CSE 350: Advanced Data Structures and Indexes (Spring 2026)

Lecture 4: B+-tree Index Overview

2/5/2026

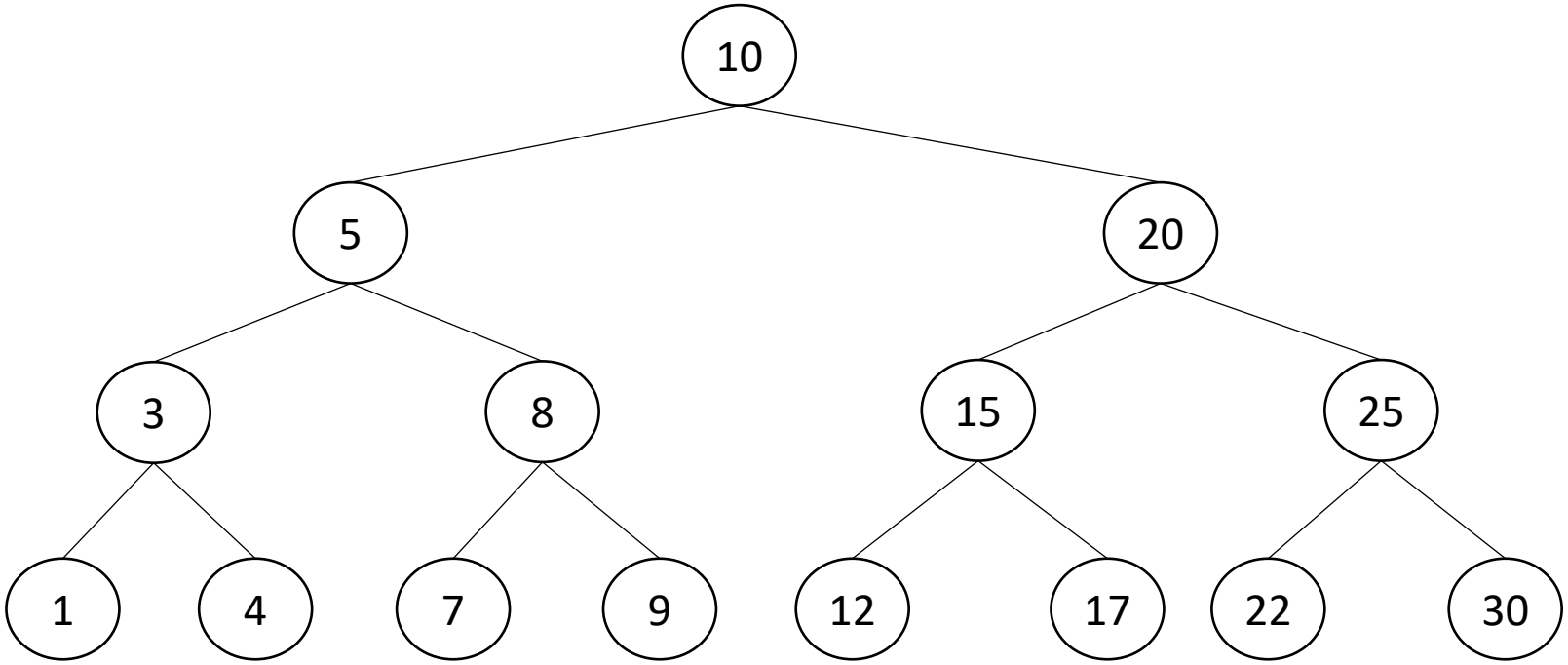
University at Buffalo
Department of Computer Science and Engineering
School of Engineering and Applied Sciences

