Tree-Structured Indexes

Chapter 10

Recap – Storage and Indexing

- Can store a database table in
  - Heap: i.e., no indexing. (Not same as the Heap data structure in EECS 281).
  - Indexed. Create index
    - Searching: Given a search key k, look up matching K* entries, Retrieve record(s) from those entries.
  - Three alternatives for K*: Alternative 1 = record order = entry order
    1. (k, Record). This can be considered to be a clustered index.
    2. (k, RID). RID points to a block/offset/size of Record on the disk
    3. (k, [n, [RID1, RID2, …RIDn])

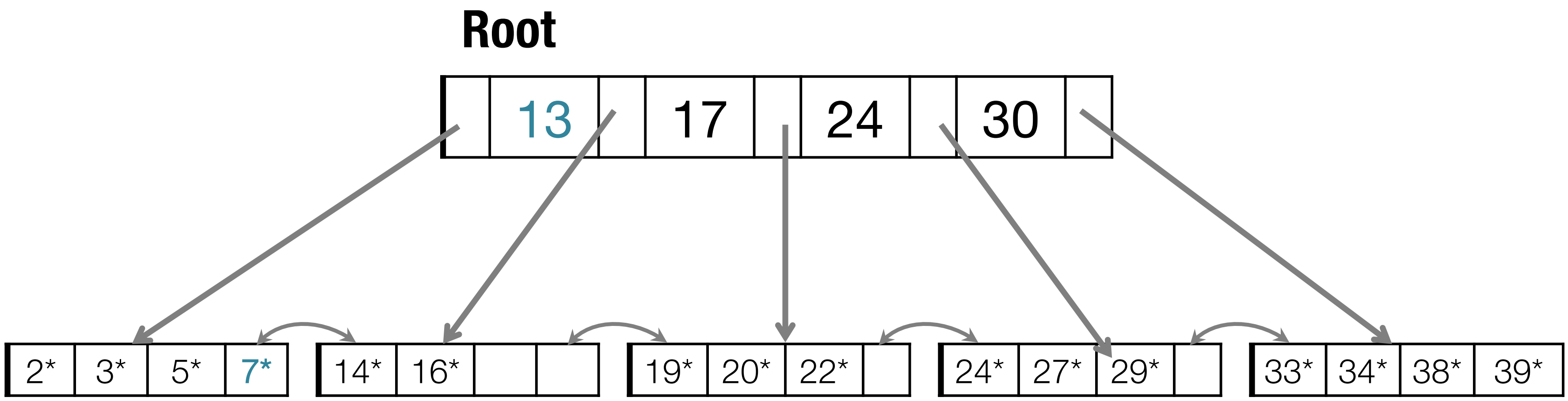
This lecture

- 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: A search tree but persistent with each tree node stored in a disk block

Terminology

- First there was a B Tree.
- People immediately loved it
  - Many people started making improvements to it.
  - B+ Tree, B* Tree, …
- Eventually, all agreed that B+ Tree was the best of these data structures in the BTree family.
- Name "B+ Tree" is a twister
  - Often casually referred to as BTree.

Example B+-tree of height 1



- Height-balanced (dynamic) search tree
- Persistent: Each node in the tree is stored in a disk block!
  - Thus, each tree node has many pointers, unlike a binary search tree!
  - Given a key, say 16, efficient to find its entry, 16*.
  - Range queries (e.g., all records between 16-25) also can be done efficiently

B+ Tree

- Additional Properties:

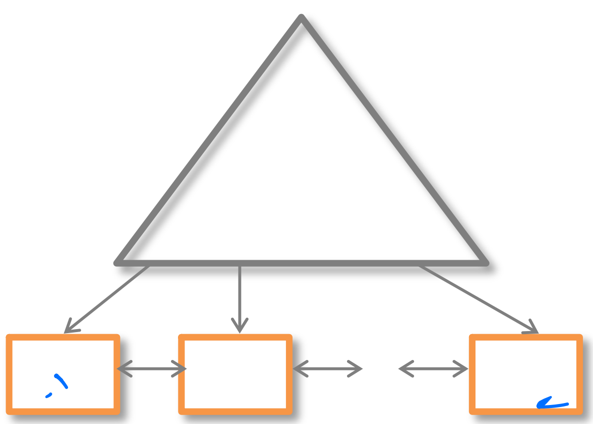
min amount of entry in a page

1. **Invariant:** Minimum 50% occupancy (except for root), i.e. each node contains d <= *m* <= 2d entries, where d = order of the tree. Root has min of 1 entry and max of 2d entries
2. **Invariant:** Remains balanced after insert/delete operation

Height of tree: the length of any path from the root to any leaf.

2d entries : dividers
⇒ 2d+1 children

Fan out = # children for non-leaf node

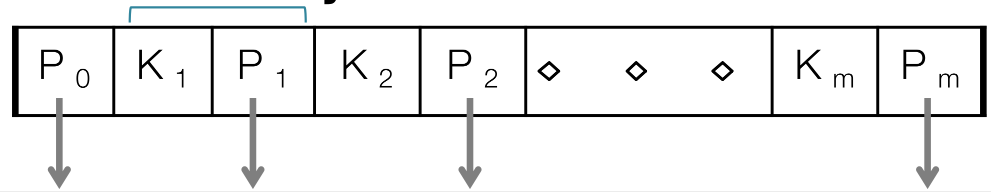


Index entries (direct search)

