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

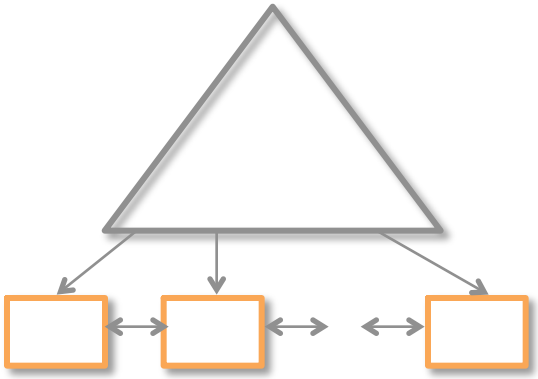
1. Actual Data record
   index = file
2. <k, rid>
   actual records in a diff file
3. <k, list of rids>

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 / range queries.

B+ Tree

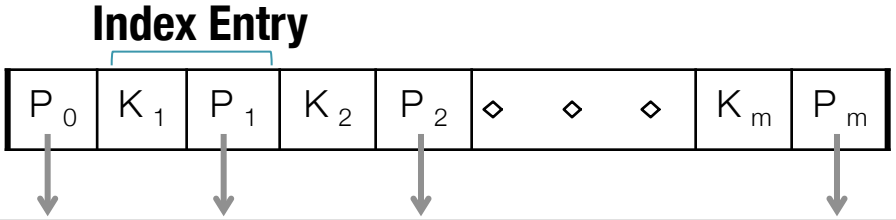
- Height-balanced (dynamic) tree search structure
  - Supports equality and range-searches efficiently
- Main Characteristics:
  1. Minimum 50% occupancy (except for root), i.e. each node contains d <= *m* <= 2d entries, where d = order of the tree.
  2. Height of tree: the length of any path from the root to a leaf.
  3. Remains balanced after insert/delete operation.



Index entries (direct search)

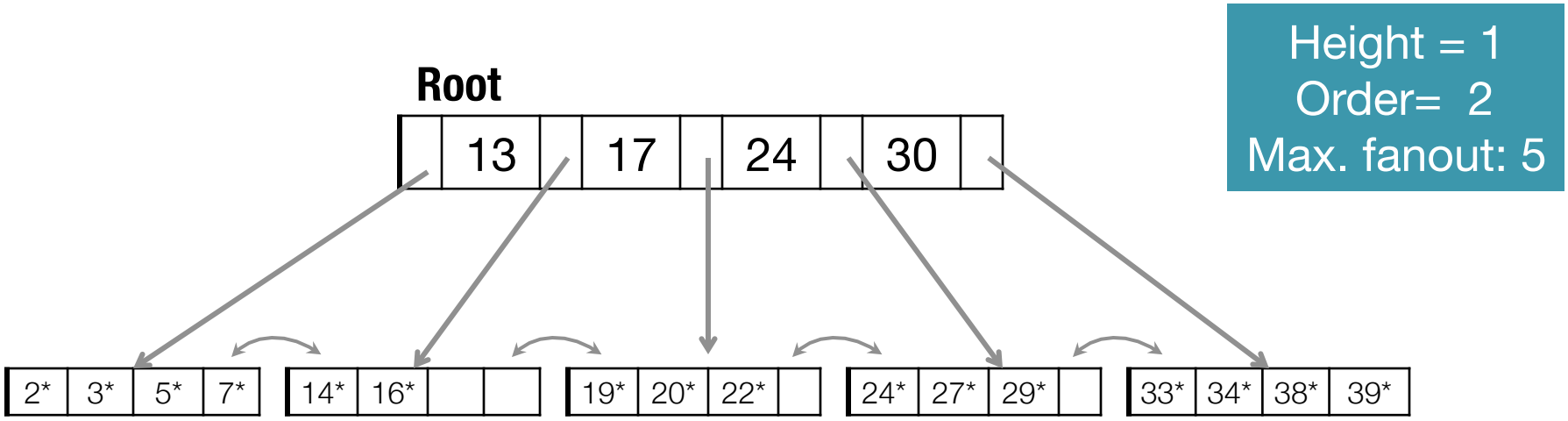
Data entries

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



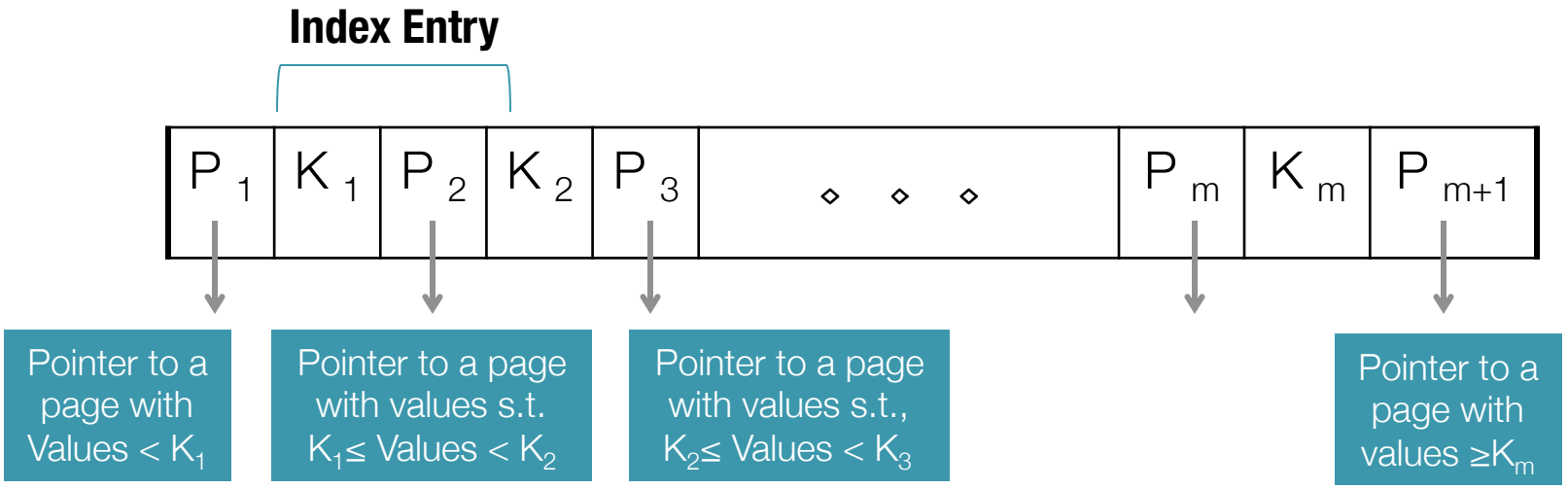
Searching in a B+ Tree

- 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.
- Try it out: Search for 5*, 15*, all data entries >=24*

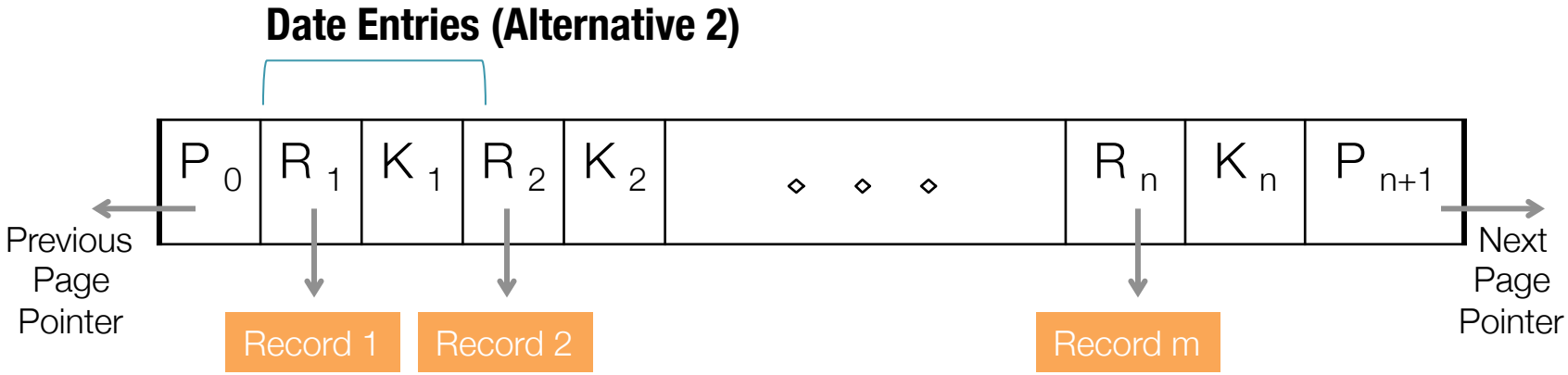


B+-Tree Page Format

Non-leaf Page



Leaf Page



B+ Trees: Height

What is the height of a B+ tree? (formula)

