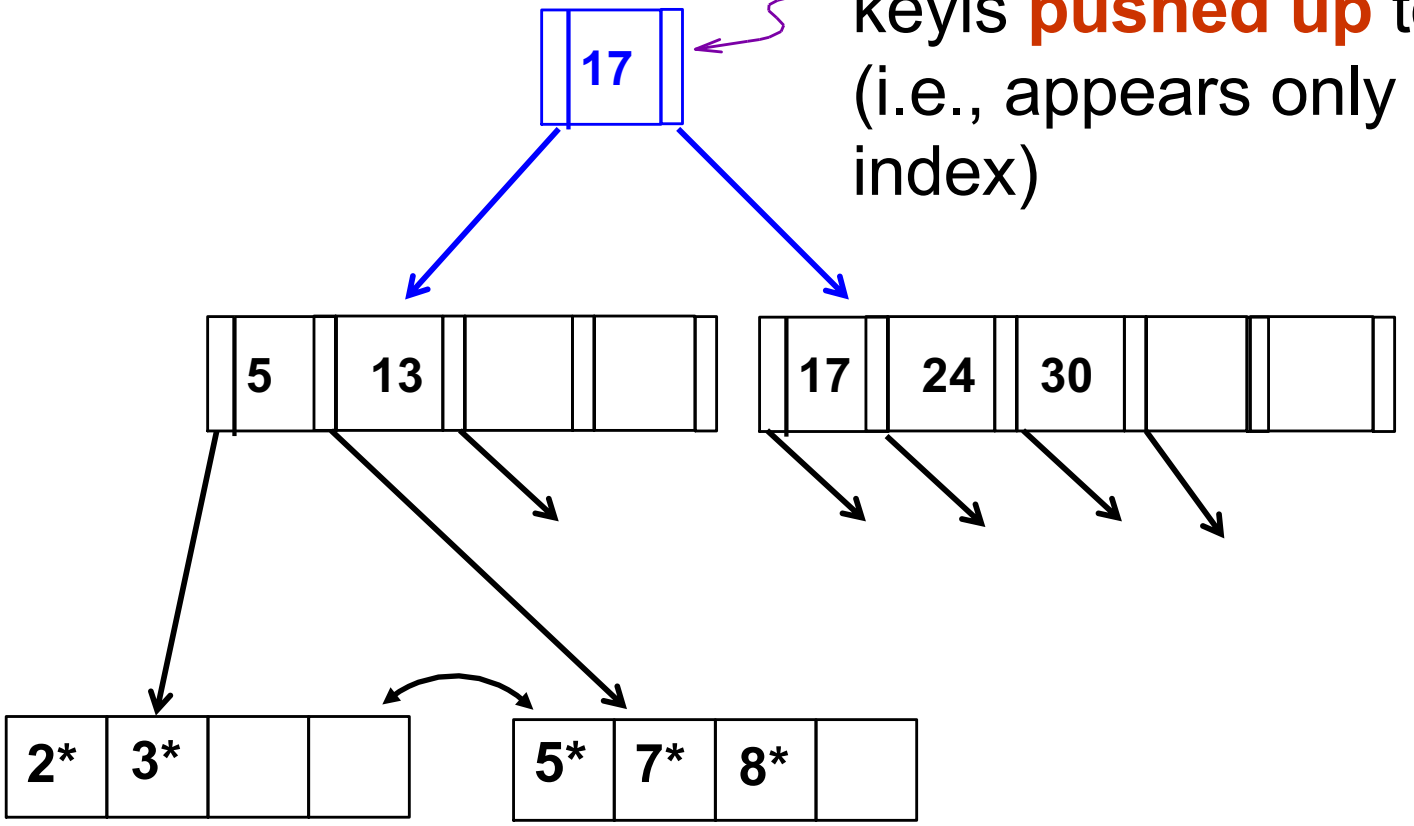
Splitting overfull non-leaf nodes

Split



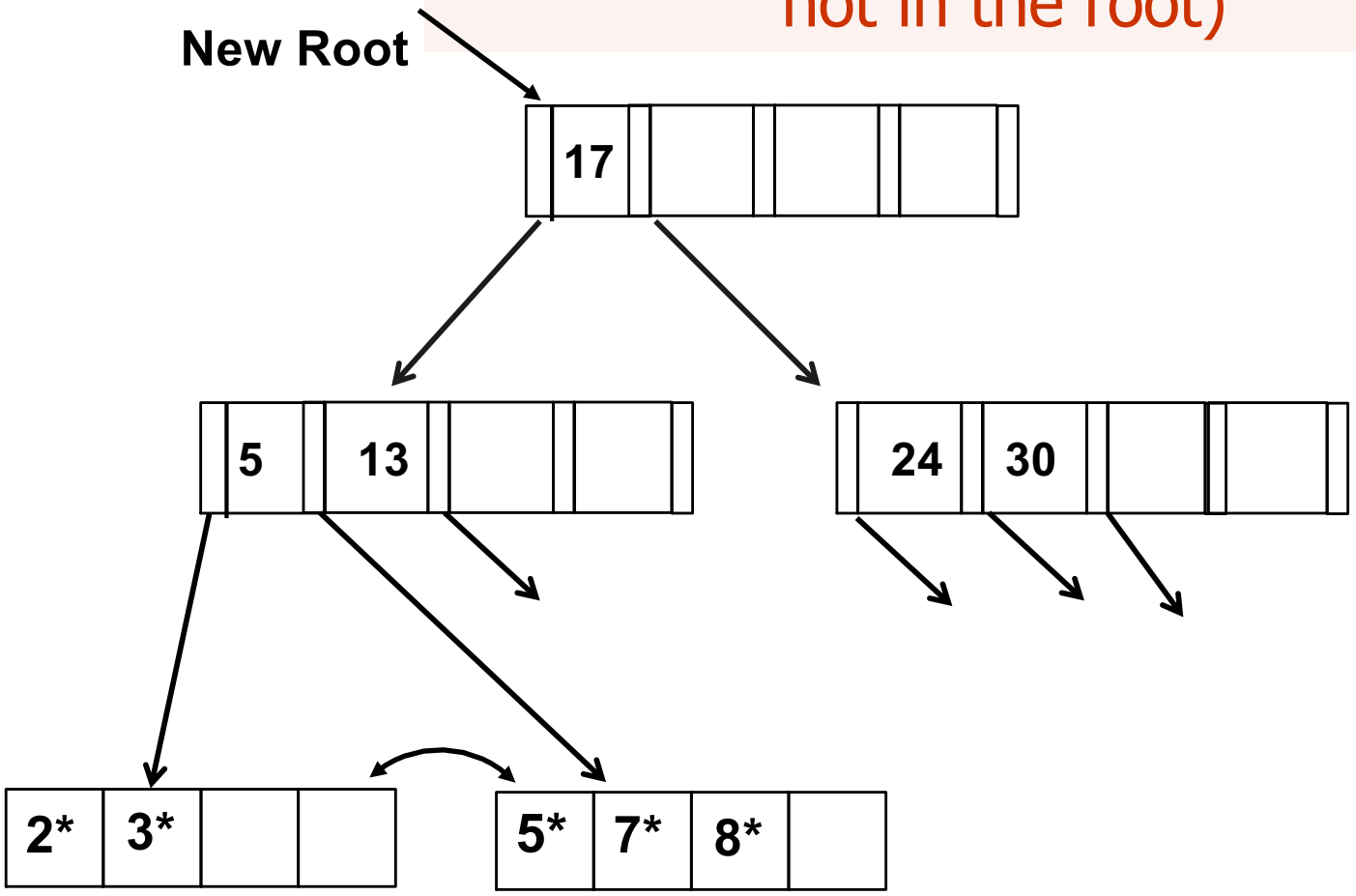
Splitting overfull non-leaf nodes

Non-leaf split: The middle keyis **pushed up** to the parent (i.e., appears only once in the index)