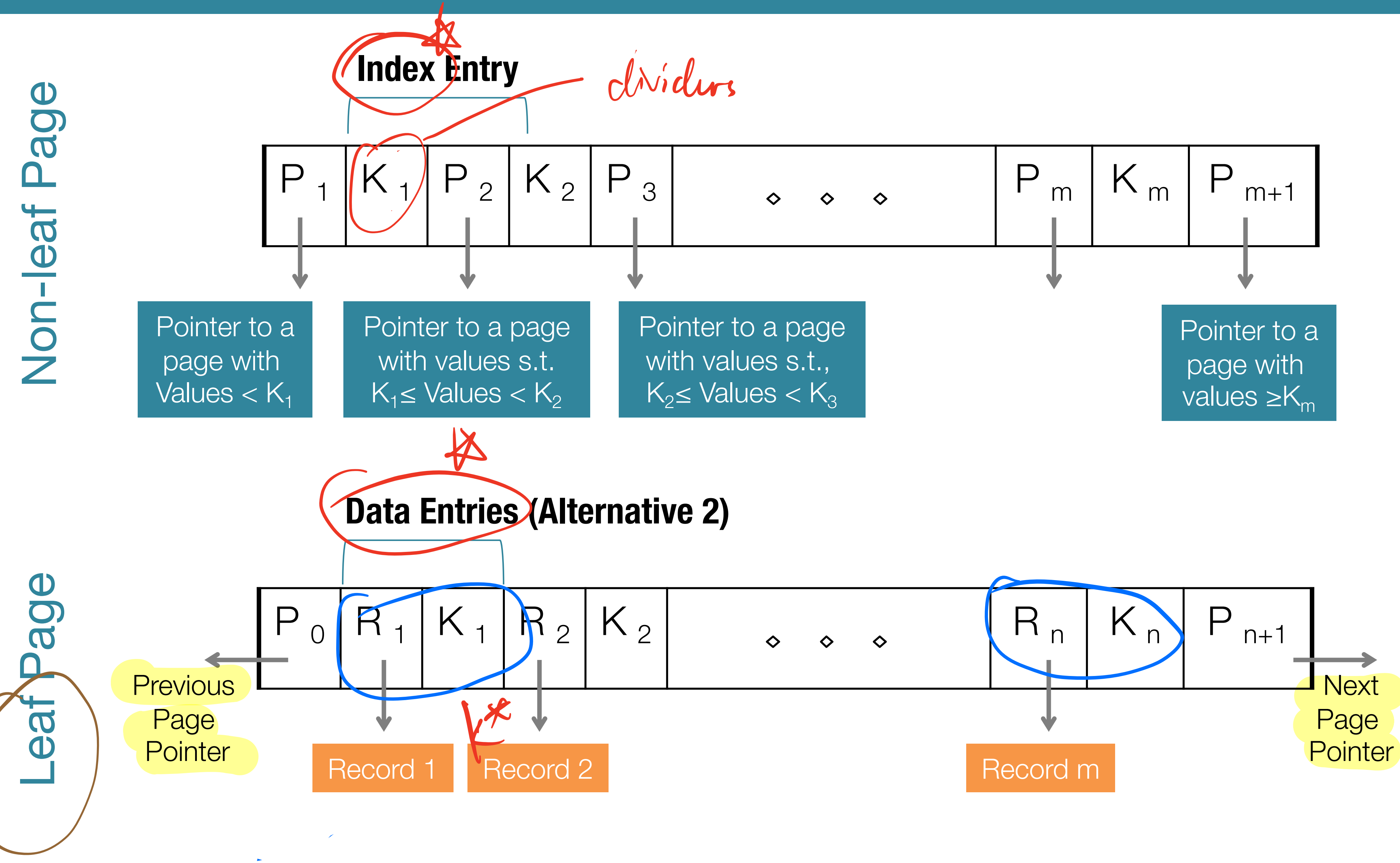
Data entries

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

Index Entry

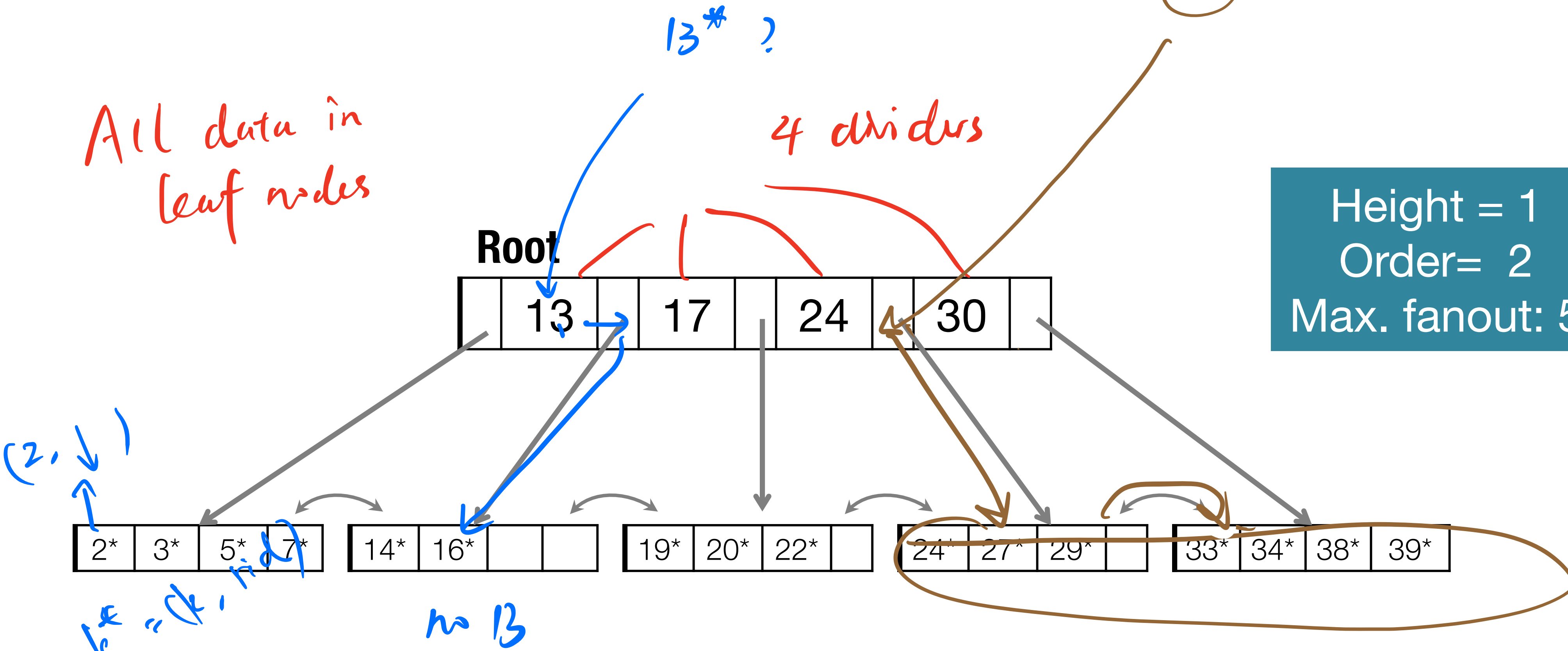


B+-Tree Page Format



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*



Height = 1
Order= 2
Max. fanout: 5

B+ Trees: Height

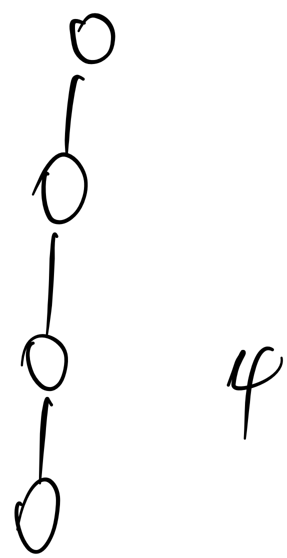
What is the height of a B+ tree? (formula)

Let:

MAX fan out = 2d+1

F: average fanout (average number of children for non-leaf node) (large #)

N: total number of leaf pages



Then,
height = $\lceil(\log_F(N)\rceil$

B+ Trees in Practice

- Typical order: 100. Typical fill-factor: 2/3
  - Maximum fanout: 201 $= 2d+1$
  - 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 KB
  - Level 2 = 133 pages = 1 MB
  - Level 3 ≈ 17.7K pages = 133 MB
  - Level 4 ≈ 2.35M pages ≈ 17.7 GB

$$\text{height} = \log_{(2d+1)\cdot f}\left(\frac{\#\text{ records}}{(2d \cdot f)}\right)$$

Arithmetic Example

- You are given a file of 10 million records
- Suppose you store an average of 10 data entries per leaf page

need 1 million leaf pages

151

- You build a B+ Tree with order 75, 67% average fill-factor ~100

- What is the avg fanout? ~100
- What is the height of your B+ Tree?

$\log_{100} 1\text{ million} = 3$

Question??

The average fanout is close to C

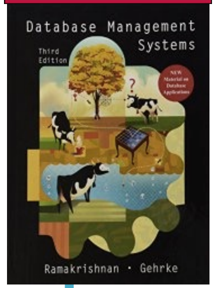
A. 67
B. 75
C. 100
D. 1000000

Question??

The height of the tree is B

A. 2
B. 3
C. 4
D. More than 4

A Note on Order



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

  (book uses this)

- *Order d* concept suggested by physical space criterion in practice (e.g. at least half-full)
  - Index (i.e. non-leaf) pages may hold many more entries than leaf pages, particularly for alternative 1.

B+ Tree Operations

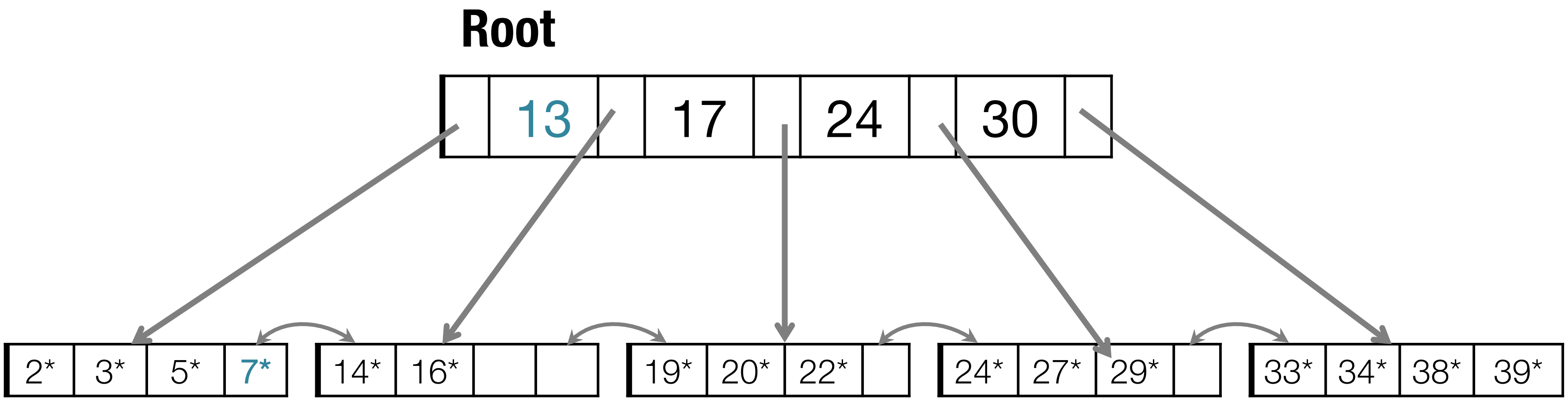
- Search
  - Equality
  - Range
- Insert data entry
- Delete data entry
- Bulk load

B+ Tree: Inserting a Data Entry

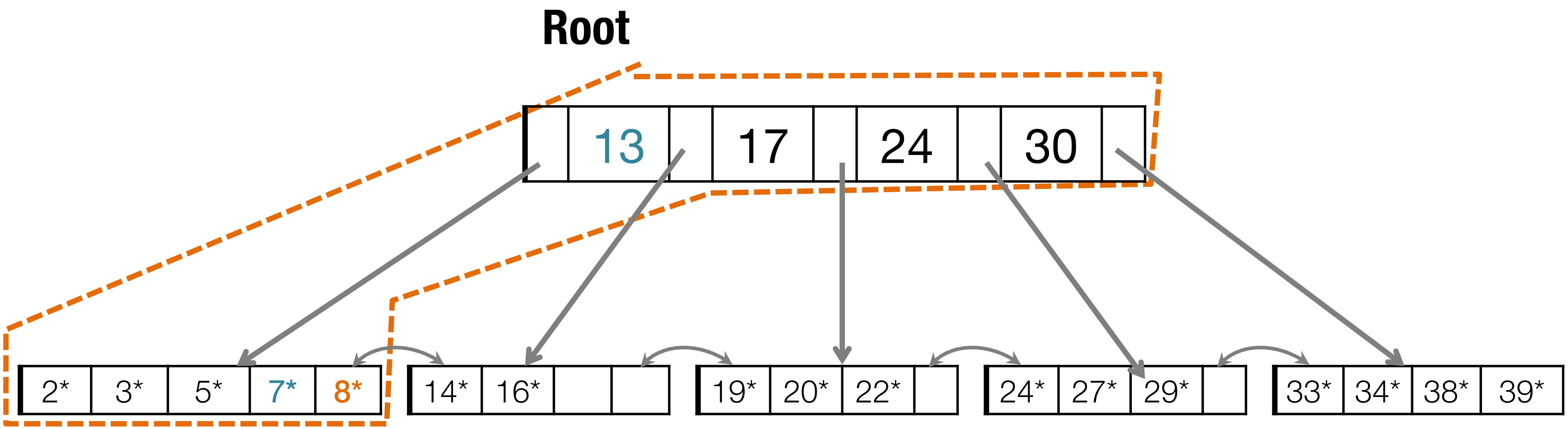
- Maintain invariants:
  - Search-tree property
  - All nodes must be at least ½ full (except root node), i.e., has between d and 2d entries
  - Root node is allowed to have a single entry

- 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.