Key-Value Store Interface

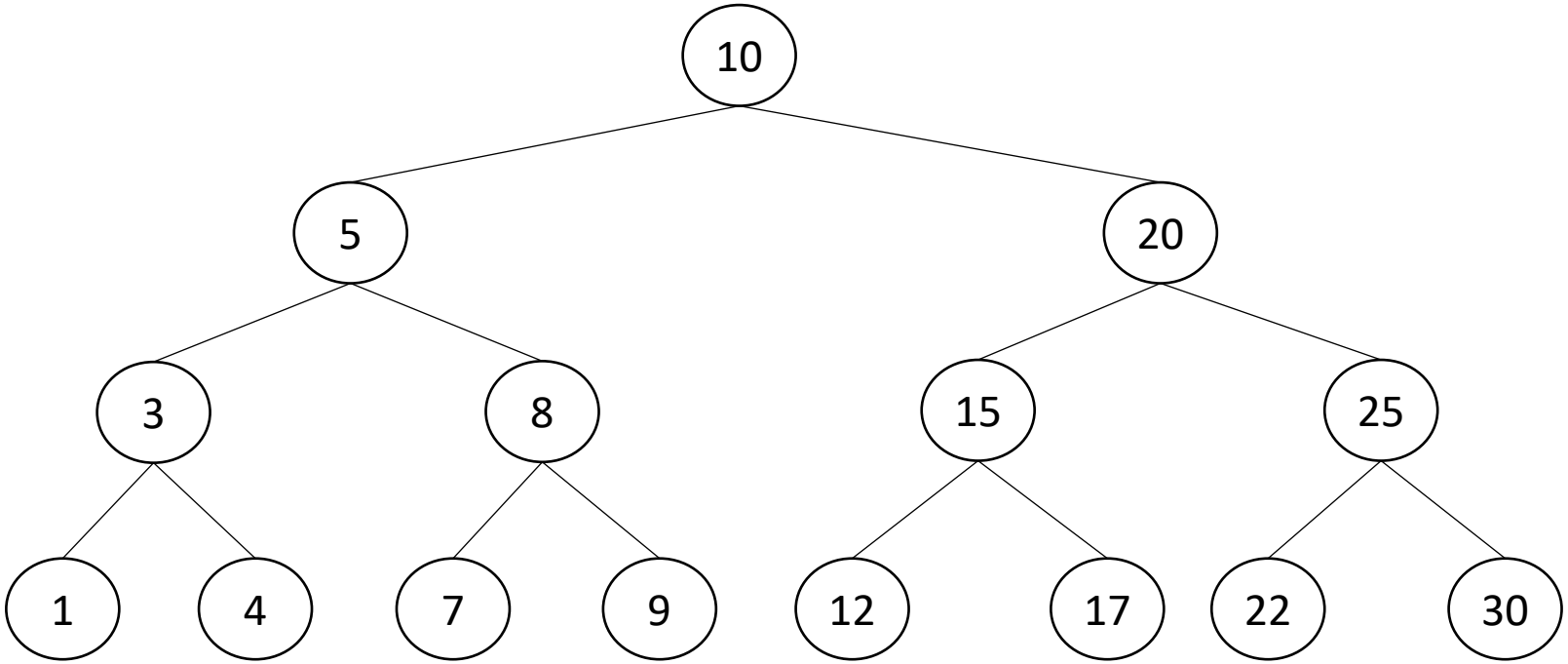
- Associative map from `key` to `value`
  - For simplicity, let's assume
    - `key and value: uint64_t`
- Simple in-memory versions in common languages (no duplicate key):
  - `std::map<uint64_t, uint64_t>`
  - `java.util.TreeMap<uint64_t, uint64_t>`
- Supports these operations in $O(\log N)$ time
  - `find(key), insert(key), delete(key)`

- *Under the hood: (balanced) binary search trees*

How does a binary search tree work?