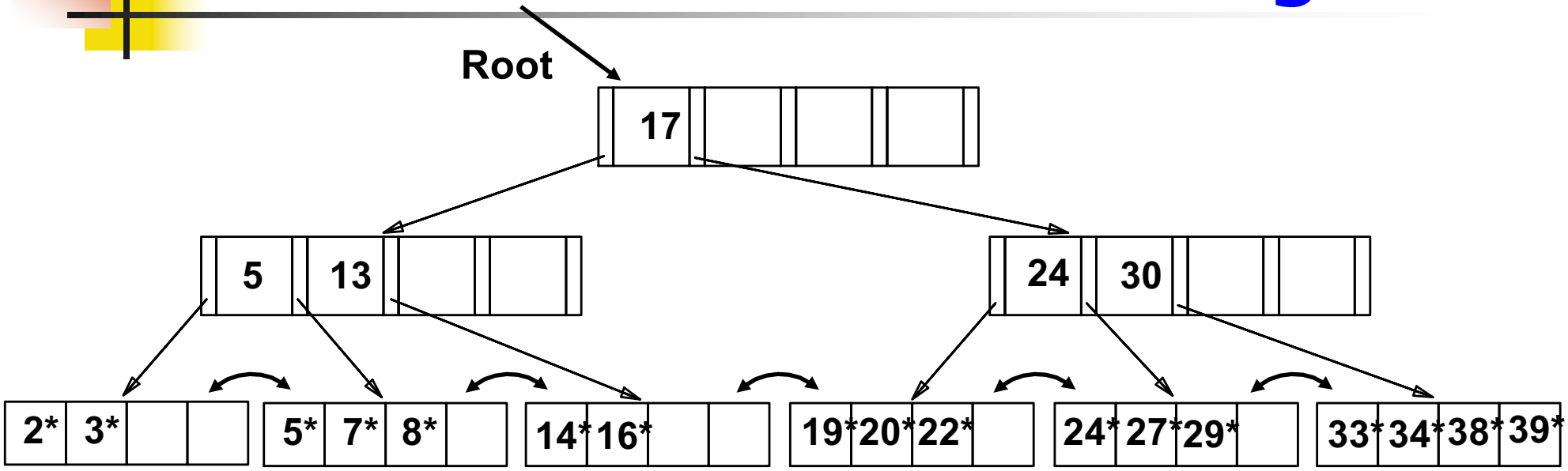


Splitting overfull non-leaf nodes

Minimum occupancy is guaranteed in both leaf and index page splits (but not in the root)

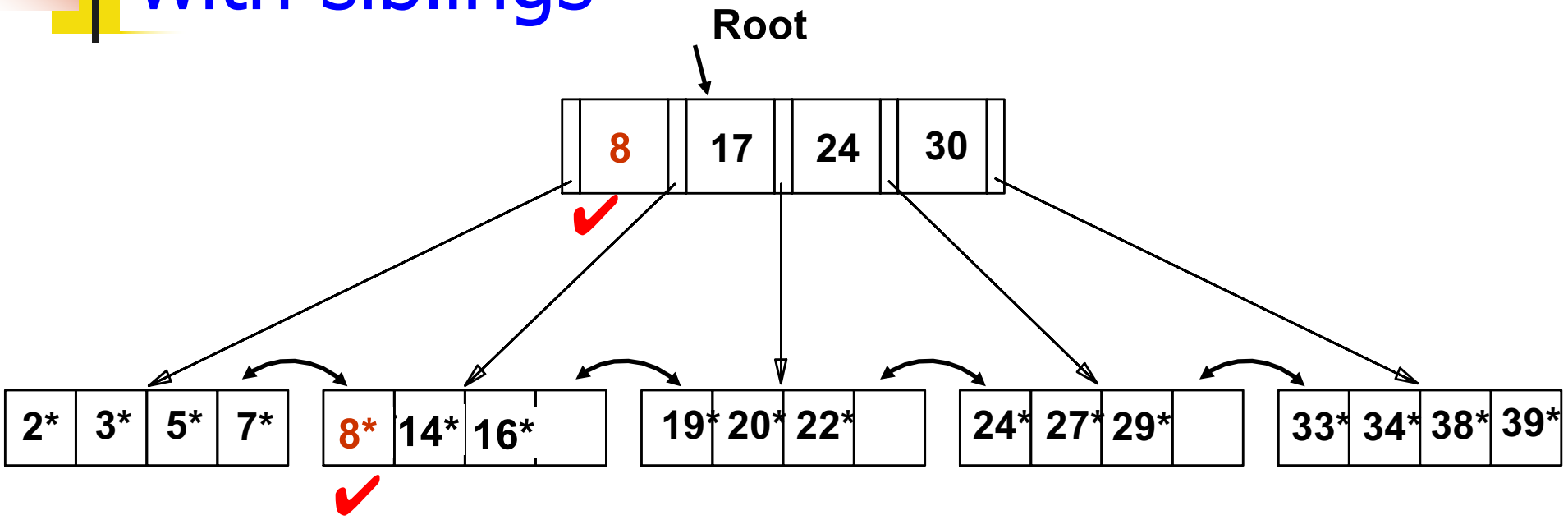


The B+-tree after inserting 8*

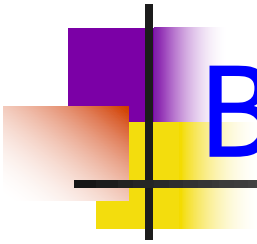


- Root was split: height increases by 1
- Could avoid split by re-distributing entries with a sibling
  - Sibling: immediately to left or right, **and same parent**

Inserting 8* via entry re-distribution with siblings



- Re-distributing entries with a sibling
  - Improves page occupancy, possibly reduces height
  - Usually not used for non-leaf node splits. Why?
    - Increases I/O, especially if we check both siblings
    - Better if split propagates up the tree (rare)
  - Use only for leaf level entries
    - have to set pointers