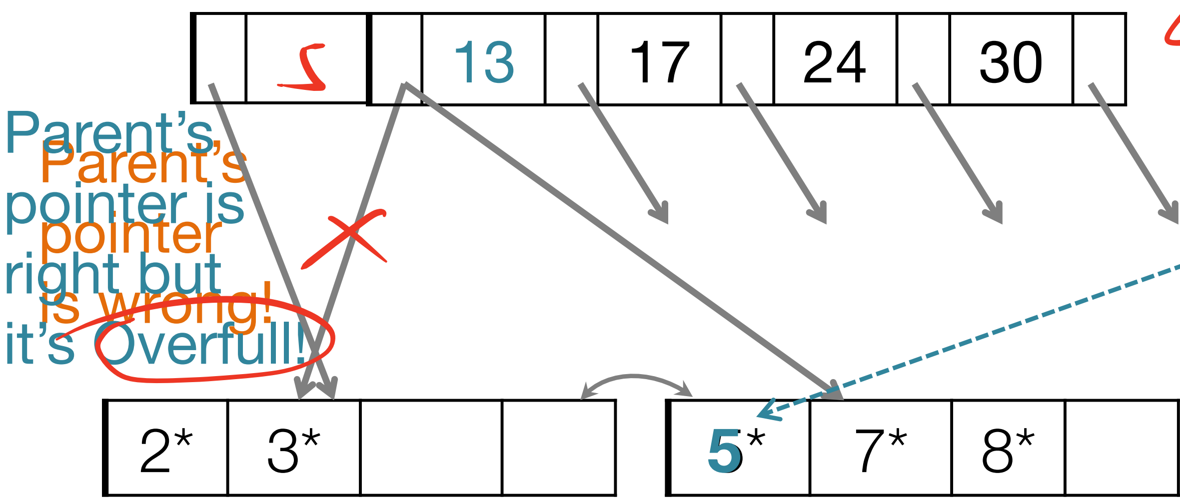
Inserting 8* into B+ Tree



Inserting 8* into B+ Tree



Overfull

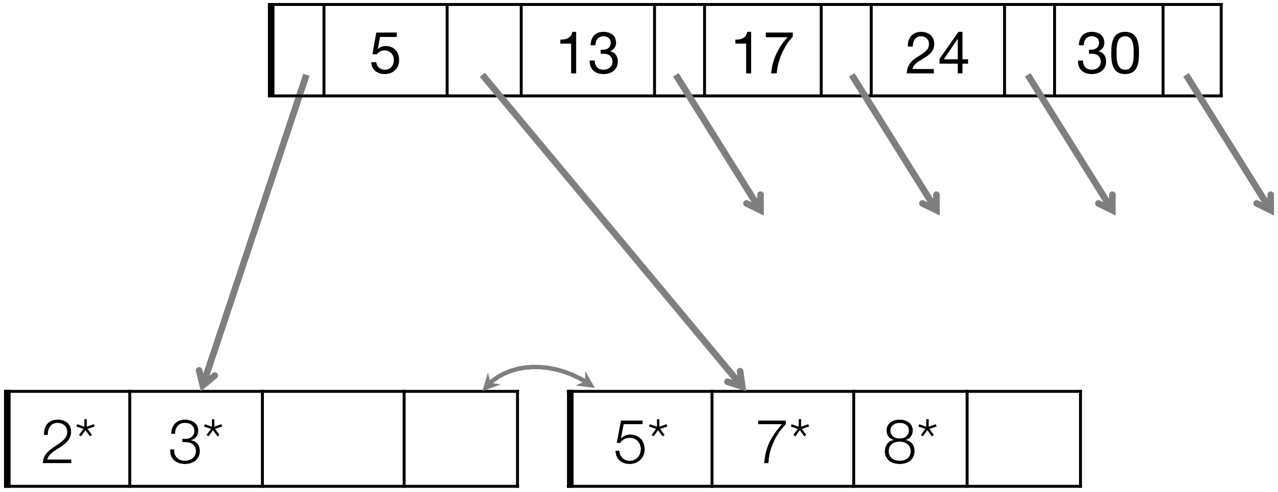


⇐ parent doesn't have enough room

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

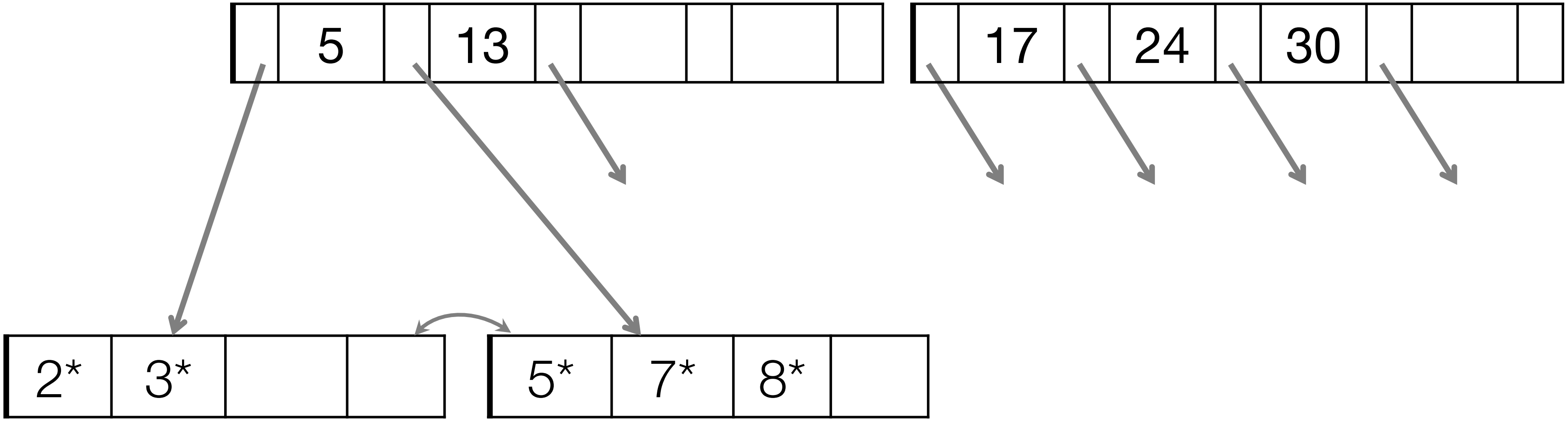
Splitting Overfull Non-Leaf Nodes

Overfull

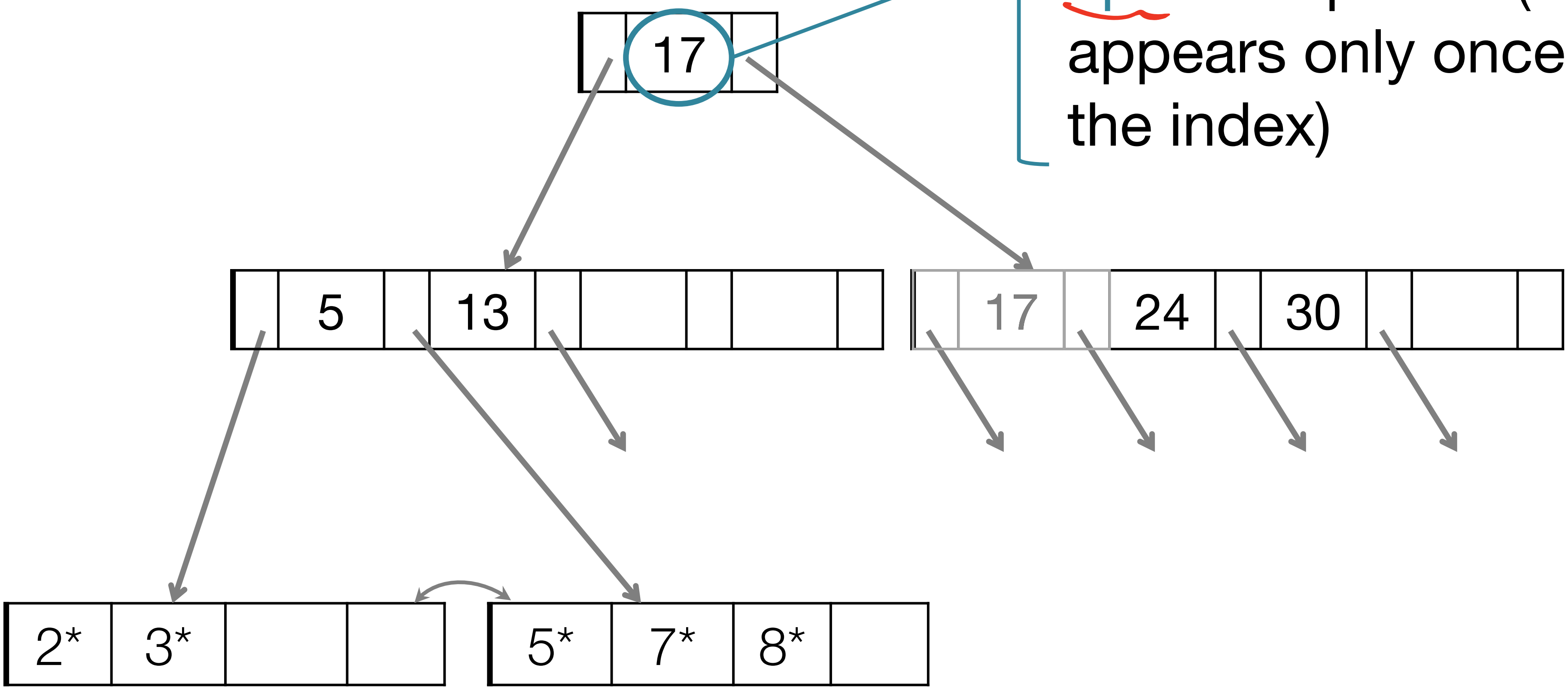


Splitting Overfull Non-Leaf Nodes