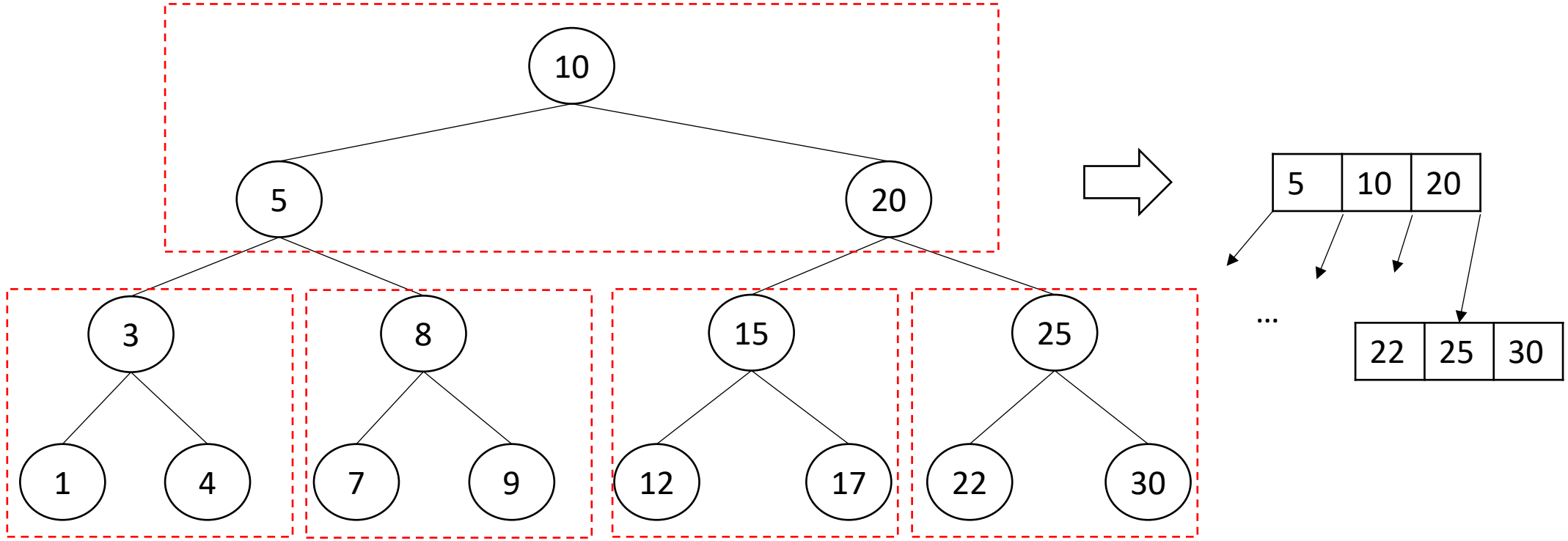


What if we want to store it on disk?



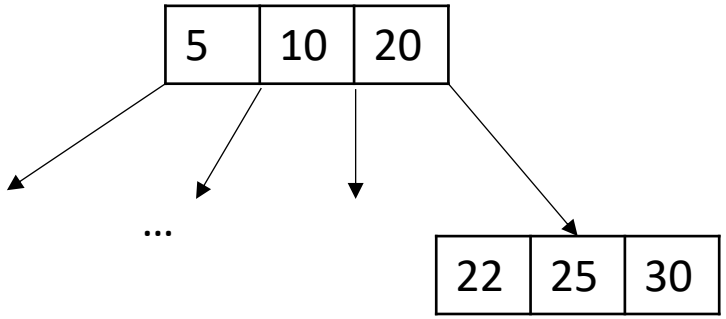
- Naïve approach:
  - treat a file as memory space and implement a memory allocator for it (e.g., mmap)
  - pointers becomes indirections
  - Lookup cost?

Reducing I/O via locality: 2-3-4 tree



- Pack up 2 - 4 branches & 1 - 3 separator keys together in a node
  - Reduces I/O by up to 3/4