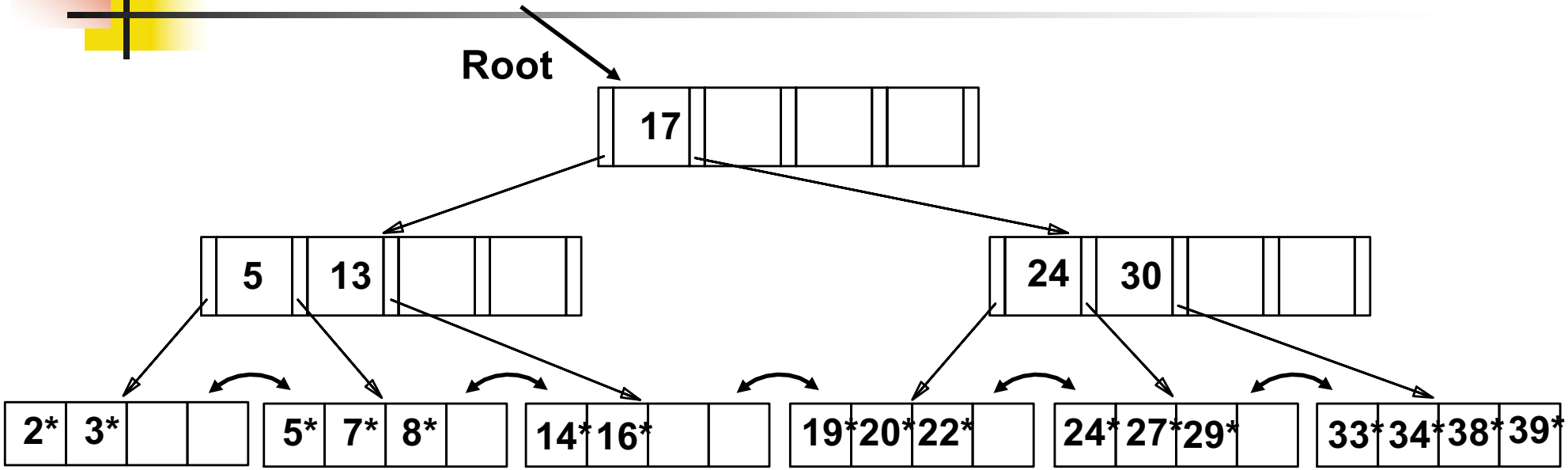
B+ Tree Operations

- Search
  - Equality
  - Range
- Insert data entry
- **Delete data entry**
- Bulk load

B+-Tree: Deleting a Data Entry

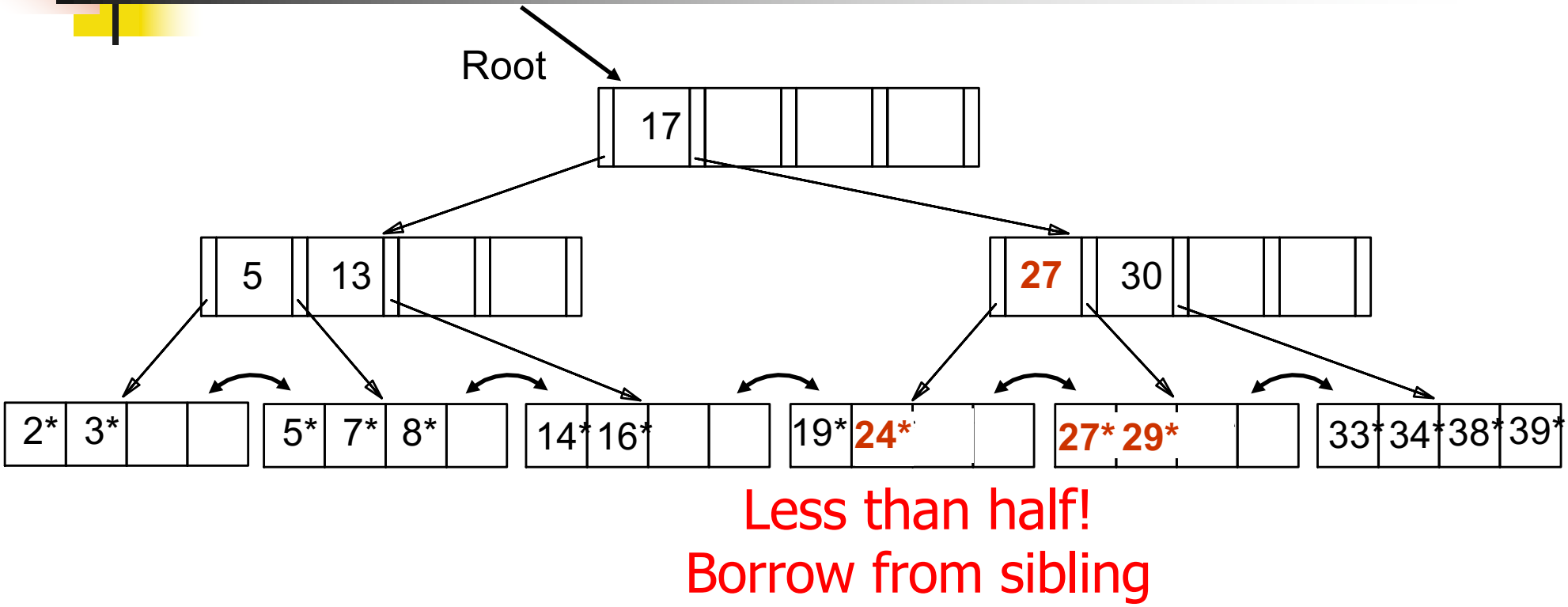
- Find the data entry (will always be at a leaf)
- Delete it
- Restore the B+-tree invariant
  - If L is at least half-full, *done!*
  - If L has only **d-1** entries,
    - Try to re-distribute, borrowing from a *sibling (adjacent node with same parent as L)*.
    - If re-distribution fails, *merge* *L* and sibling.
- On merge, delete relevant entry in parent
- Merge could propagate to root, decreasing height.

Example tree



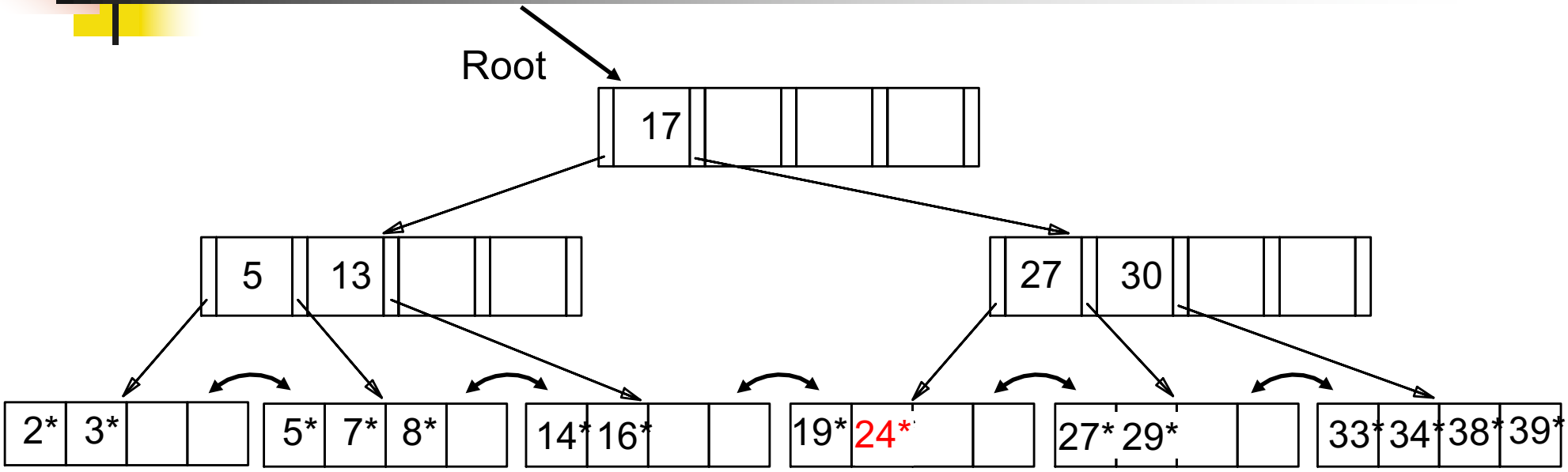
- Task: Delete 22, 20, 24
- Deleting 22 is easy. Invariant maintained
- Deleting 20 is harder. Node would become less than half full