Definitions:

F: fanout (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 66.5%
  - 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

Arithmetic Example

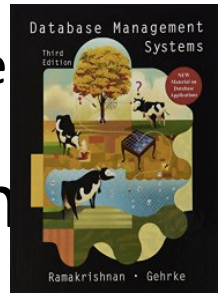
- You are given a file of 10 million records
- Suppose you can store 10 data entries per leaf page
- You build a B+ Tree with order 75, 67% average fill-factor

- What is the avg fanout?
- What is the height of your B+ Tree?

A Note on Order

- Some literature uses: order = max # of entrie
- **In this class**: order = minimum # of en

  (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

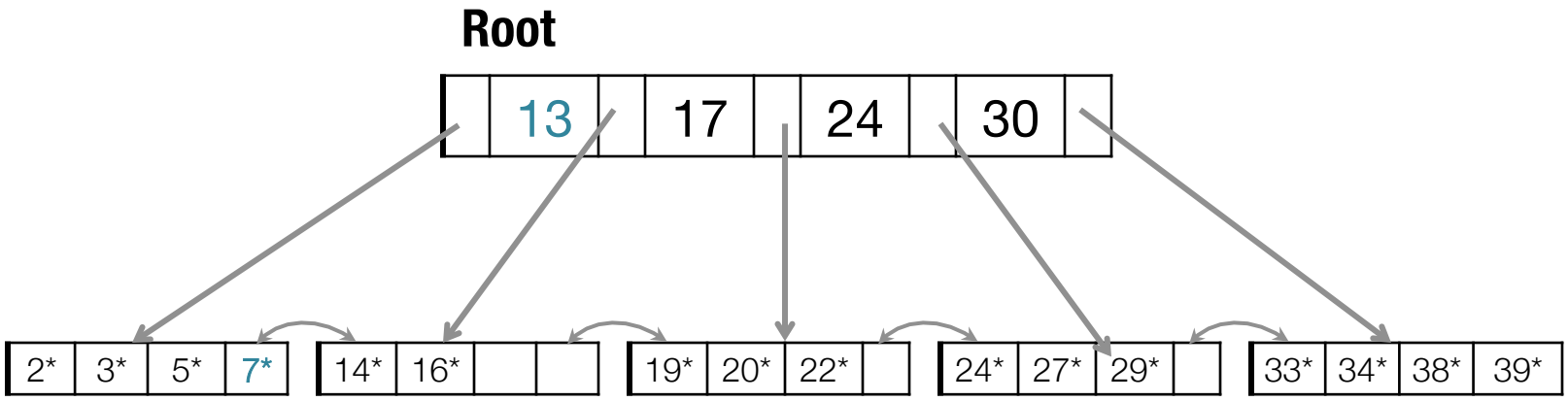
- Search
  - Equality
  - Range
- Insert data entry
- Delete data entry
- Bulk load

B+ Tree: Inserting a Data Entry

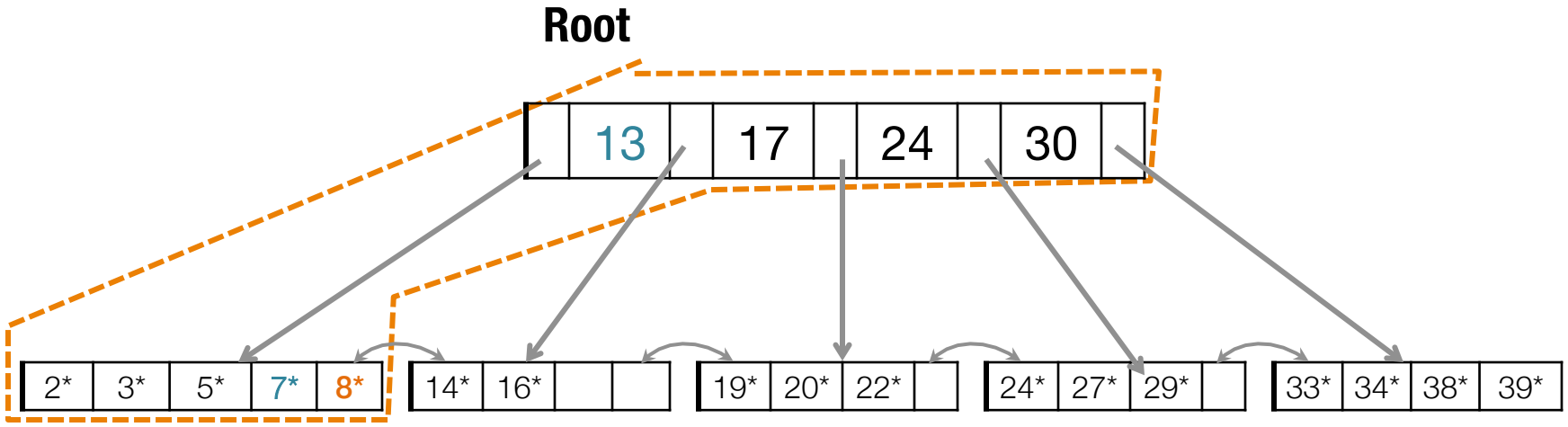
- Maintain invariants:
  - Search-tree property
  - All nodes must be at least ½ full (except root node), i.e., has between d and 2d entries
  - Root node is allowed to have a single entry

- Strategy:
  - Split nodes when they become full and a node is added:
    - Split an overfull node with (2d + 1) entries into two nodes with d and (d+1) entries, resp.