Split



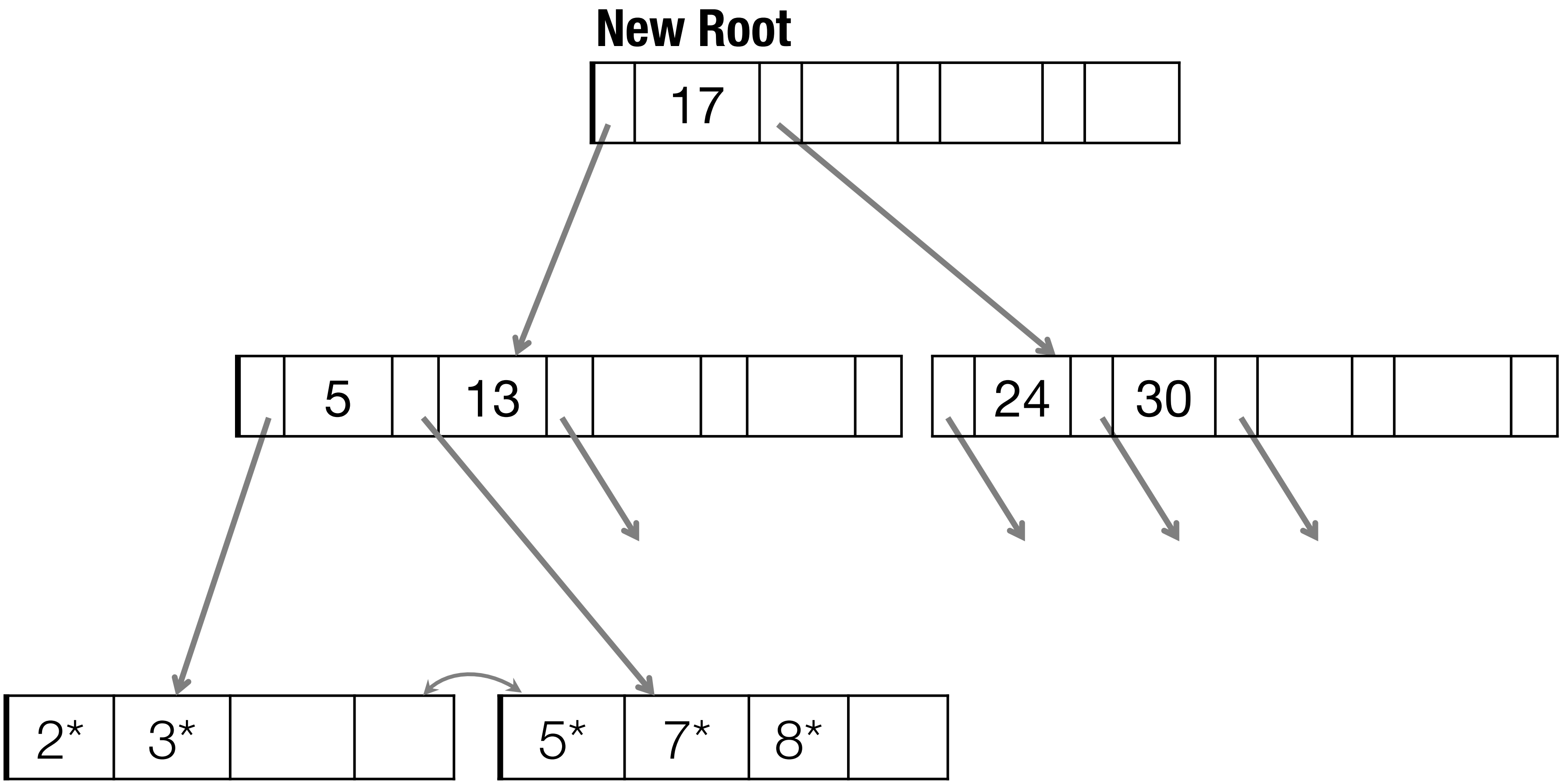
Splitting Overfull Non-Leaf Nodes



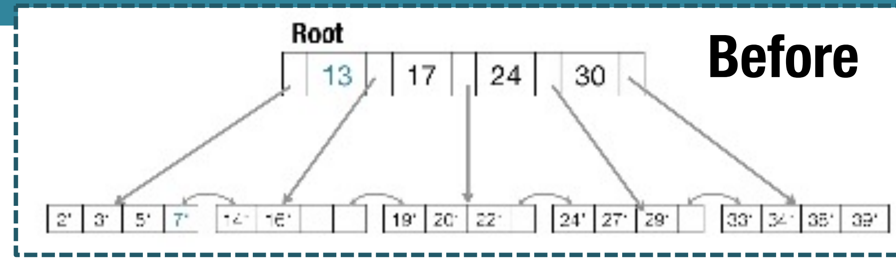
Non-leaf split: The middle key is pushed up to the parent (i.e. appears only once in the index)

Splitting Overfull Non-Leaf Nodes

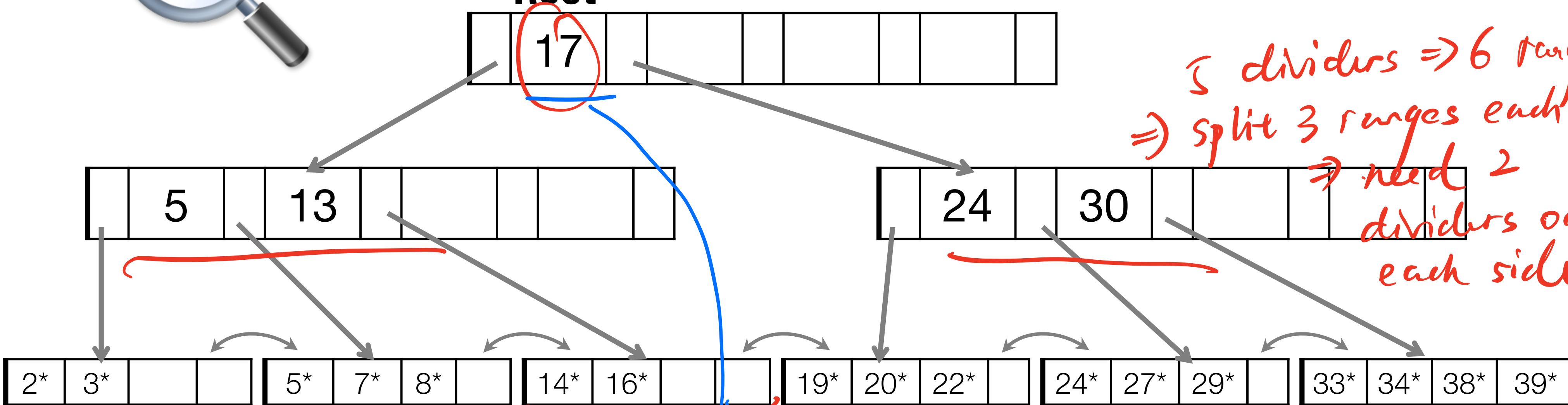
Minimum occupancy is guaranteed in both leaf and index page splits (but not in the root)