Deleting 22* and 20*



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

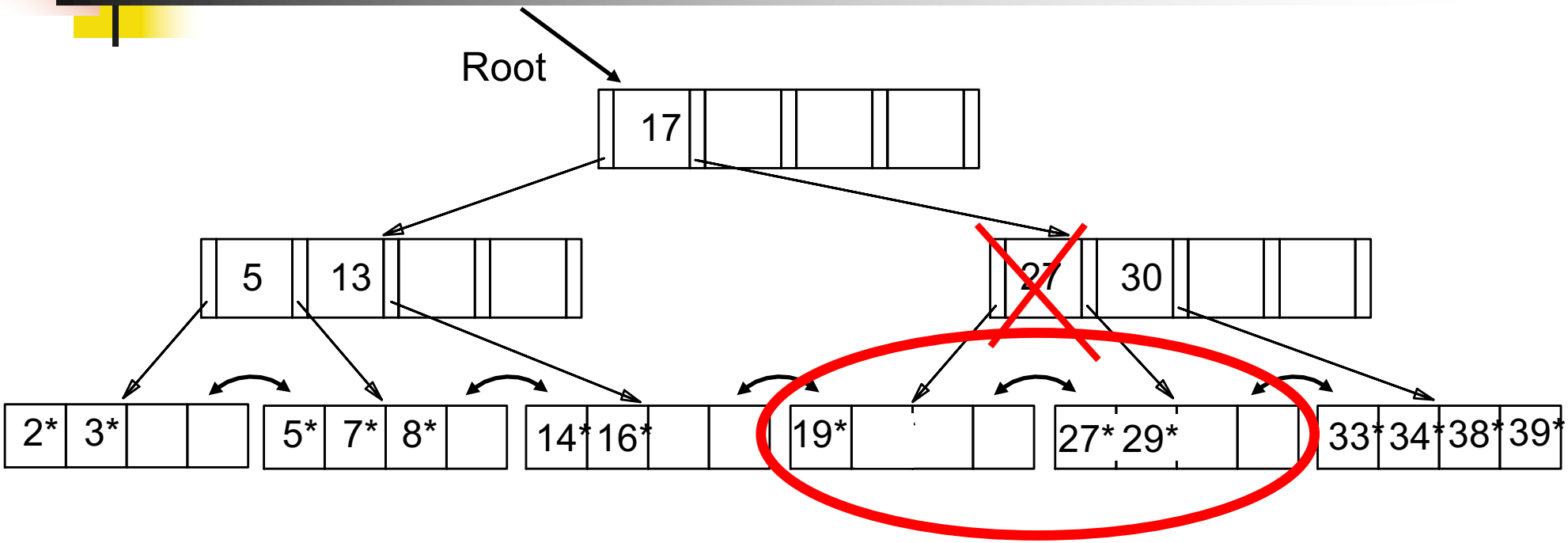
… And Then Deleting 24*



Less than half!
And cannot borrow
from its sibling!

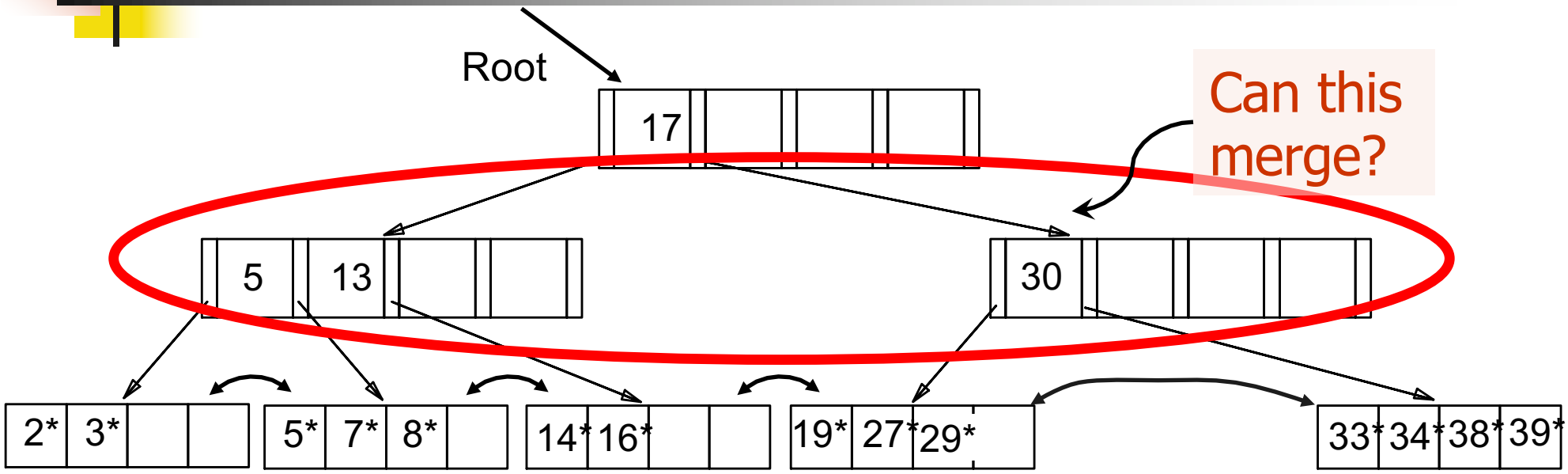
- Must merge.
- In the non-leaf node, ***toss* the** index entry with key value = 27

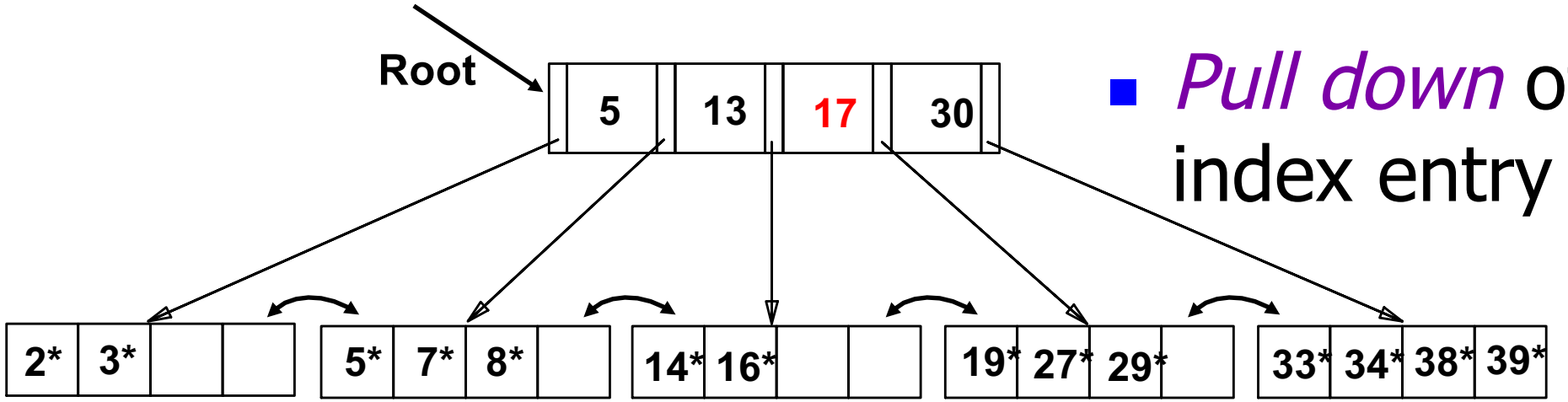
... And Then Deleting 24*



- Must merge.
- In the non-leaf node, ***toss* the** index entry with key value = 27