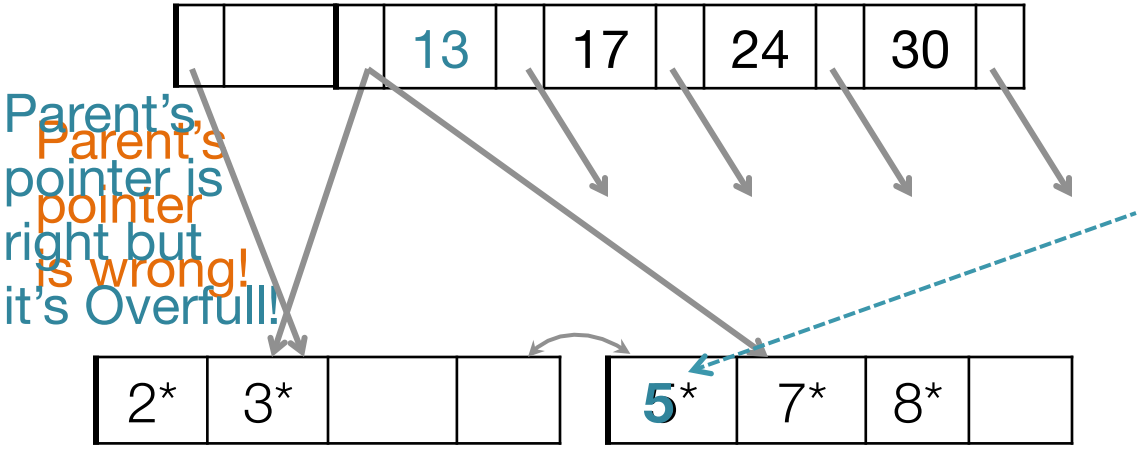
Inserting 8* into B+ Tree



Inserting 8* into B+ Tree



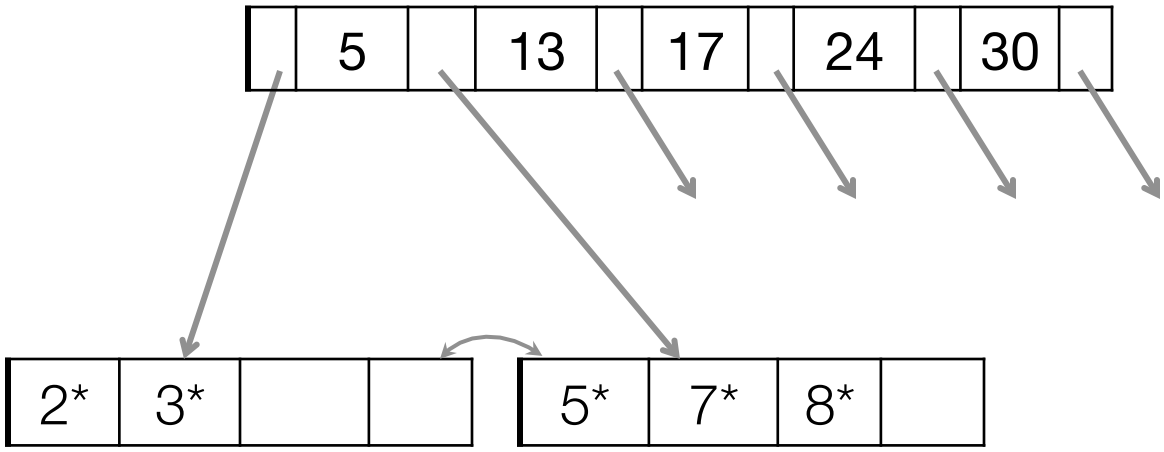
Overfull



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

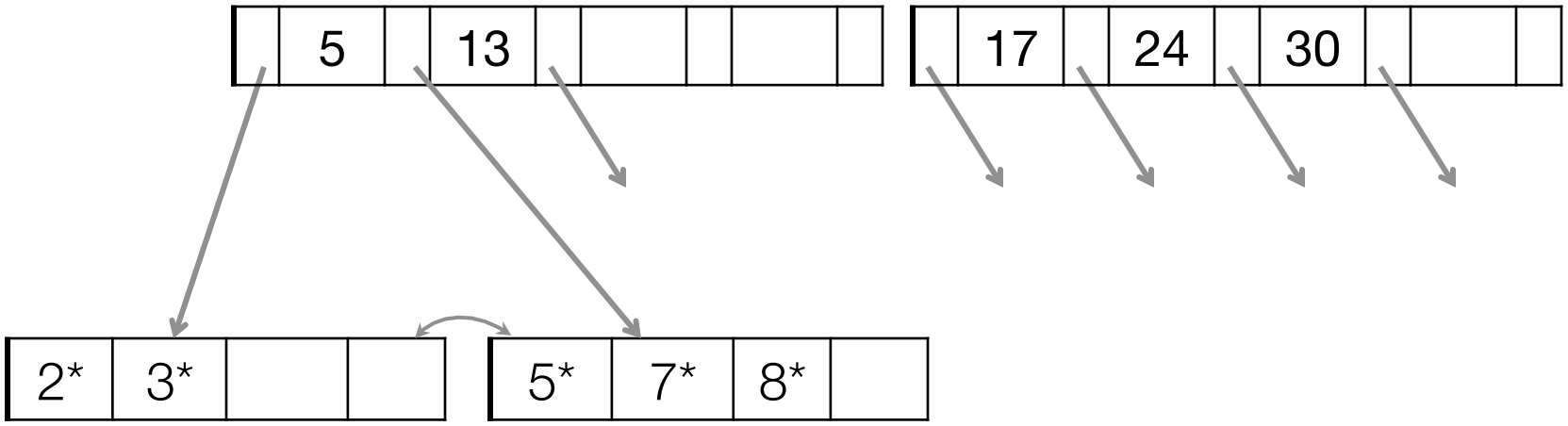
Splitting Overfull Non-Leaf Nodes

Overfull

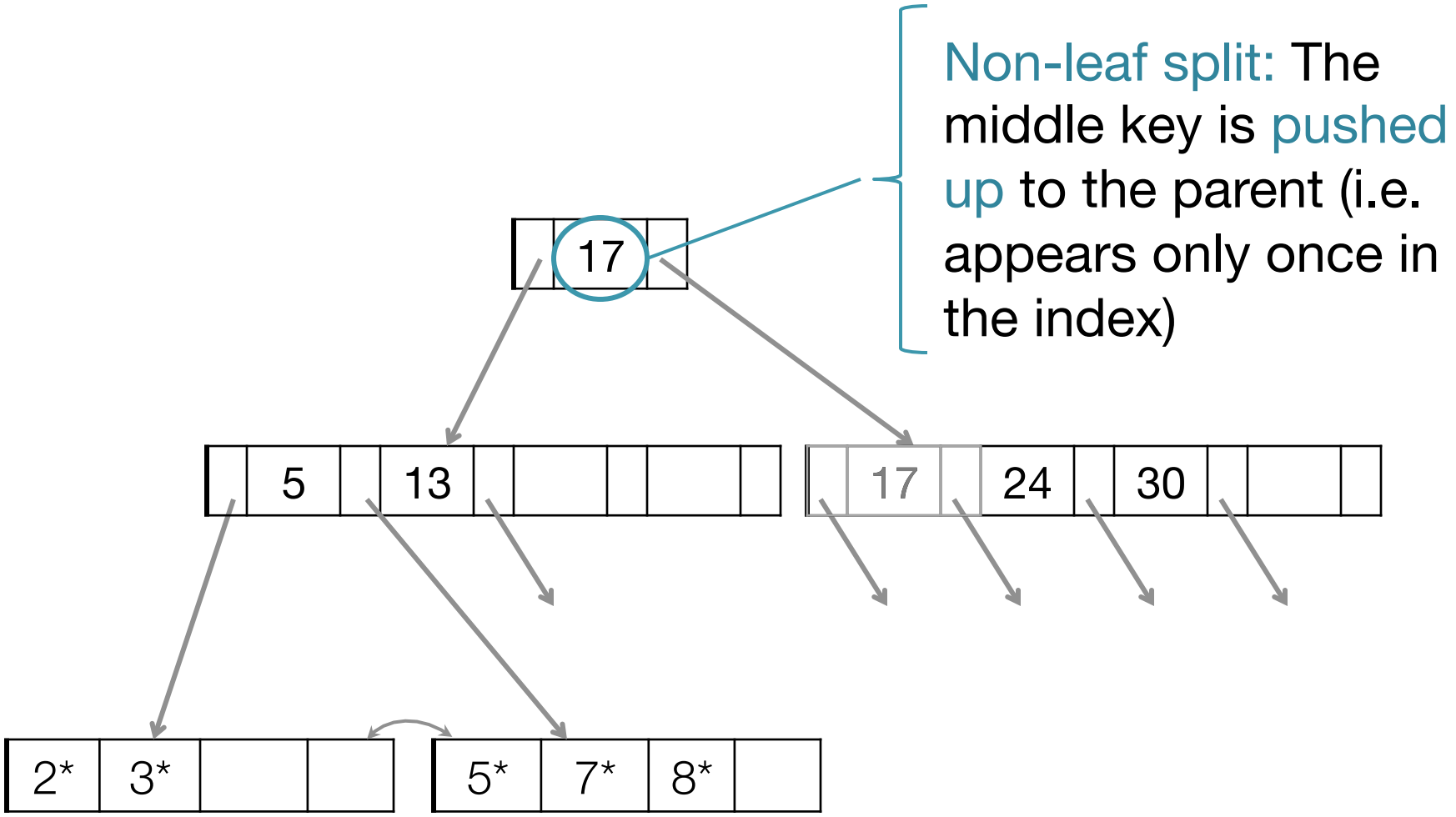


Splitting Overfull Non-Leaf Nodes

Split

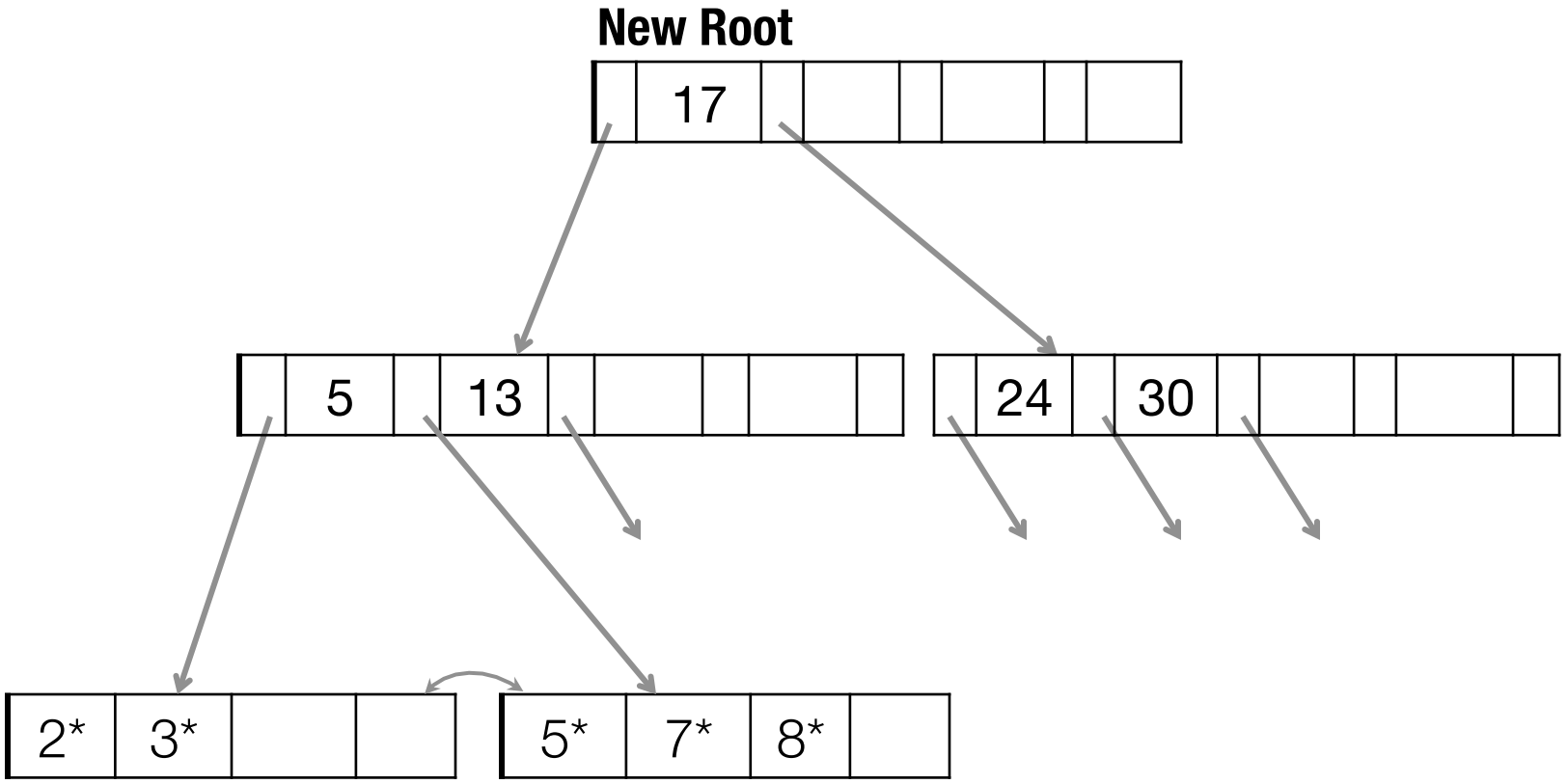