Insertion & deletion in 2-3-4 tree

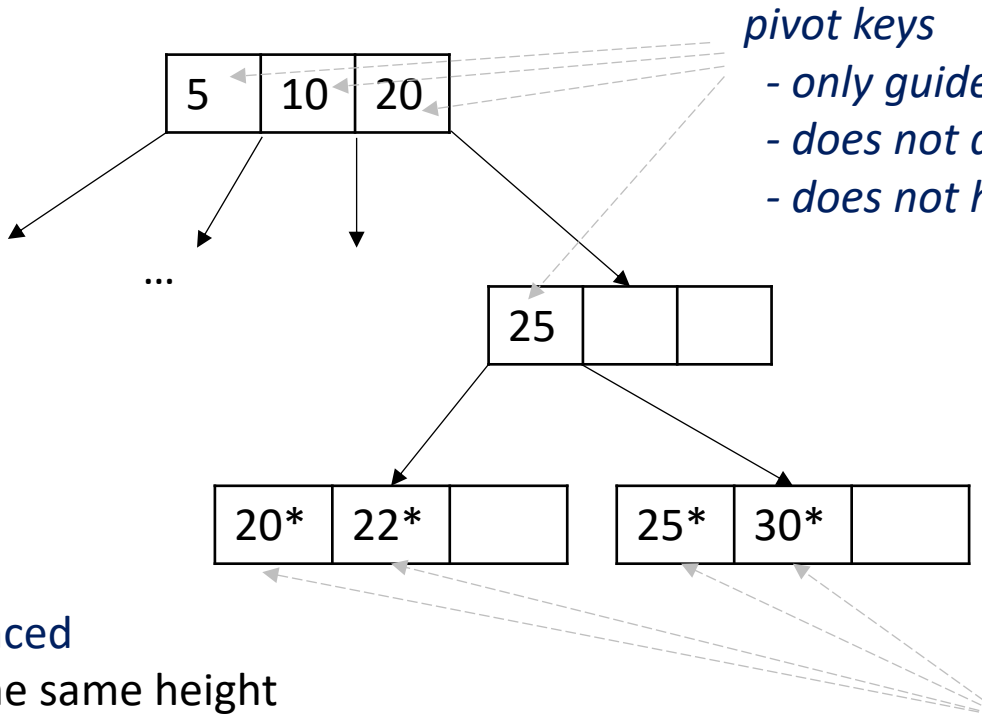


Problem? have to keep it balanced

- all leaves must be at the same height
- no 1-node

- Complex rotation, merging, split logics
  - Still $O(\log N)$ operations, but have large constants

Simplify insertion & deletion



pivot keys
- *only guides search*
- *does not associate with values*
- *does not have to correspond to keys in the map*

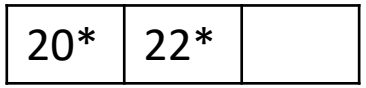
actual keys (and values follow)

Problem? have to keep it balanced
- all leaves must be at the same height
- no 1-node

- Complex rotation, merging, split logics
  - Still $O(\log N)$ operations, but have large constants

- *Instead of doing so, let's move all actual keys down into the leaves*
  - *internal nodes only contain "pivots"*

Increasing “fanout”



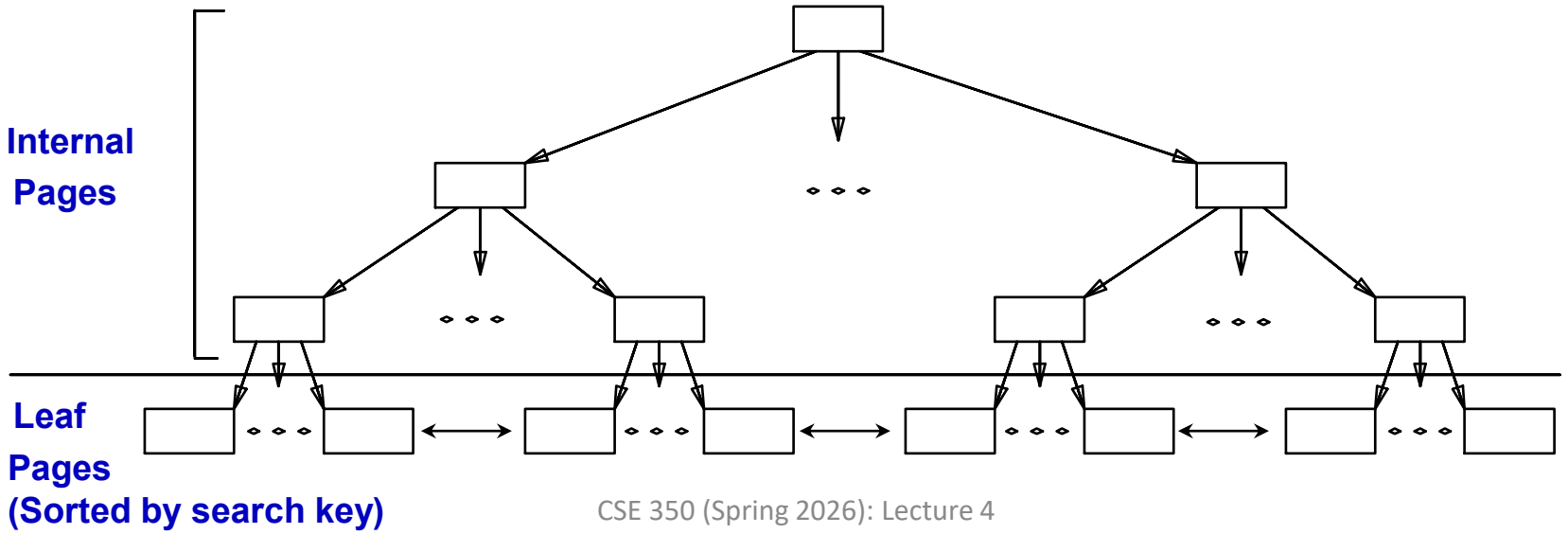
- How large is a leaf/internal 2-node? 3-node? 4-node?
  - No more than tens of bytes
  - Page size: 4096 bytes
  - Still wastes I/O