... And Then Deleting 24*

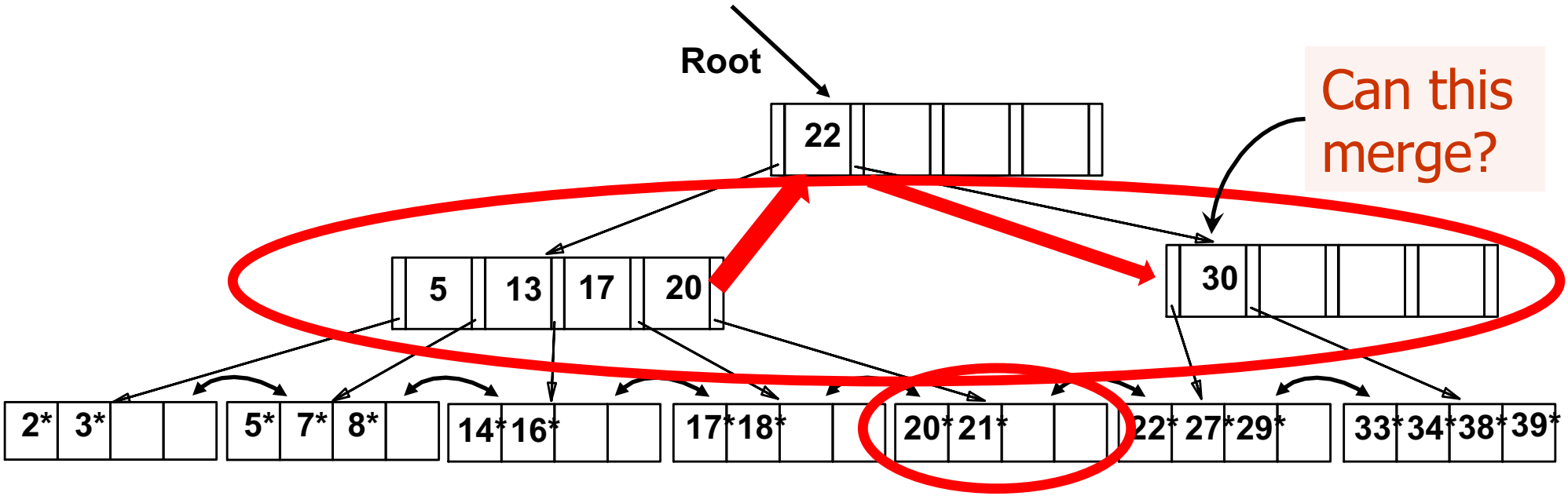




- *Pull down* of index entry

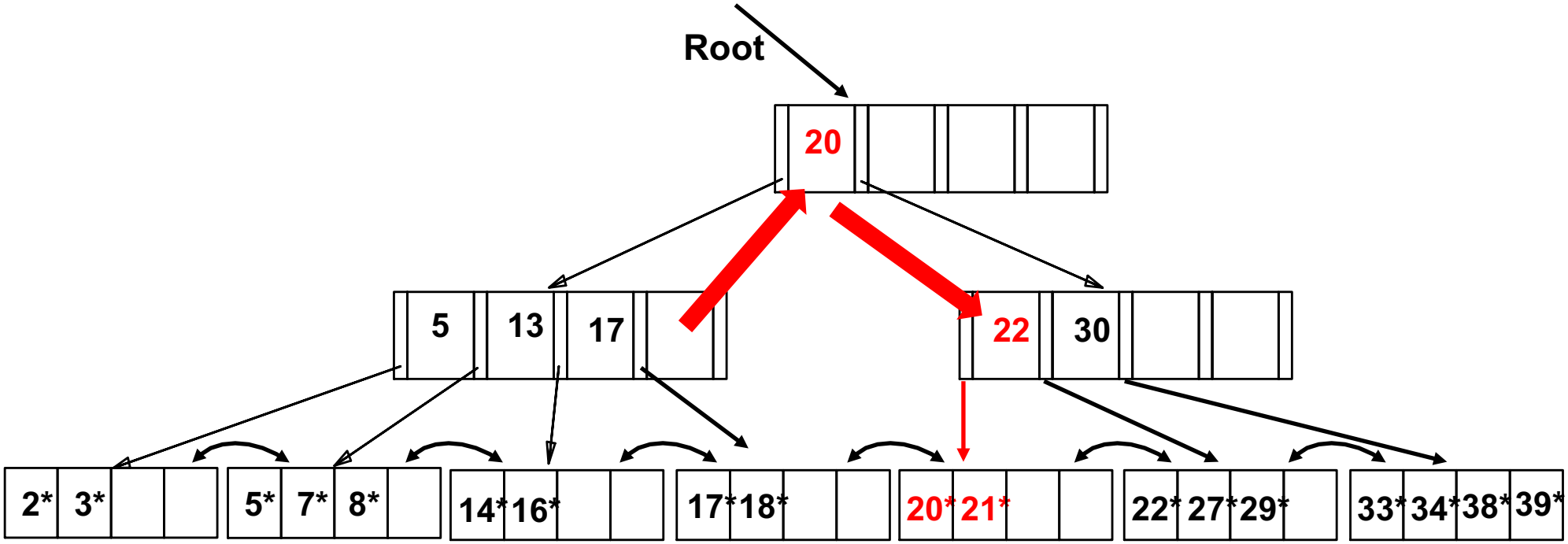
Another Deletion Example: Non-leaf Re-distribution

- Can re-distribute entry from left child of root to right child.