The B⁺-tree After Inserting 8*



new root created
height ↑



5 dividers => 6 ranges
=> split 3 ranges each
=> need 2 dividers on each side

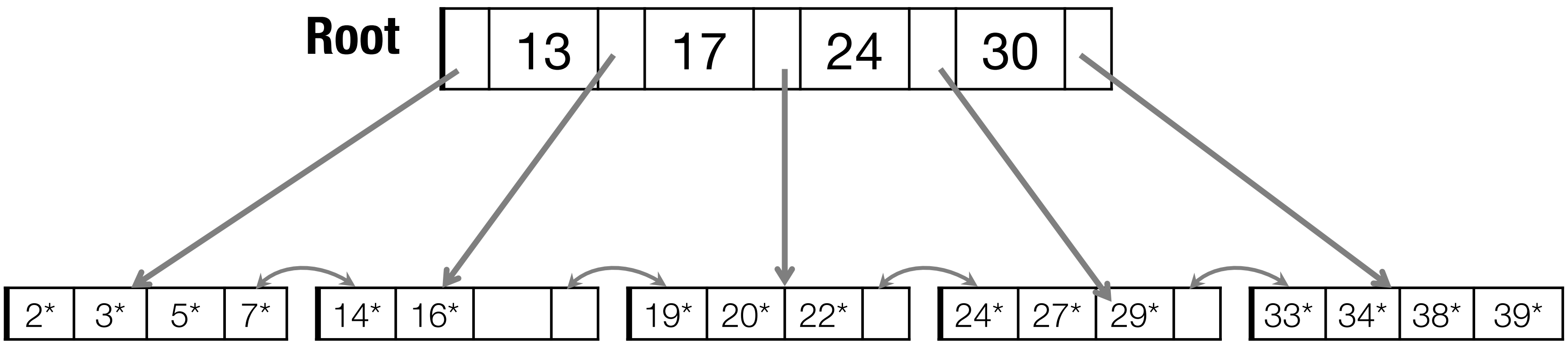
13 – 17

17 – 24

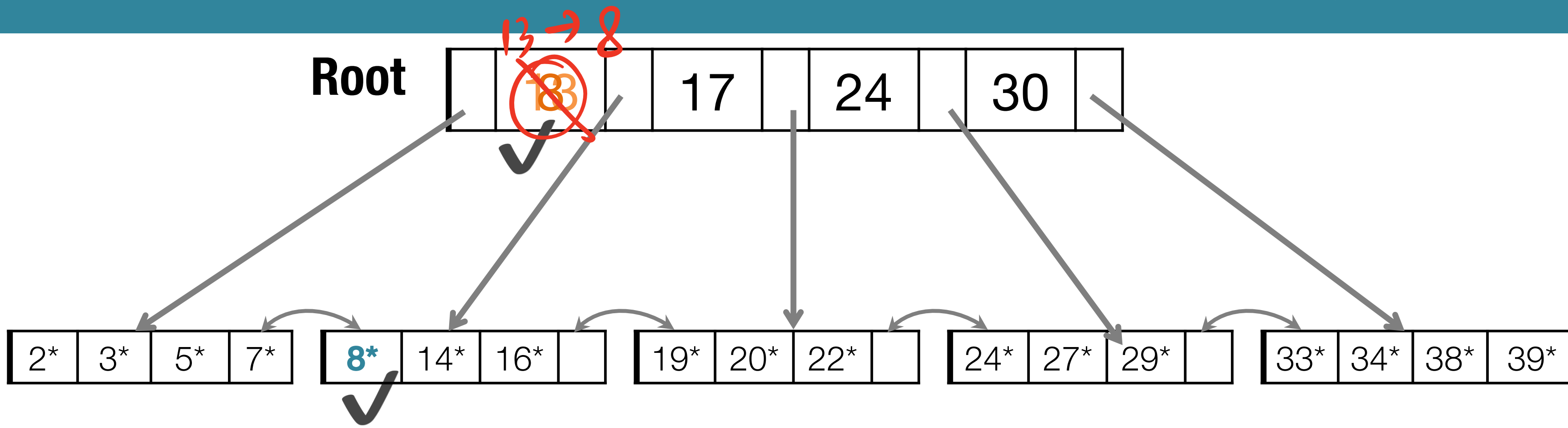
- Root was split: height increases by 1
- Could avoid split by re-distributing entries with a sibling
  - Sibling: immediately to left or right, and same parent

=> shift the boundary btw siblings

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
    - Can be OK to use splitting even though it propagates up the tree since that propagation would be rare and result in fewer splits on future inserts.
  - Use only for leaf level entries
    - Only have to set pointers

Question??

Select all that are true.

The height of a B-tree

A. Can be increased by splitting a leaf ⇒ not directly but can sing root to split

B. Can be increased by adding a new child to a leaf ✗

C. Can be increased be splitting the root ✓

D. Cannot be changed once the tree has been built ✗

B+ Tree Operations

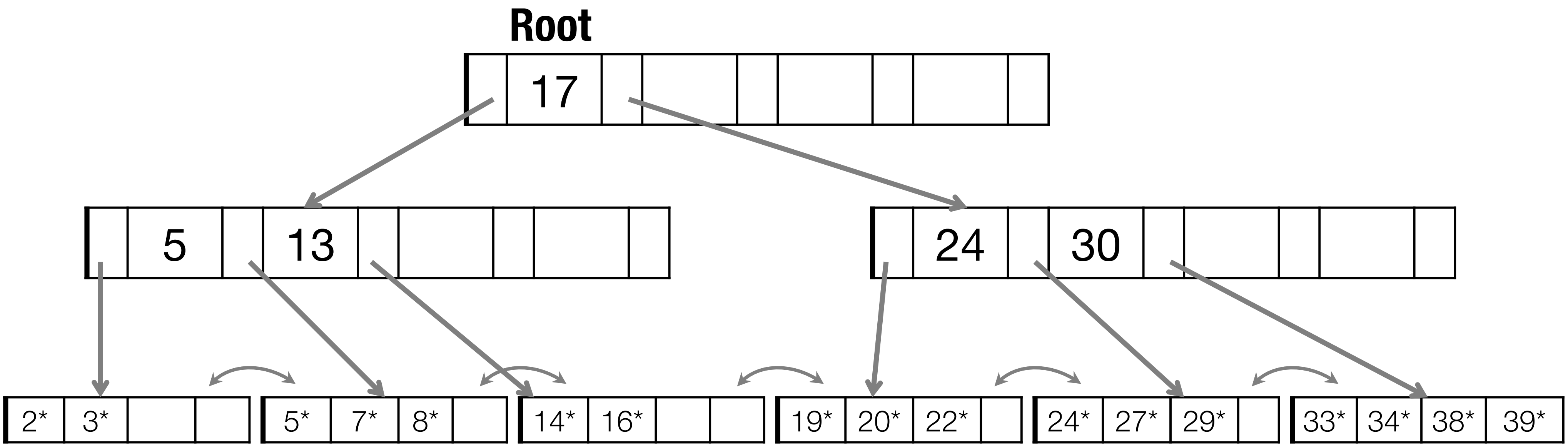
- Search
  - Equality
  - Range
- Insert data entry
- Delete data entry
- Bulk load

B+-Tree: Deleting a Data Entry

1. Find the data entry (will always be at a leaf)
2. Delete it
3. Restore the B+ tree invariant
   - If L has d or more entries, done!
   - If L has only d-1 entries,
     - 1 Try to re-distribute, borrowing from a sibling with more than d entries (sibling is an adjacent node with same parent as L)
     - 2 If re-distribution fails, merge *L* and sibling (Should always work!)
4. On merge, delete relevant entry in parent

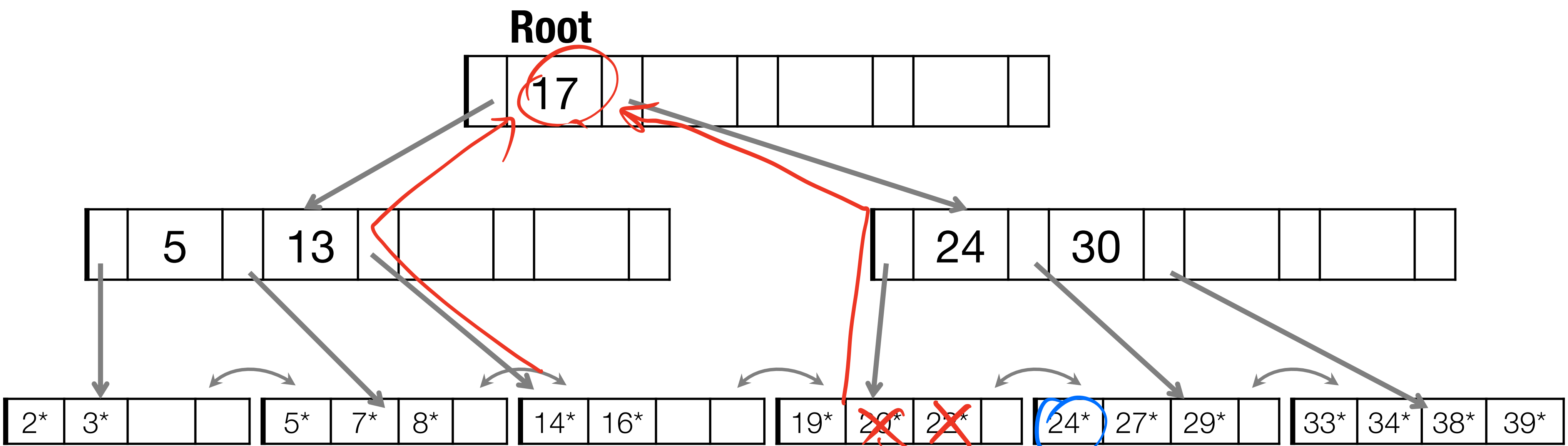
Merge could propagate to root, decreasing height.