- How about making a B-node?
  - With up to B branches in internal nodes, or B key-value pairs in leaf nodes
  - To match page size
  - Reduces tree height => saves I/O

This is called a B+-tree

B+-Tree: the most widely used index

- Dynamic structure
  - Adapts to insertion/deletion
  - Data entries are stored in the leaf pages; Index entries in internal pages
  - Balanced: all paths from root to leaf page has the same length -- called tree height h
    - There's a min occupancy for each page except for root (usually 50%)
- Each node in the tree is a page in the file
  - B+-Tree internal/leaf node ≡ B+-Tree internal/leaf page
- We use the term B-tree interchangeably with B+-tree
  - *Though B-tree does exist -- stores key-value pairs internally as in 2-3-4 trees*



Data Entry in Leaf nodes

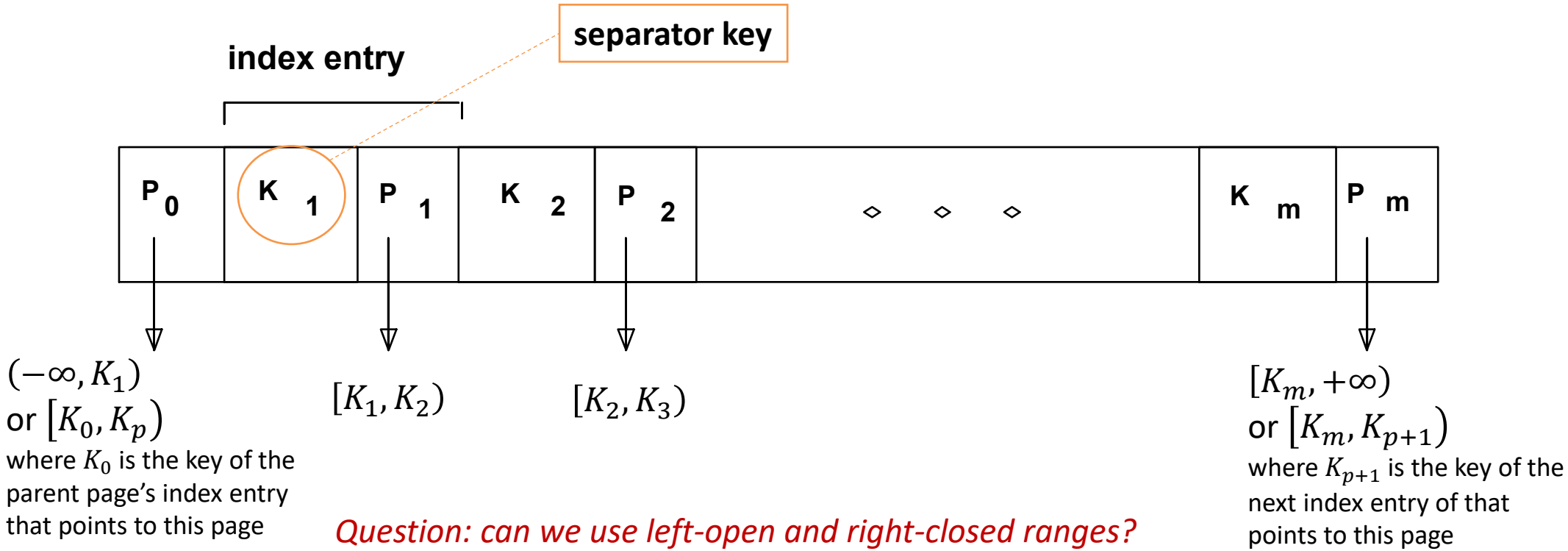
- <key, value> pairs
- logical, does not have to match physical layout