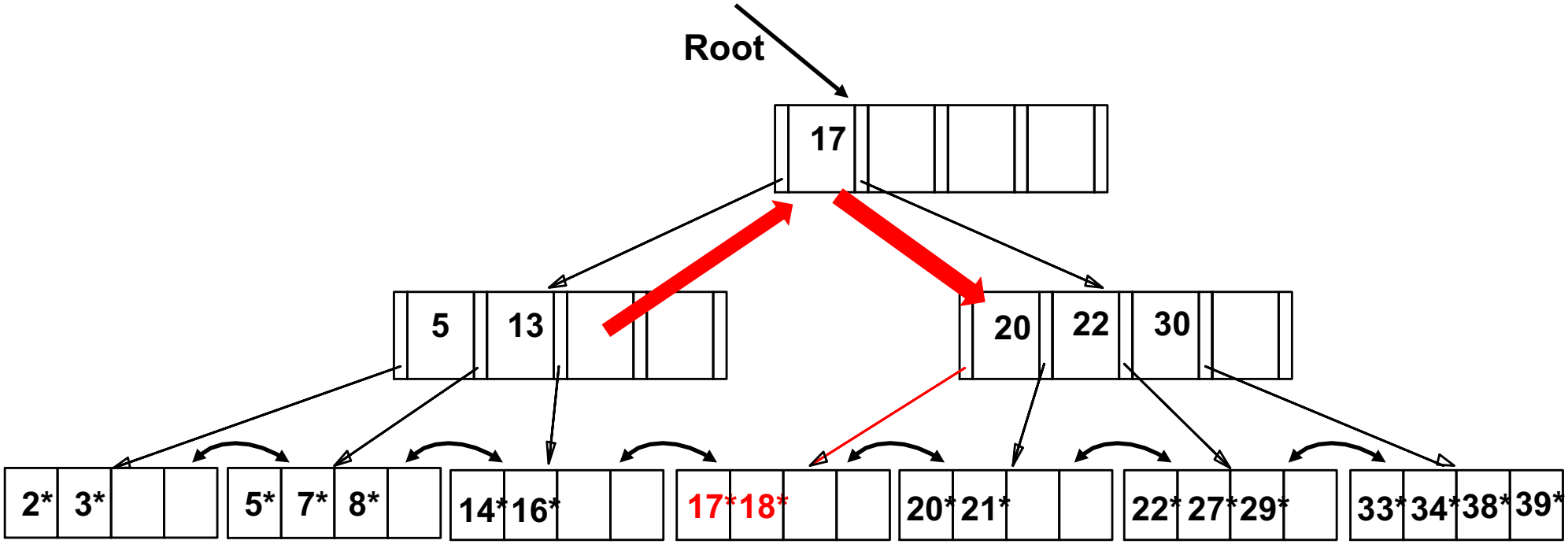
After Re-distribution

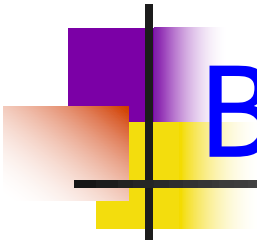
- Rotate through the parent node
- re-distribute index entry with key 20



After Re-distribution

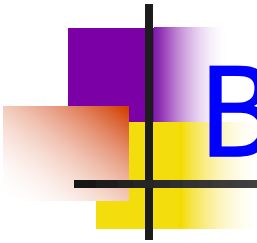
- It suffices to re-distribute index entry with key 20; For illustration 17 is also re-distributed





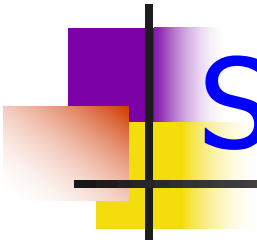
B+-Tree Deletion

- Try redistribution with all siblings first, then merge. Why?
  - Good chance that redistribution is possible (large fanout!)
  - Only need to propagate changes to parent node
  - Files typically grow not shrink!



B+ Tree Operations

- Search
  - Equality
  - Range
- Insert data entry
- Delete data entry
- **Bulk load**

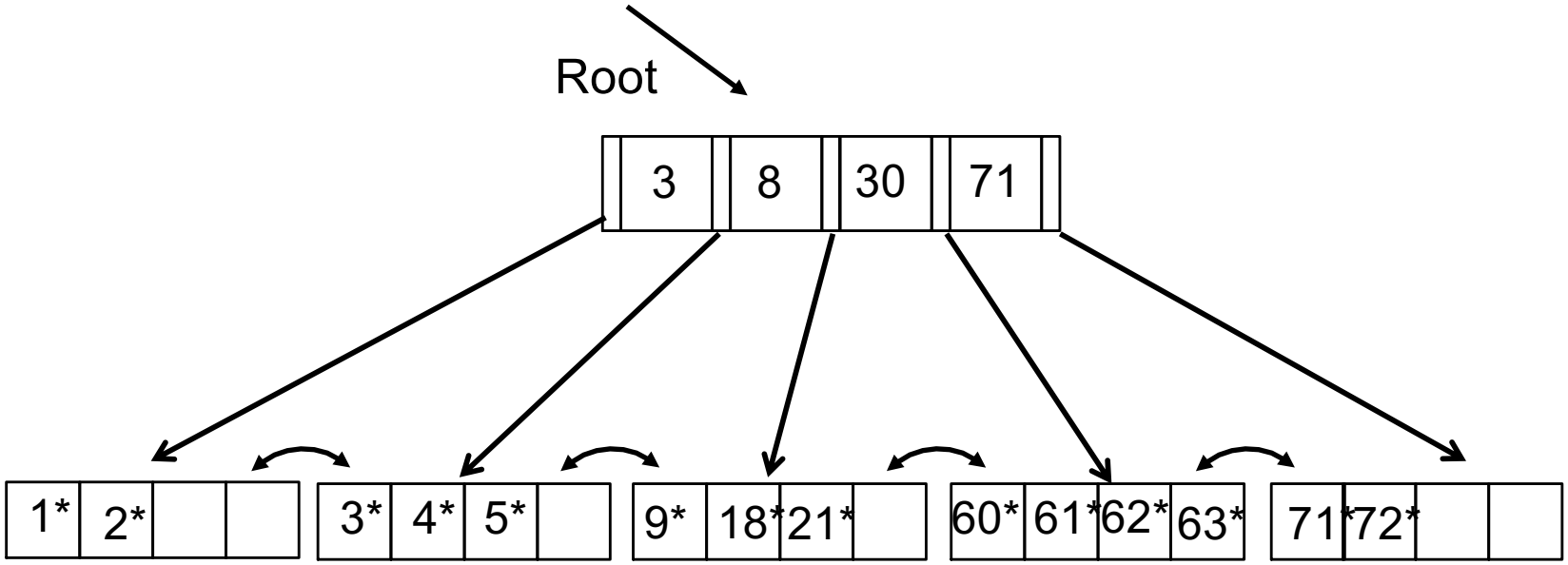


Summary

- Tree-structured indexes are ideal for range-searches, also good for equality searches.
- B+ tree is a dynamic height-balanced index structure.
  - Inserts/deletes/search costs $O(\log_F N)$.
  - High fanout (**F**) means depth rarely more than 3 or 4.
  - Most widely used index in database management systems because of its versatility. One of the most optimized components of a DBMS.