Splitting Overfull Non-Leaf Nodes

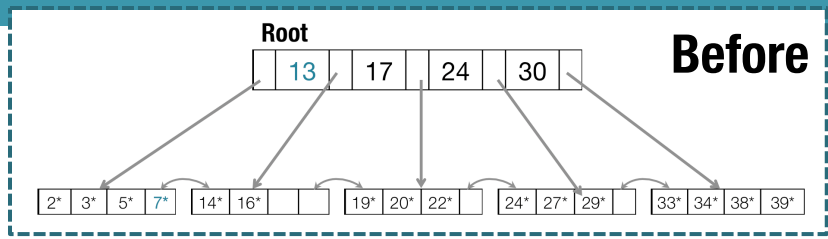


Splitting Overfull Non-Leaf Nodes

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

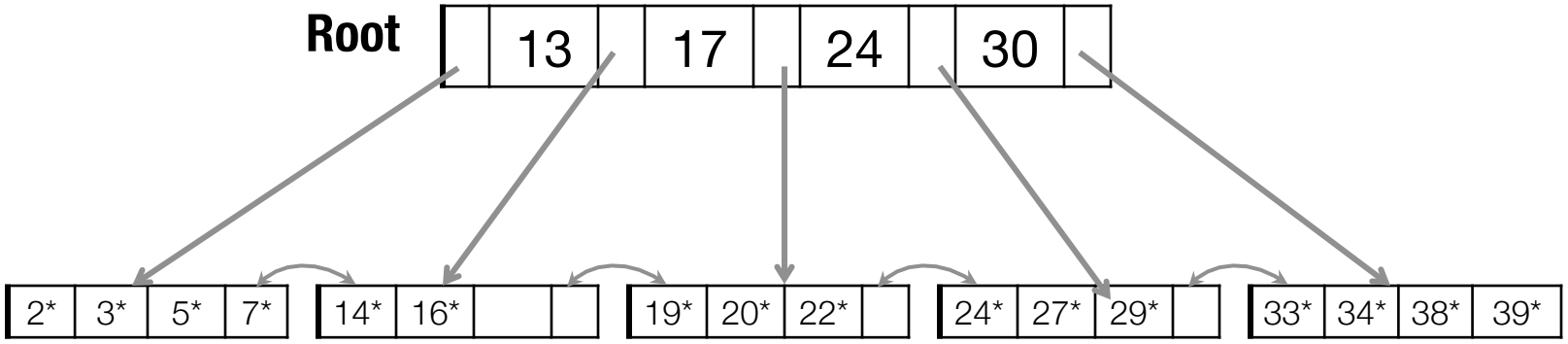


The B+-tree After Inserting 8*

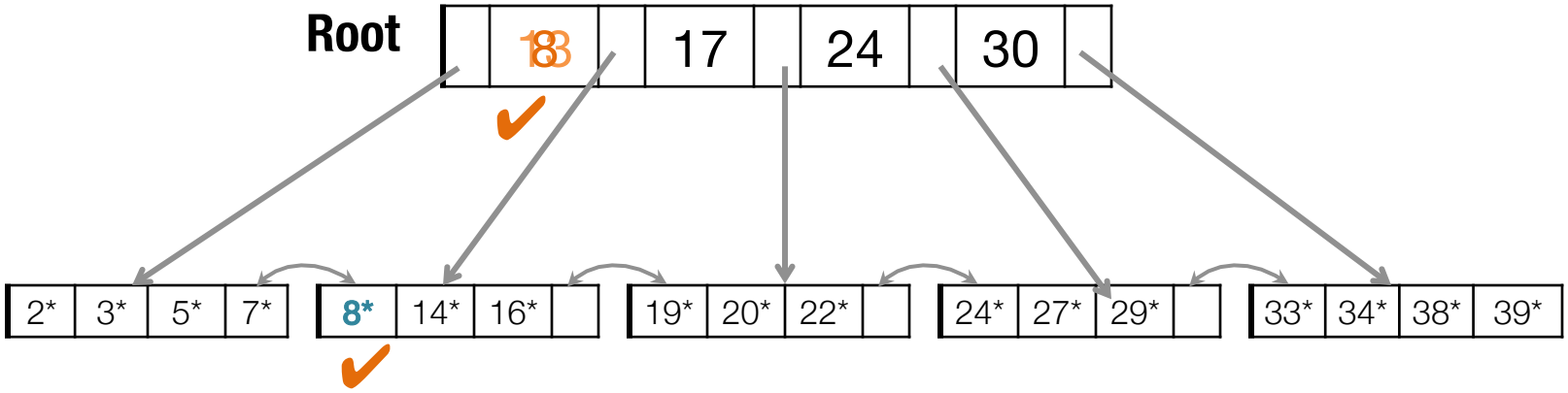


- Root was split: height increases by 1
- Could avoid split by re-distributing entries with a sibling
  - Sibling: immediately to left or right, and same parent