- we'll revisit this for supporting duplicates

Index Entries in Internal Nodes

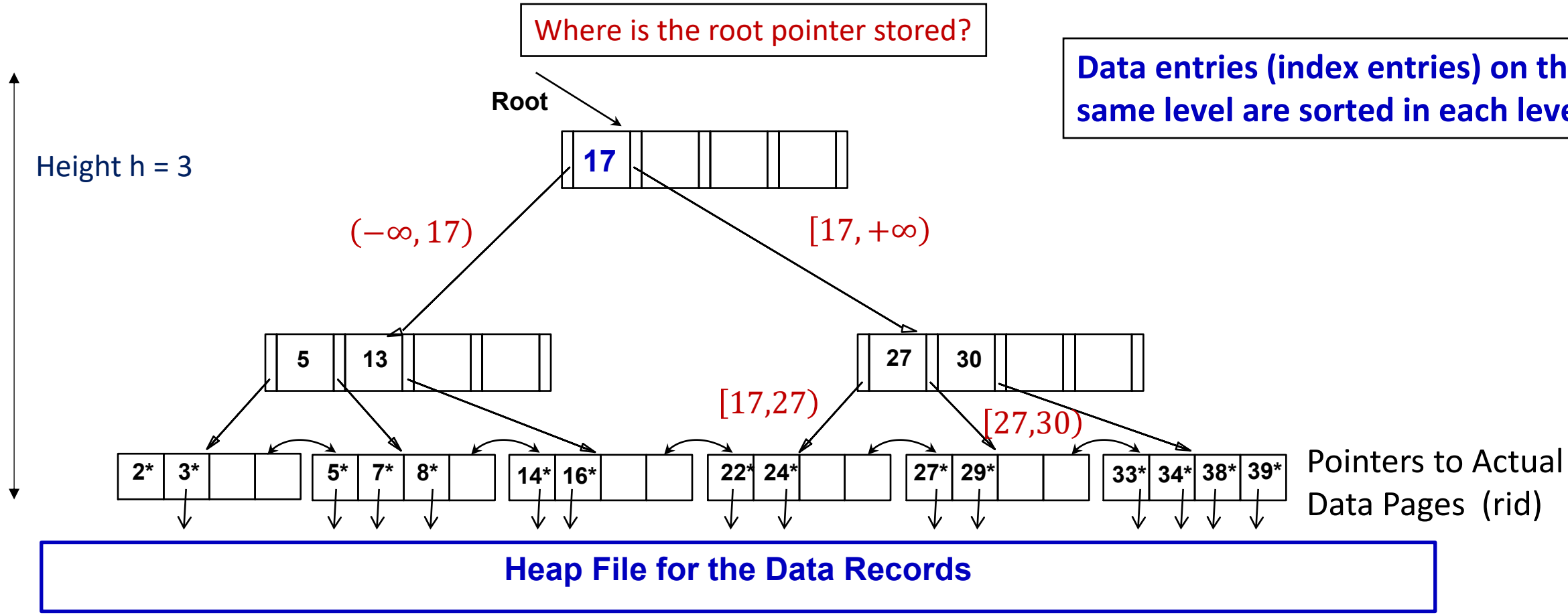
An index entry has the following format: (search key value, page id). The following shows an index page with m index entries (pay attention to the special "left-most pointer")

Note: entry 0 does not have a key; the range is implicitly defined by left child and K1