Example Tree



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

Deleting 22* and 20*



Less than half!

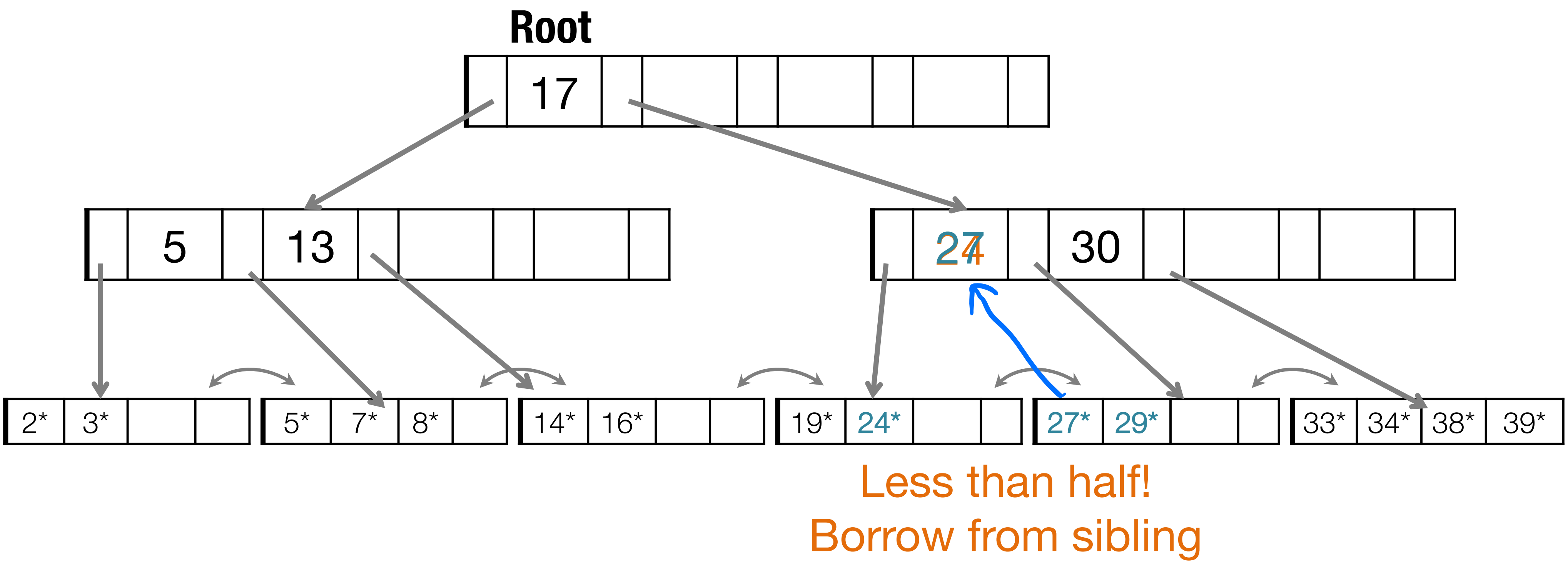
Borrow from sibling

if borrow from left
→ find common ascester

minimum 2 not satisfied

both side is OK
① borrow from wherever has more
② borrow from arbitrary side to minimize I/O

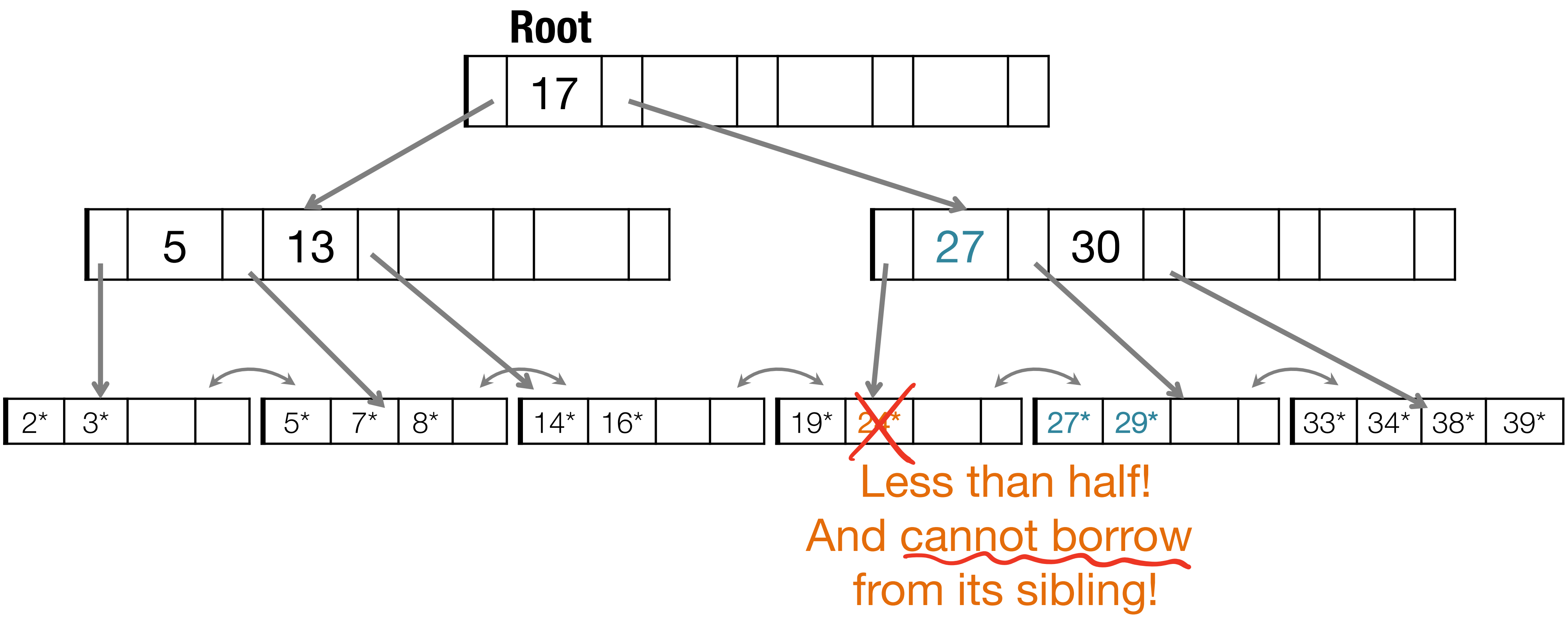
Deleting 22* and 20*



Deleting 20* is done with re-distribution.

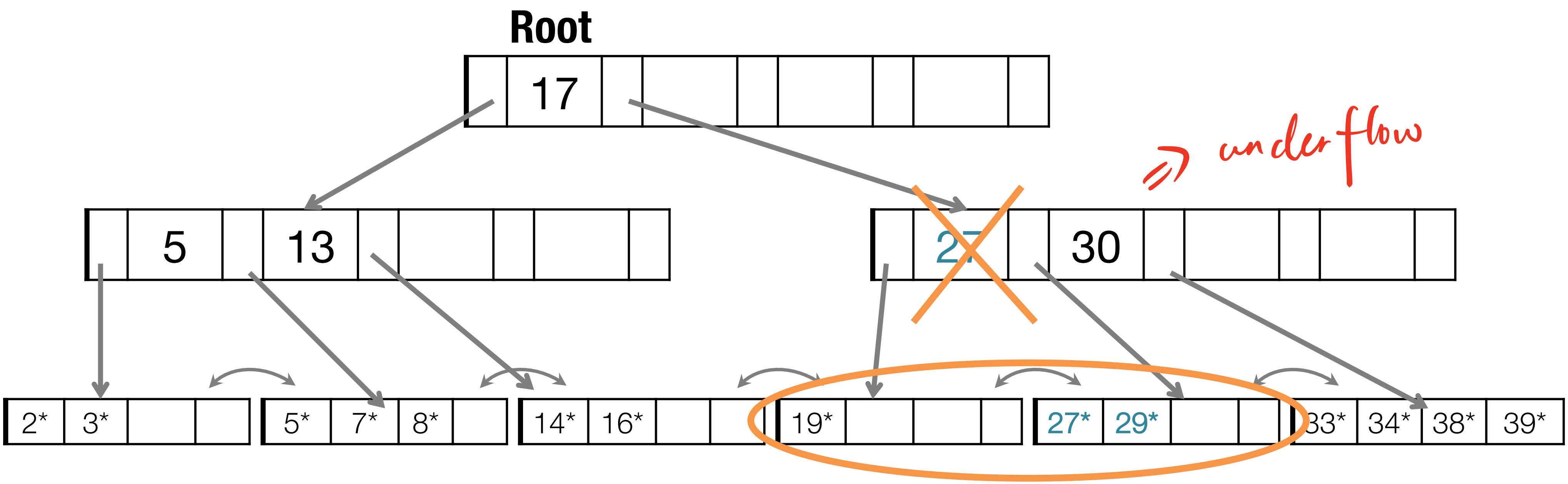
Notice how middle key is copied up.