Inserting 8* via Entry Re-distribution with Siblings



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

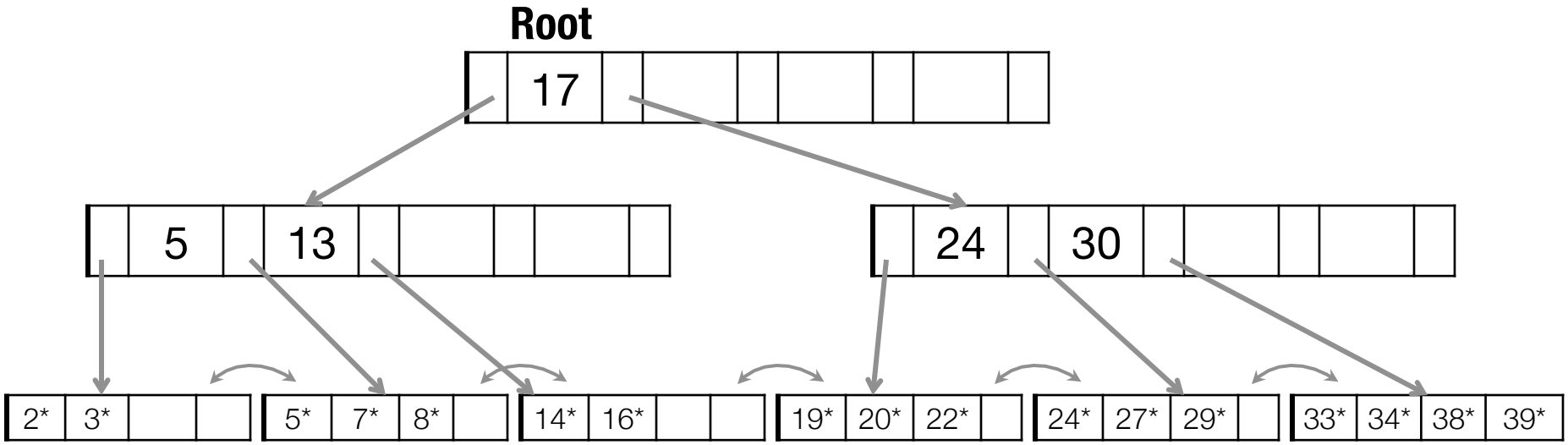
- Search
  - Equality
  - Range
- Insert data entry
- Delete data entry
- Bulk load

B+-Tree: Deleting a Data Entry

1. Find the data entry (will always be at a
2. Delete it
3. 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
4. On merge, delete relevant entry in parent

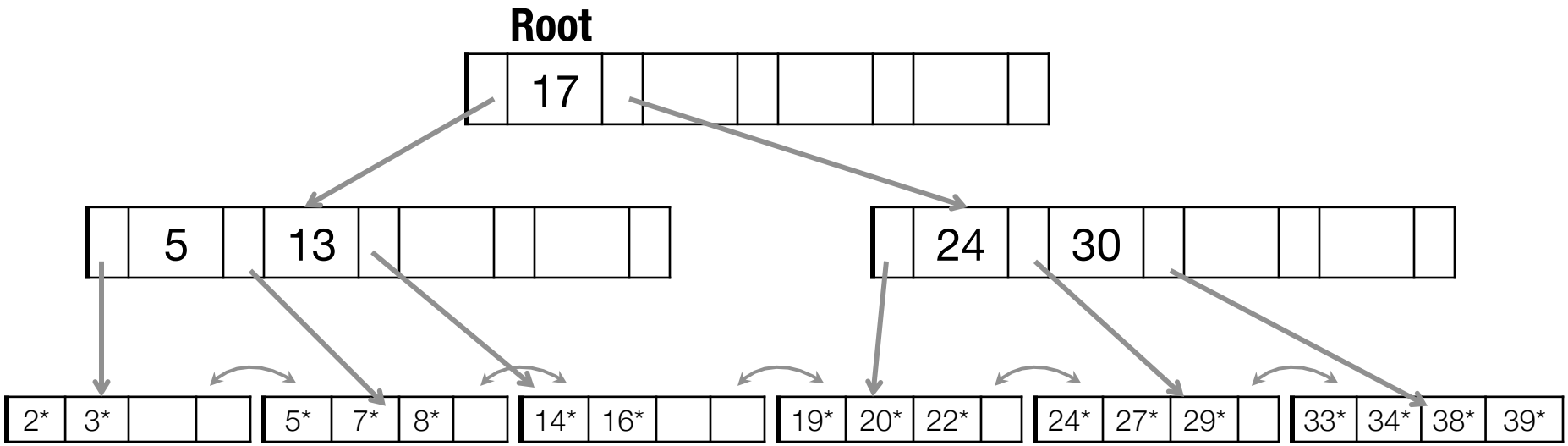
Merge could propagate to root, decreasing height.

Example Tree



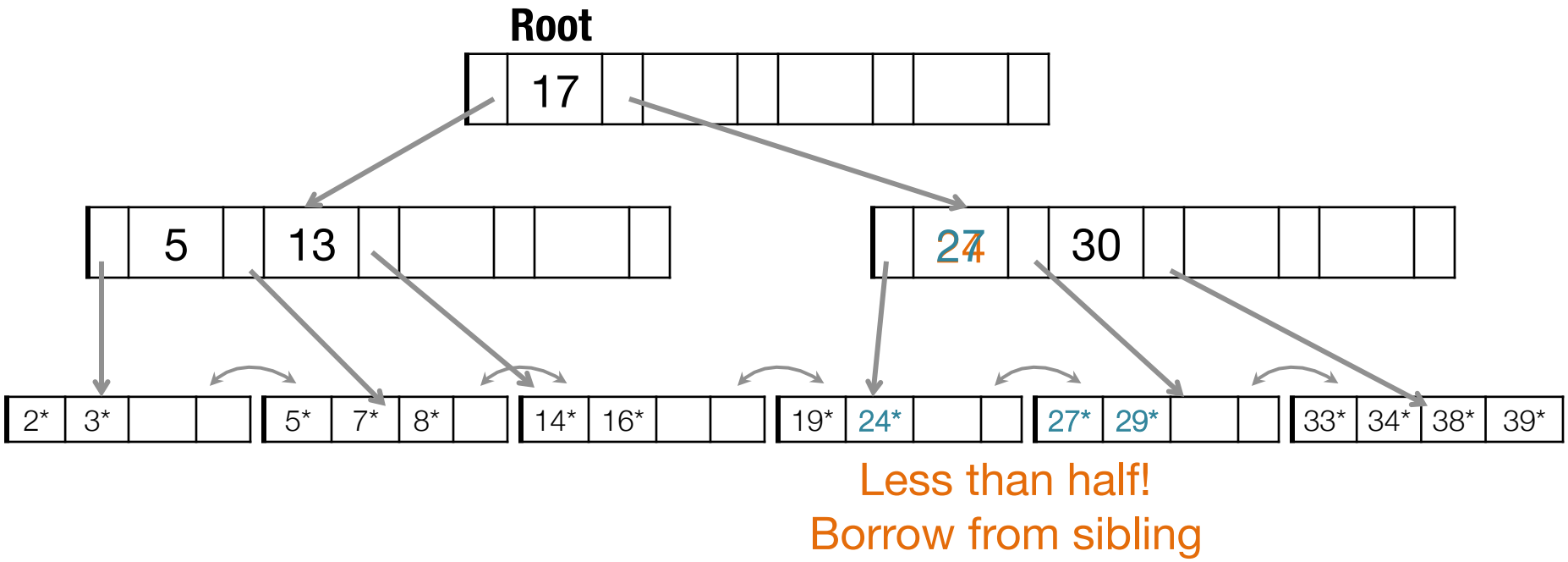
- **Task**: Delete 22, 20, 24
- Deleting 22 is easy. Invariant maintained.
- Deleting 20 is harder. Node would become less than half full.