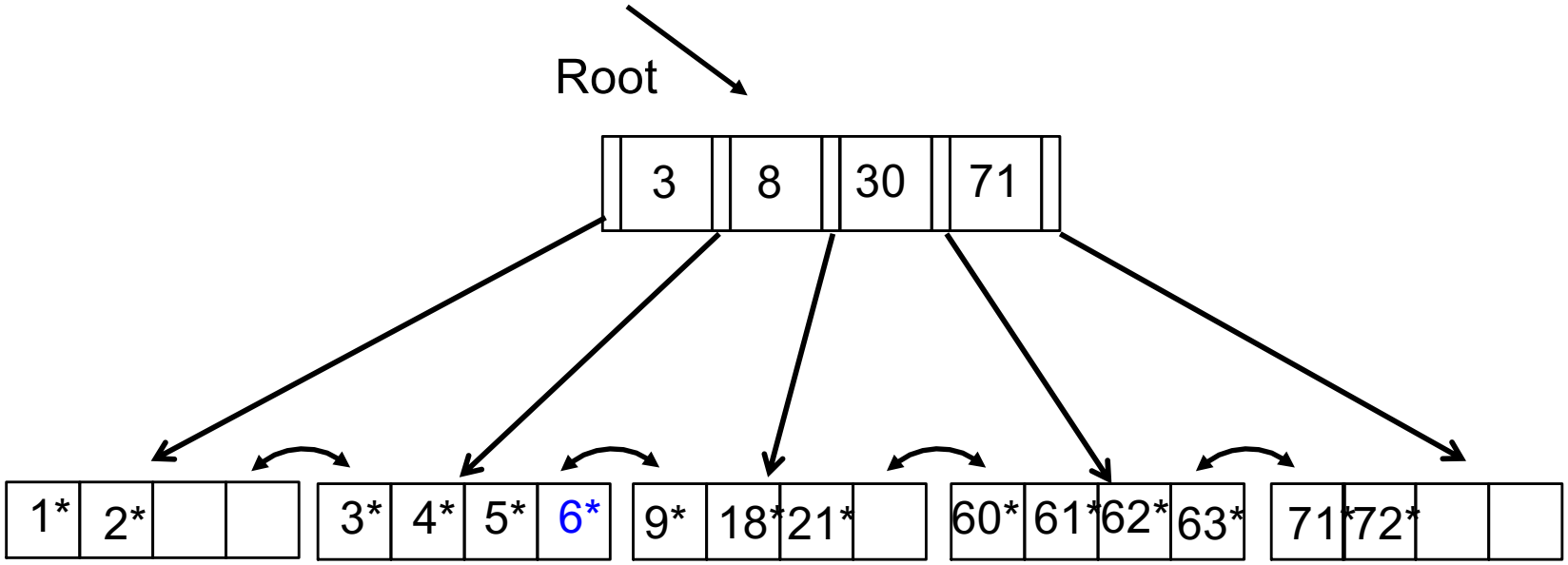
Pop Quiz

- Ins 6, ins 7, del 21, del 4 (in this order)
- Do not consider re-distribution for insertions (but redistribute for deletions)



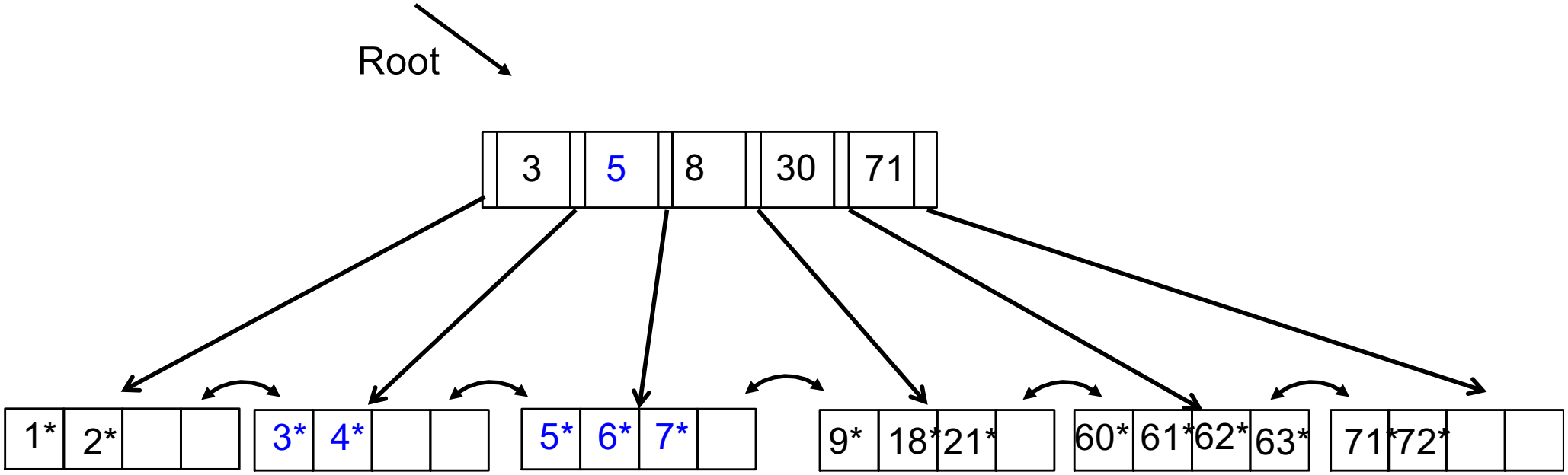
Answer

- Inserting 6 is simple



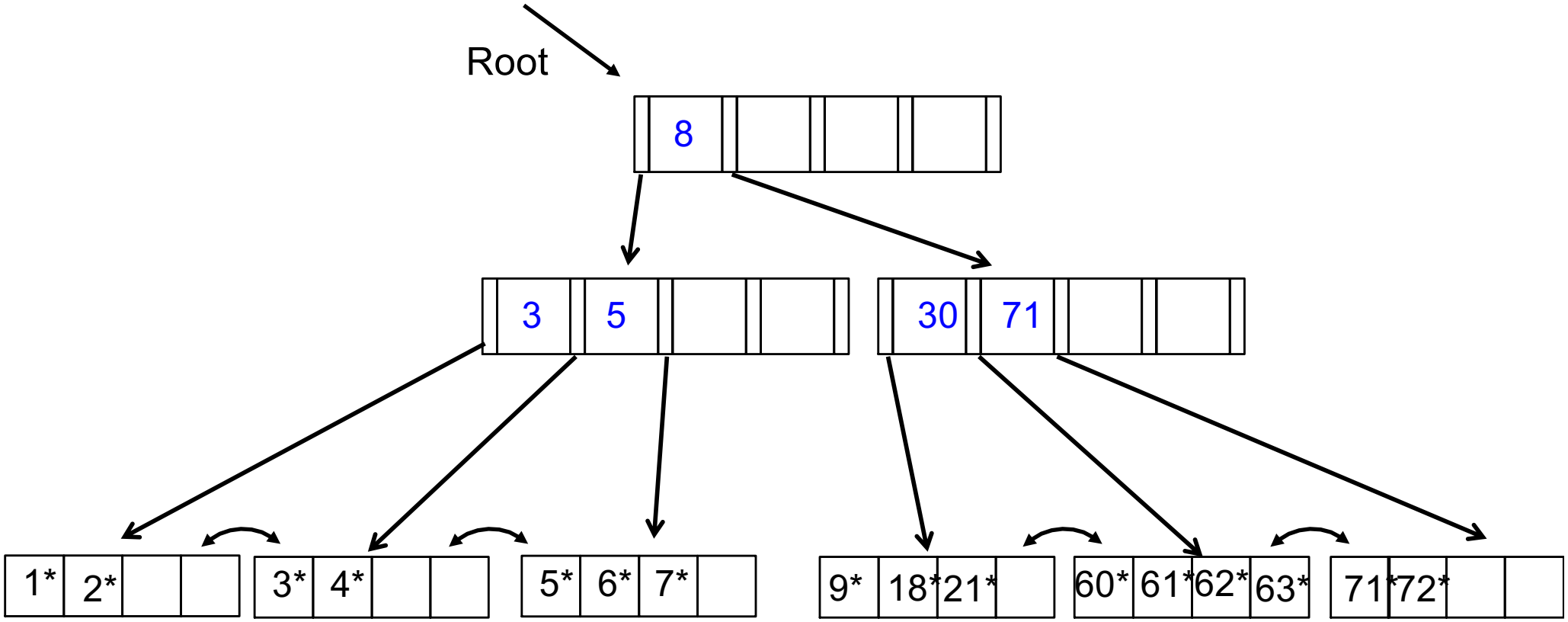
Answer

- Inserting 7 causes a split
- We need to split its parent (here, root)