… And then Deleting 24*



- Must merge ⇒ with sibling that you share parent with
- In the non-leaf node,
  toss the index entry with key value = 27

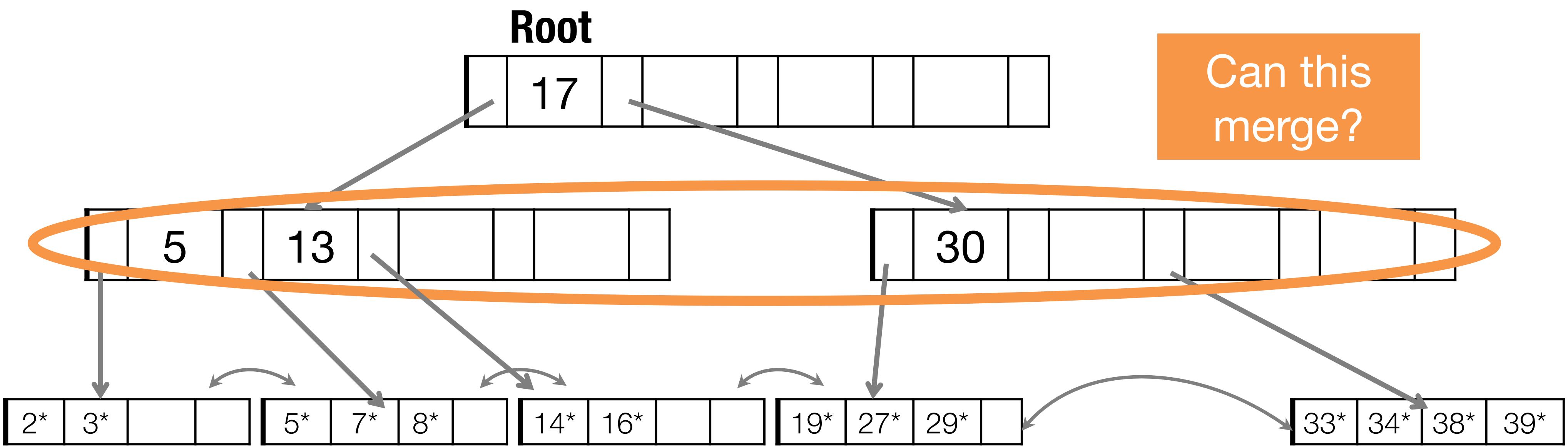
... And then Deleting 24*



two ranges into one
⇒ range divider btw
has to go away

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

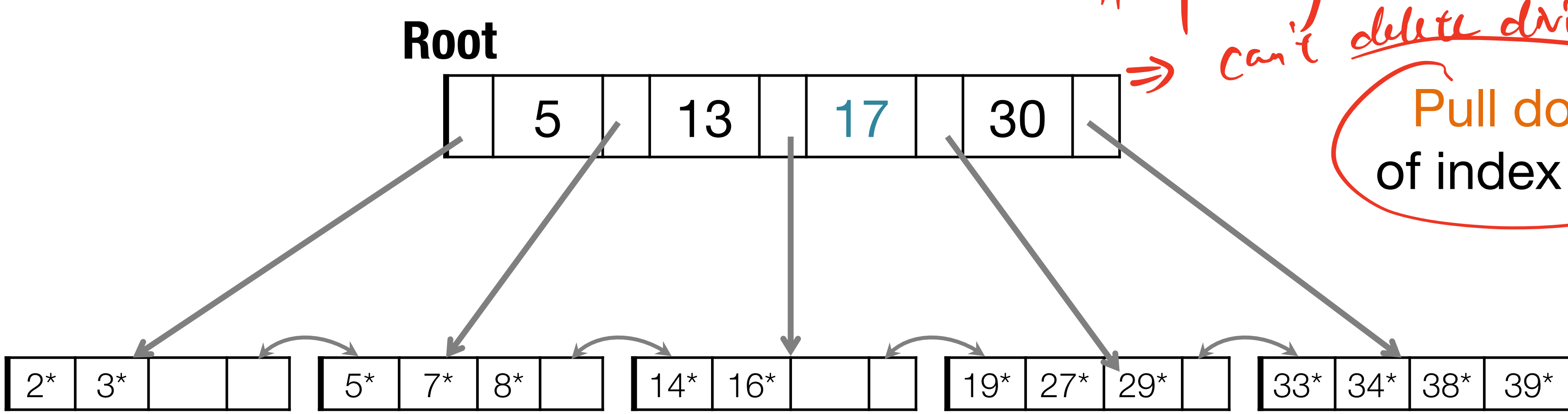
… And then Deleting 24*



Can this merge?

# of ranges doesn't change

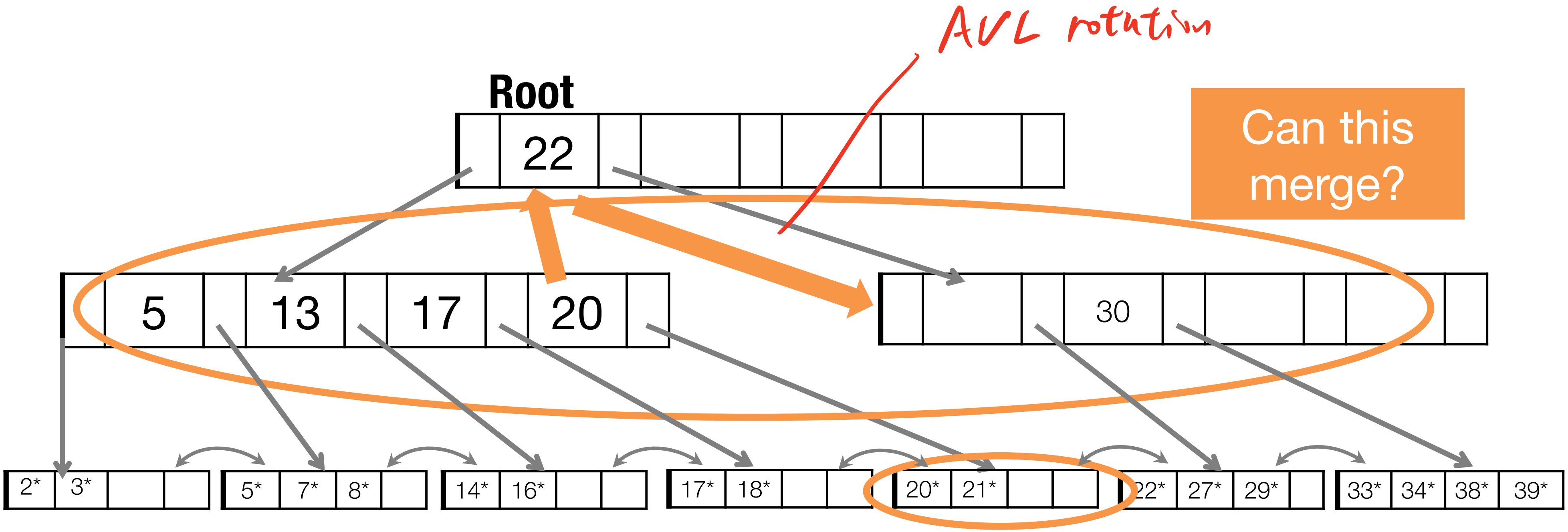
⇒ can't delete divider



Pull down of index entry

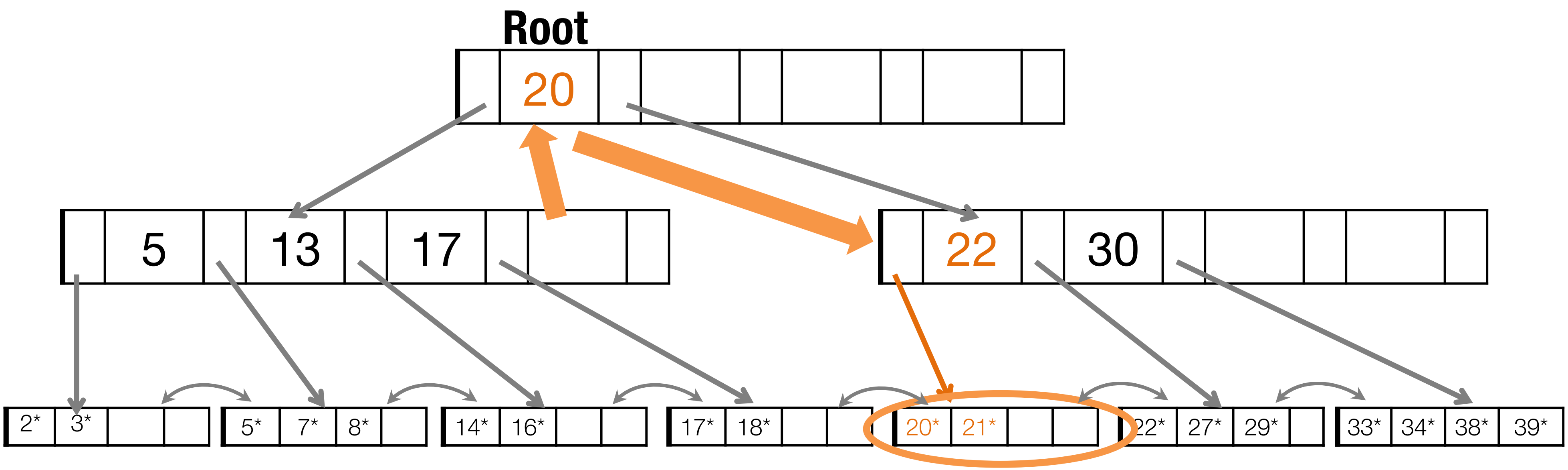
Another Deletion Example: Non-leaf Re-distribution