Deleting 22* and 20*



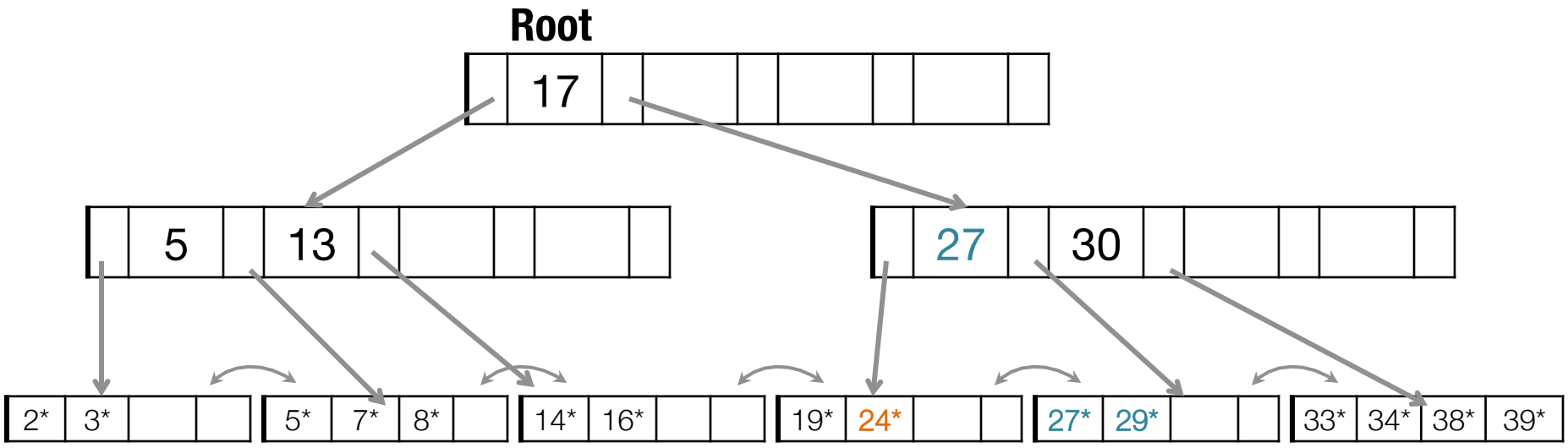
Less than half!
Borrow from sibling

Deleting 22* and 20*



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

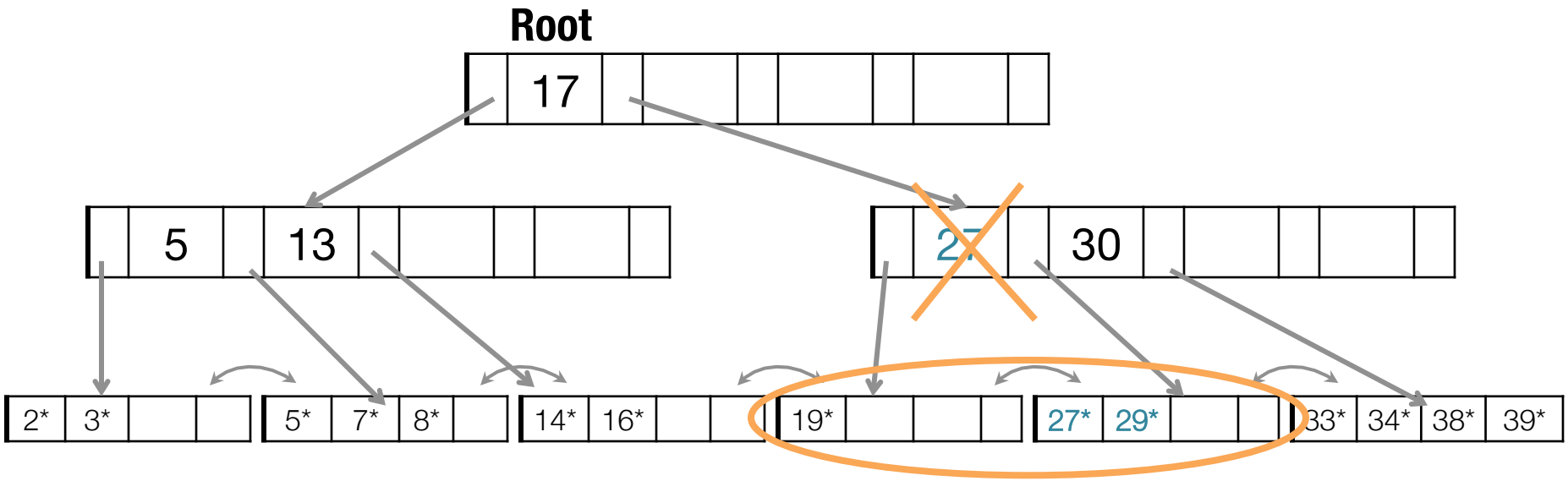
… And then Deleting 24*



Less than half!
And cannot borrow
from its sibling!

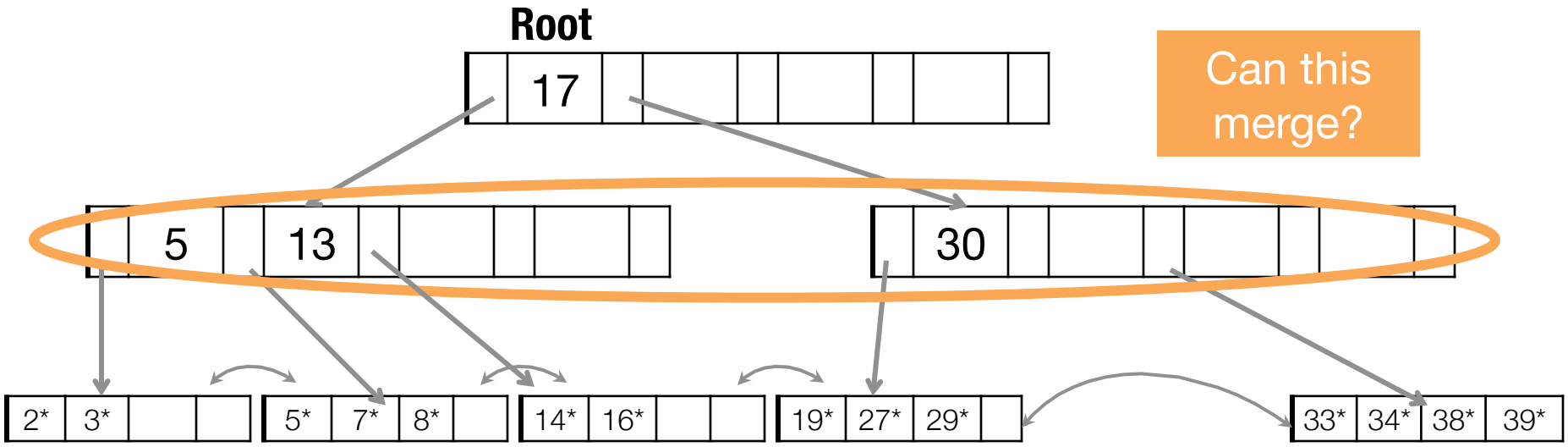
- Must merge
- In the non-leaf node,
  toss the index entry with key value = 27