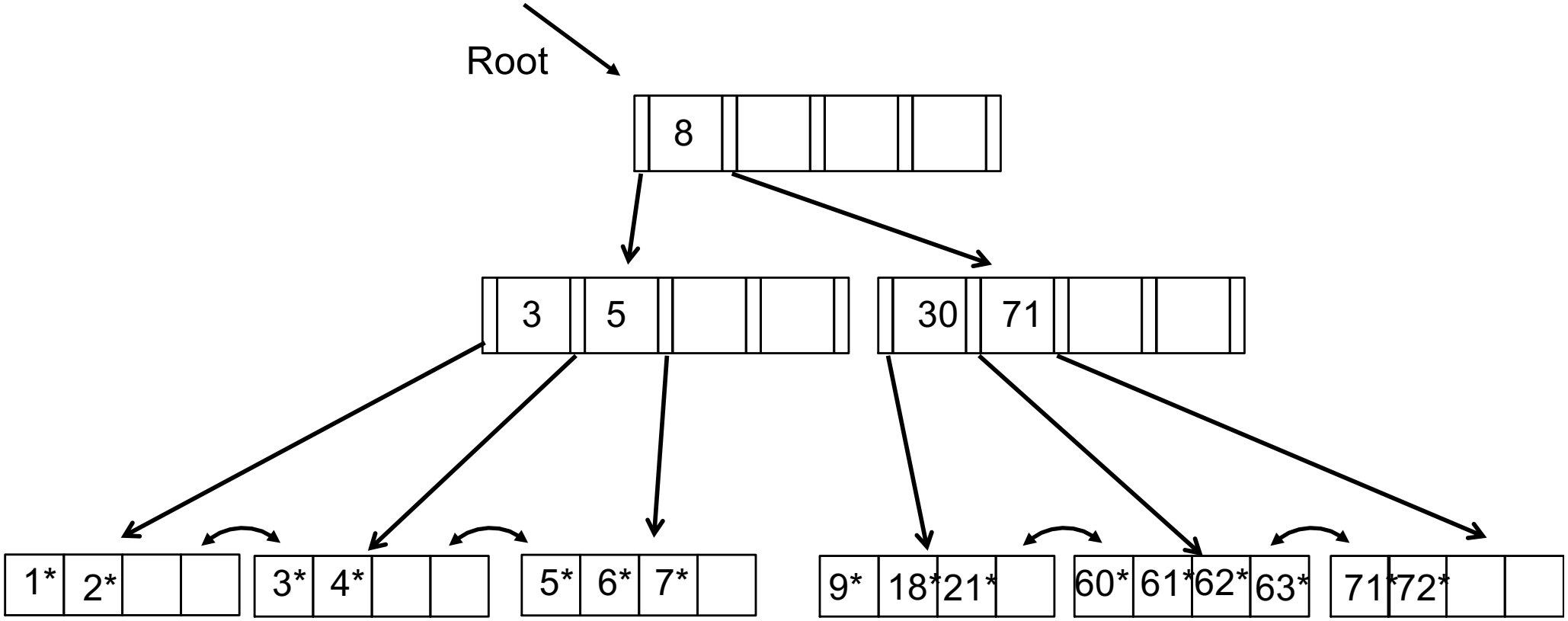
Answer

- Now we have a proper B+-tree again



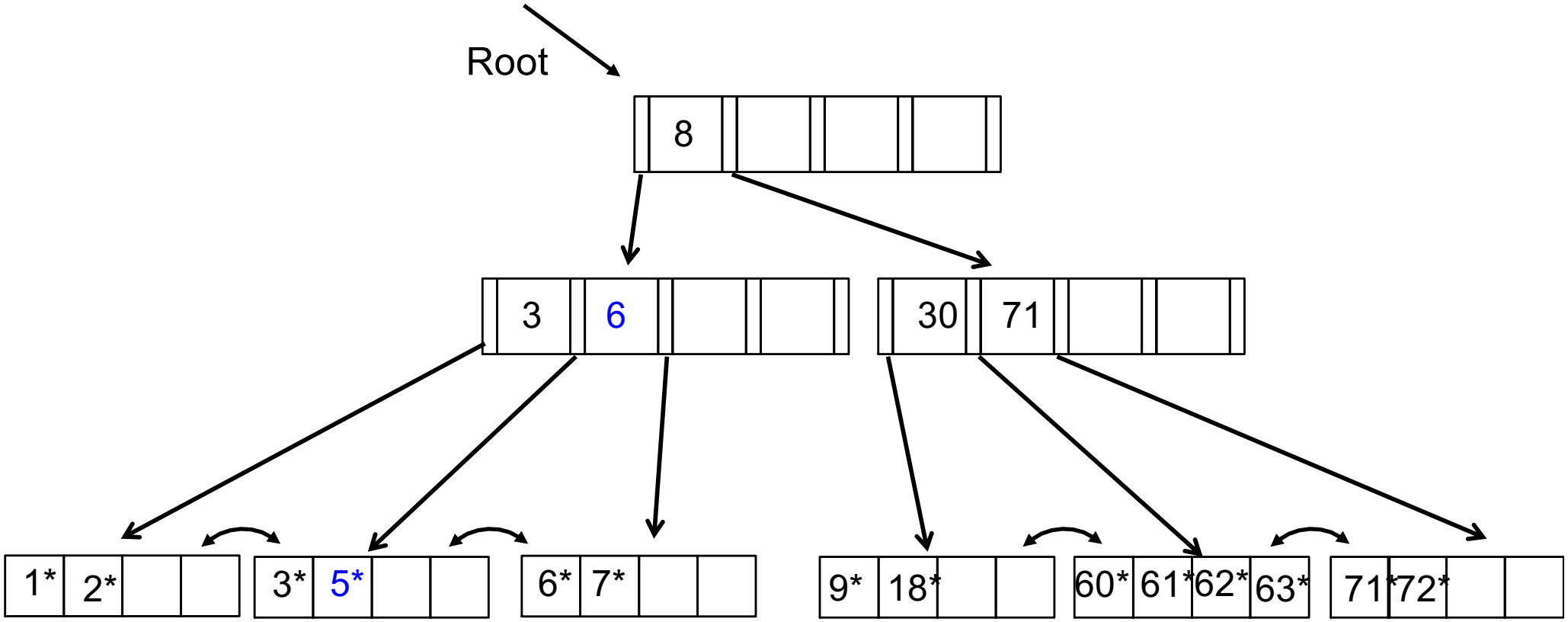
Answer

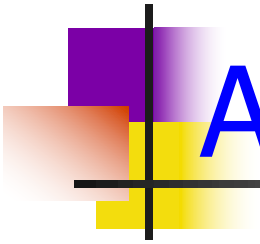
- Delete 21 (easy)
- Finally, delete 4



Answer

- Now we need to borrow from sibling





Announcements

- Suggested Exercises: 10.1, 10.5, 10.7
- Suggested readings for next lecture:
  - Read the entire chapter 11 (hash index)