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



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

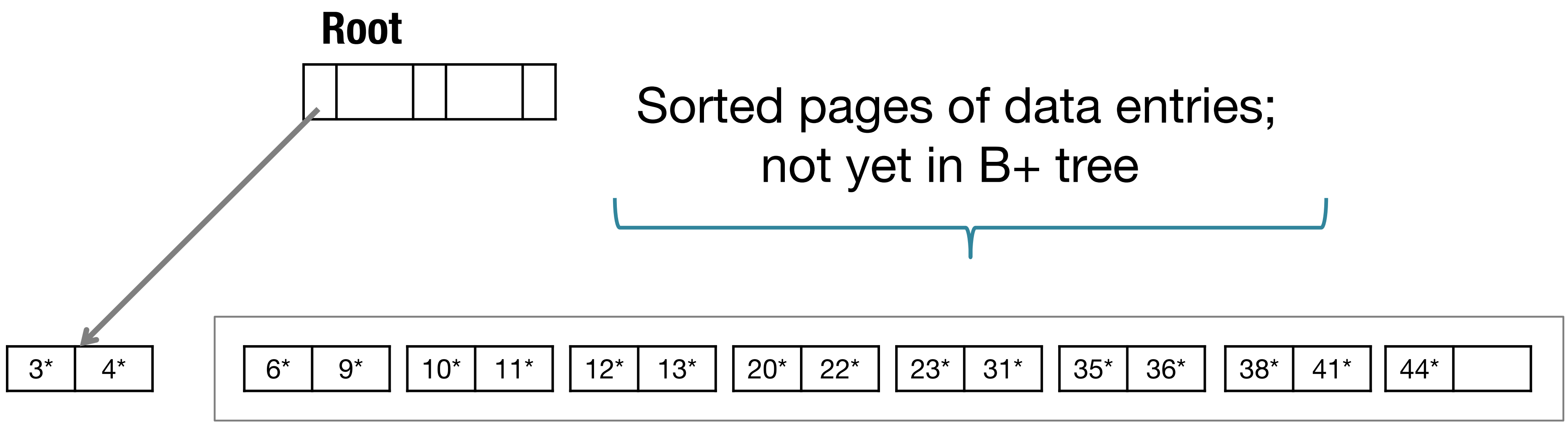
- Search
  - Equality
  - Range
- Insert data entry
- Delete data entry
- Bulk load

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 than one by one insert

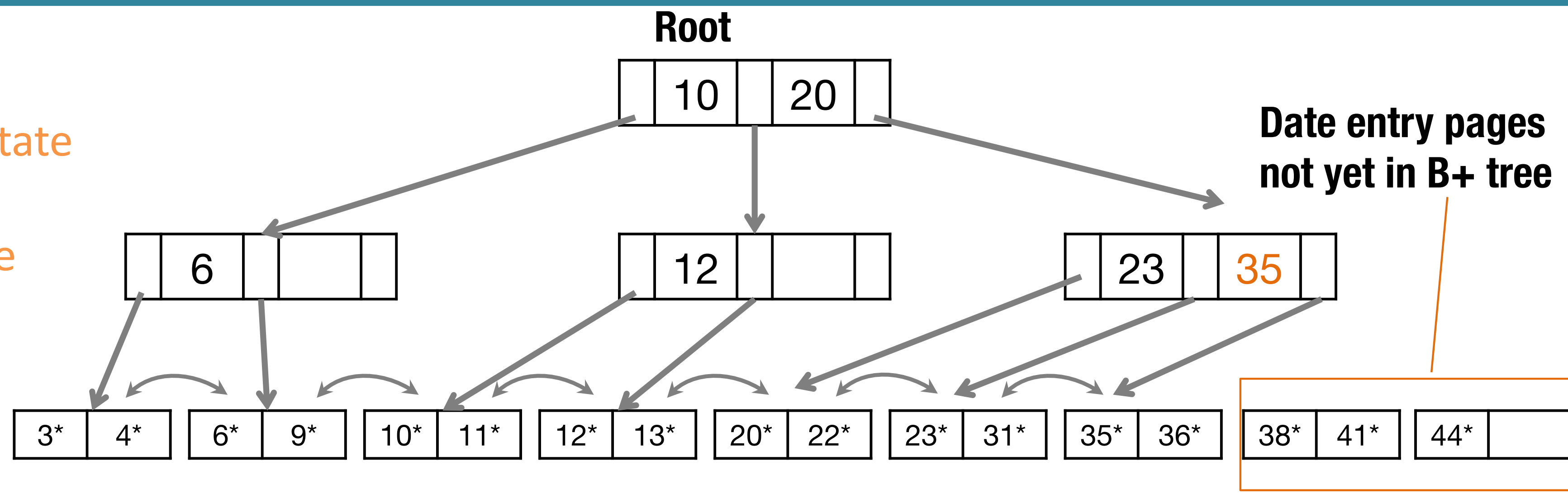
Bulk Loading of a B+ Tree

Initialization: Sort all data entries, insert pointer to first (leaf) page in a new (root) page. Then add one leaf page at a time, rather than one record at a time



Bulk Loading (Contd.)

Intermediate state before adding [38*, 41*] page

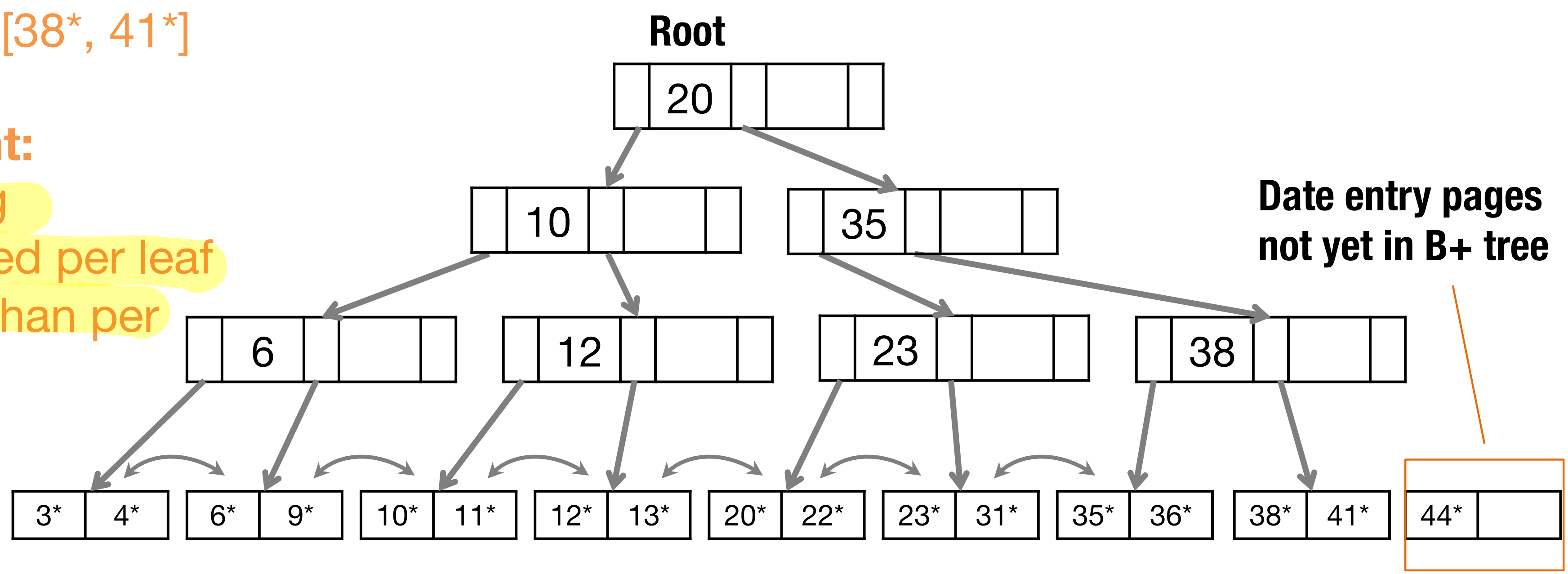


After adding [38*, 41*] to the index

Why efficient:

No searching

Index modified per leaf page rather than per entry



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. Max fanout: 2d+1.
  - 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.

keep top 2 in memory

=> 1/2 depth

Suggested Exercises + Readings

Suggested Exercises: 10.1, 10.5, 10.7