… And then Deleting 24*

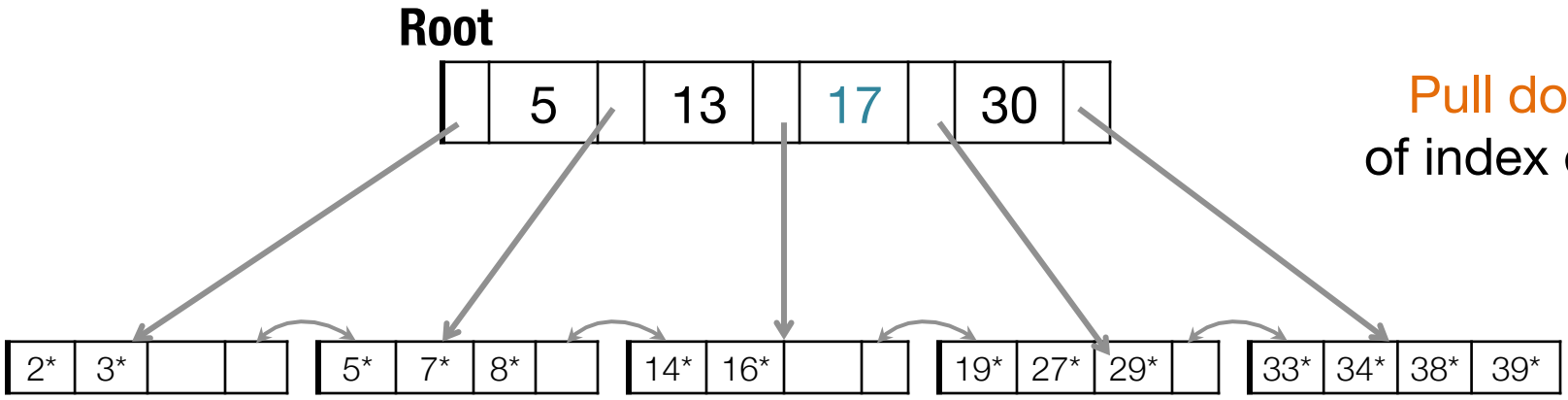


- Must merge
- In the non-leaf node,
  toss the index entry with key value = 27

… And then Deleting 24*



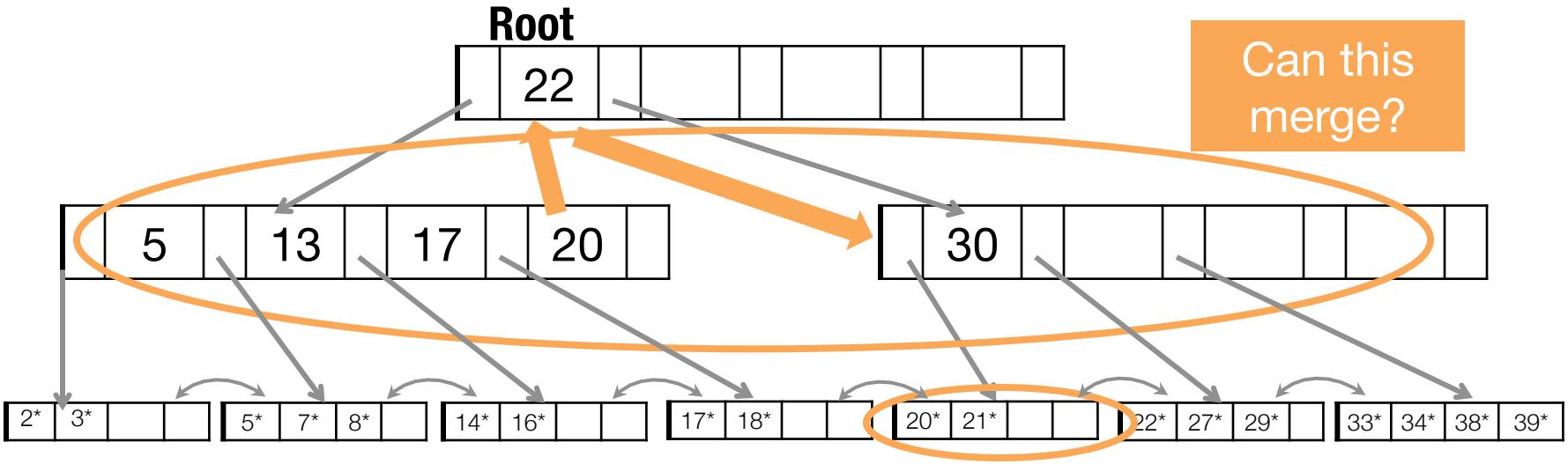
Can this merge?



Pull down of index entry

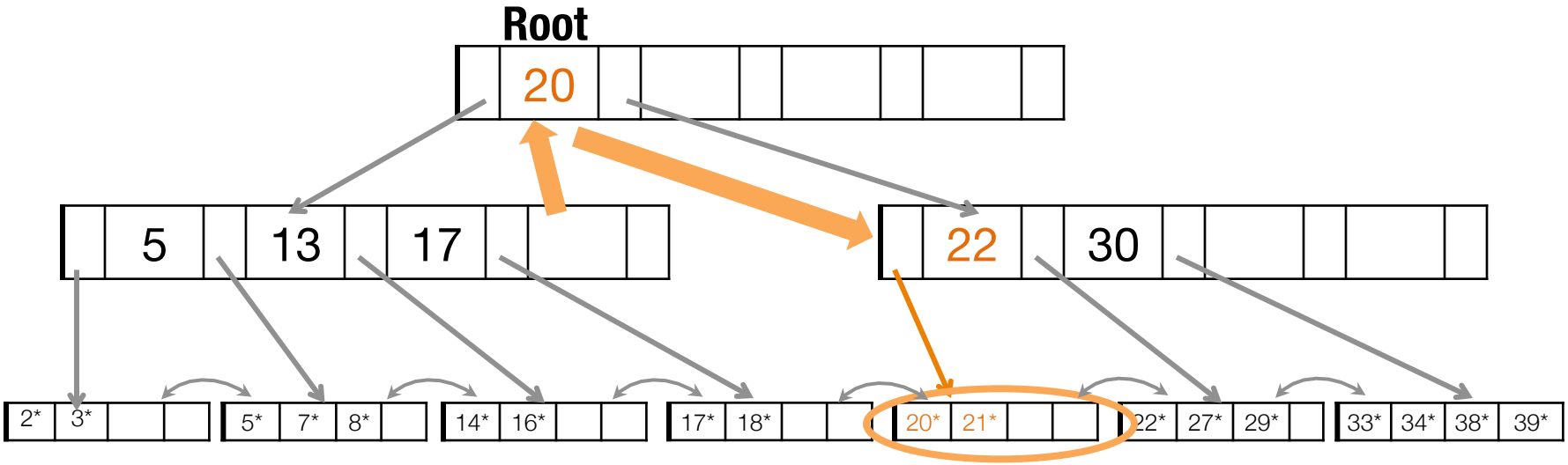
Another Deletion Example: Non-leaf Re-distribution

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



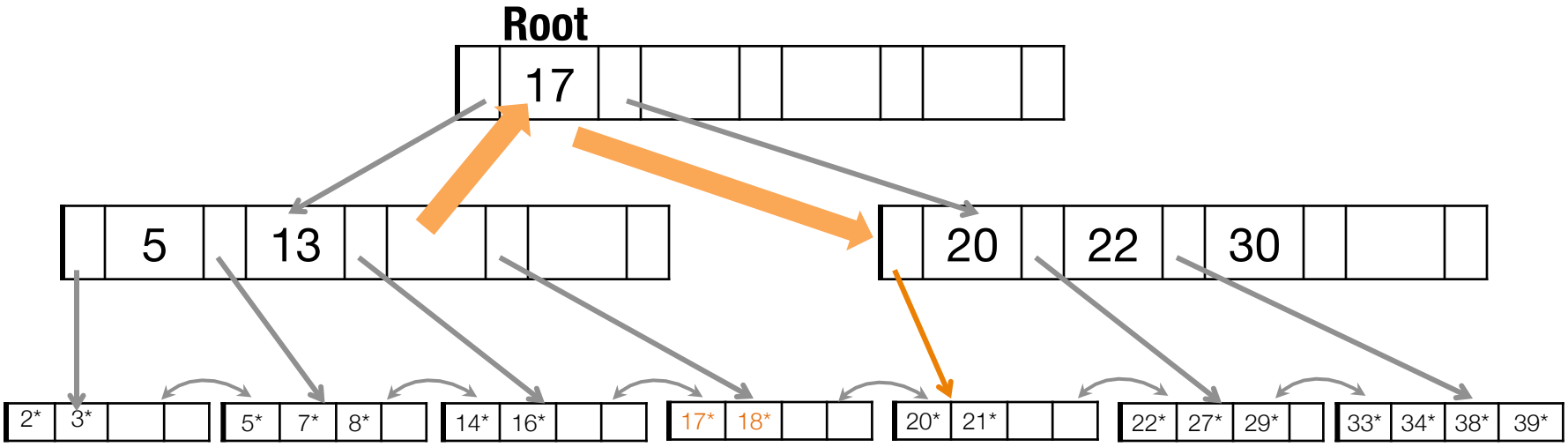
After Re-distribution

- 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

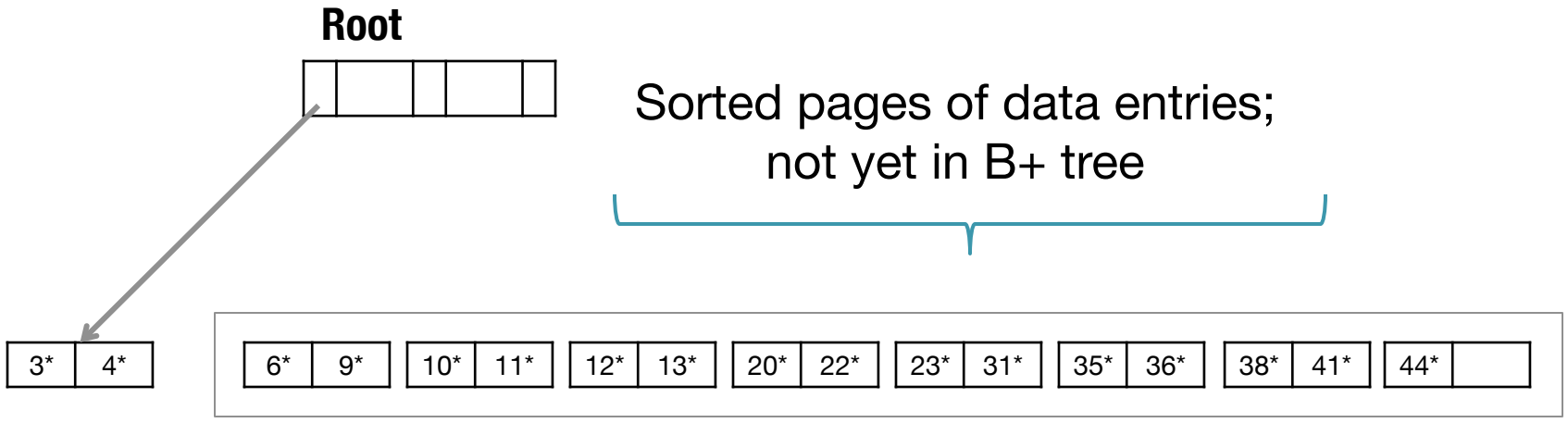
- Search
  - Equality
  - Range
- Insert data entry
- Delete data entry
- Bulk load

Adding Entries to a B+ tree

- Option 1: multiple inserts
  - Slow. Repeated re-organization
  - Does not give sequential storage of leaves
- Option 2: Bulk Loading
  - Fewer I/Os during build
  - Leaves will be stored sequentially (and linked, of course)
  - Can control "fill factor" on pages

Bulk Loading of a B+ Tree

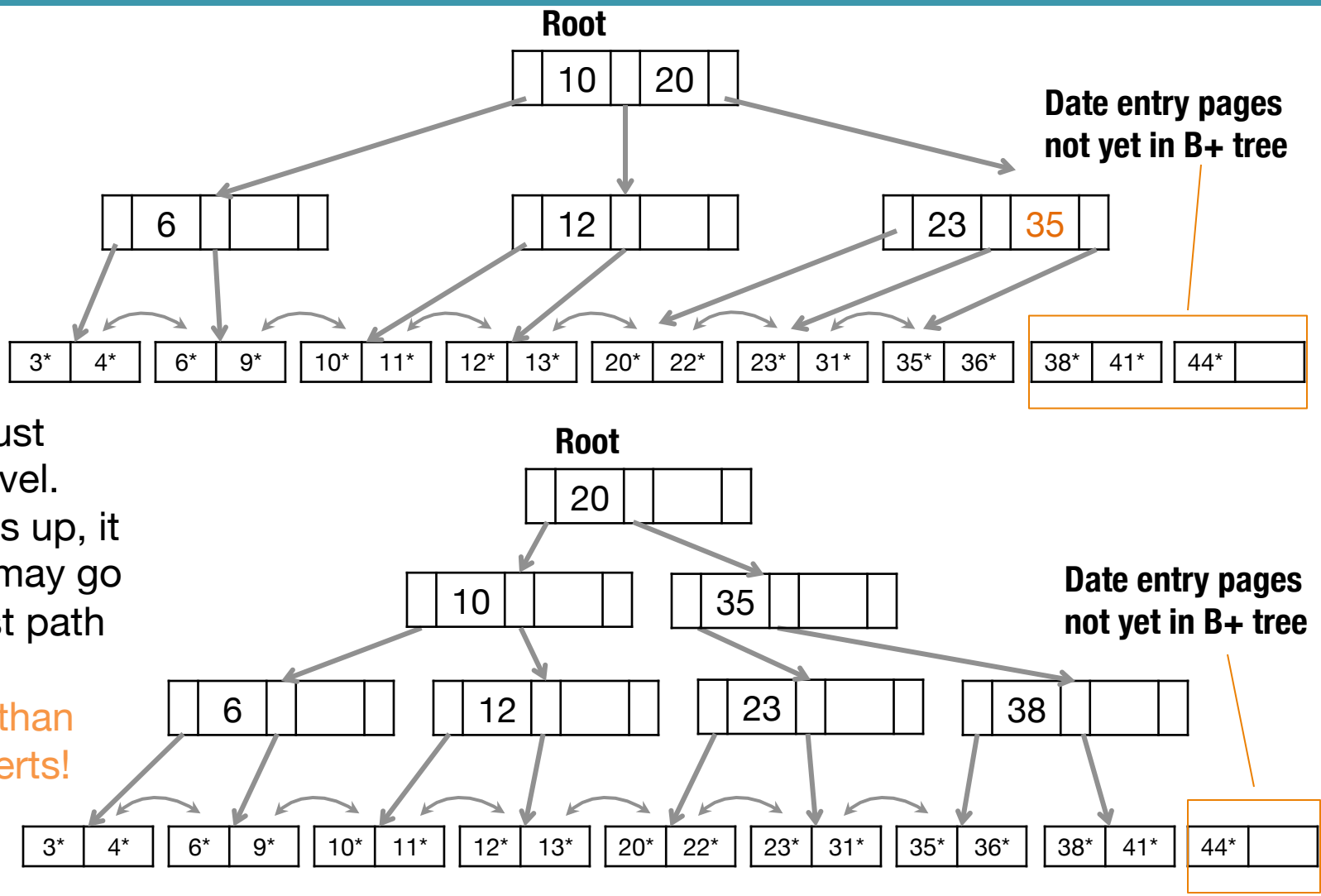
Initialization: Sort all data entries, insert pointer to first (leaf) page in a new (root) page.



Bulk Loading (Contd.)

New 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!



Summary

- Tree-structured indexes are ideal for range-searches, also good for equality searches.
- B+ tree is a dynamic height-balanced index structure.
  - Insertions/deletions/search costs $O(\log_F N)$.
  - High fanout (F) means depth rarely more than 3 or 4.
  - Max occupancy = 2d
  - Min occupancy = order = d (at least half full – exc. root)
  - Most widely used index in database management systems because of its versatility. One of the most optimized components of a DBMS.

Suggested Exercises + Readings

Suggested Exercises: 10.1, 10.5, 10.7

Suggested readings for the lecture on hash-based index:

Read the entire chapter 11 (hash index)