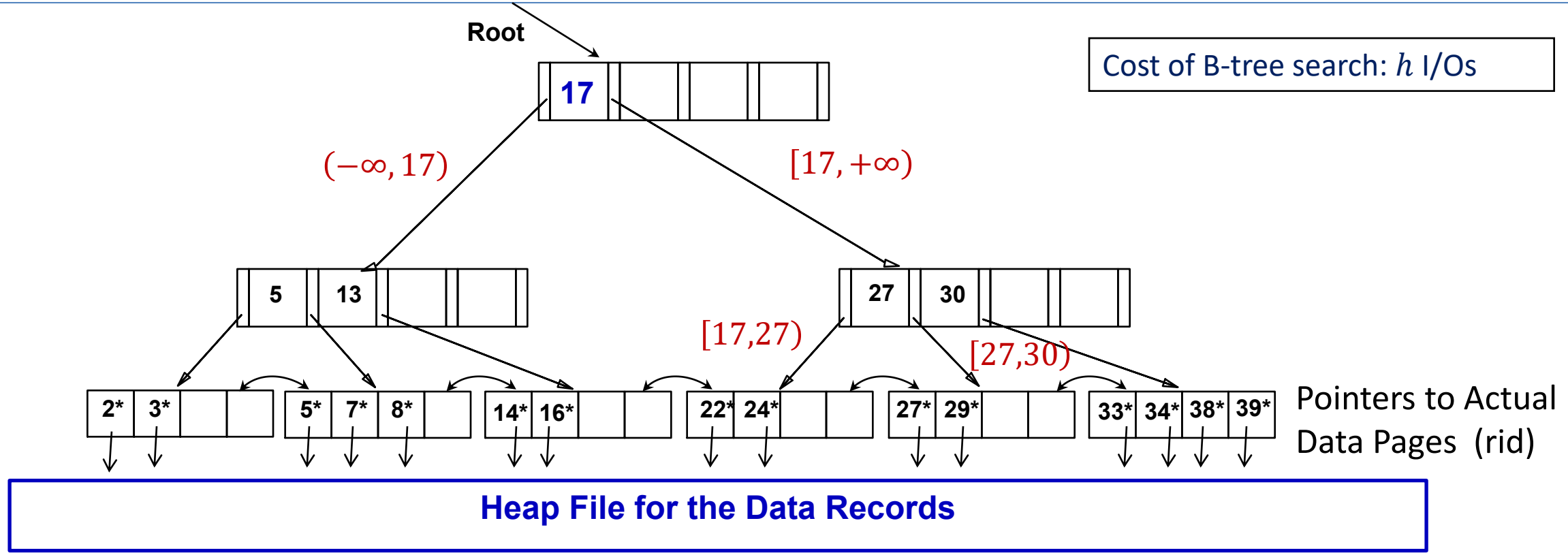
Question: can we use left-open and right-closed ranges?

B-Tree example



Let's assume unique and fixed-length keys for now. Leaf node capacity: $B = 4$. Fan-out $F = 5$.

B-Tree search



Cost of B-tree search: h I/Os

Pointers to Actual Data Pages (rid)

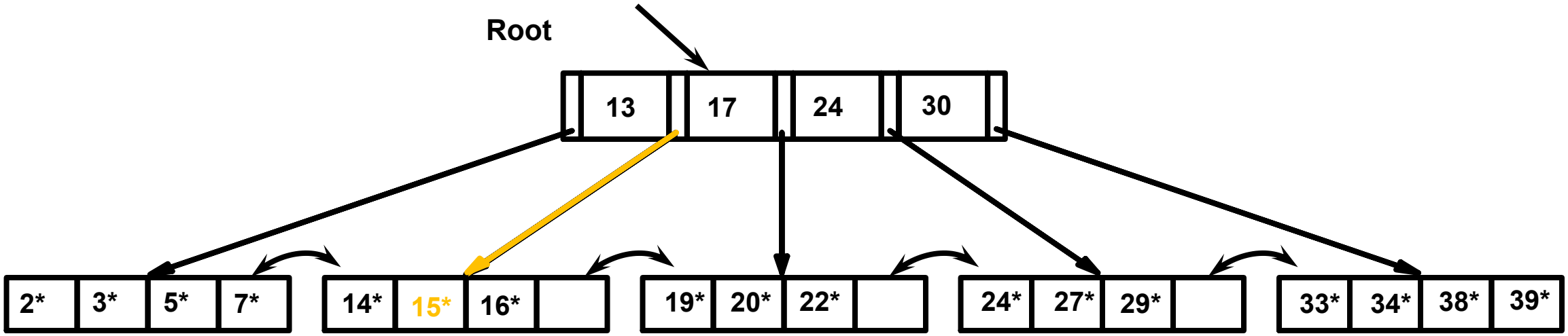
- Find 28*? 29*? All > 15* and < 30*
  - Starting from root and use key comparison to follow the correct pointers until reaching leaf.
  - To scan a range
    - Locate the lower bound of the key range
    - move right on the data entries until there're no left or you find one that's out of range
  - Can we locate the upper bound and move left instead?

B-Tree insertion

- Find correct leaf *L.*
  - Which one? see next slide
- 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* with the middle key.
- 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.*

B-Tree insertion example -- inserting 15*

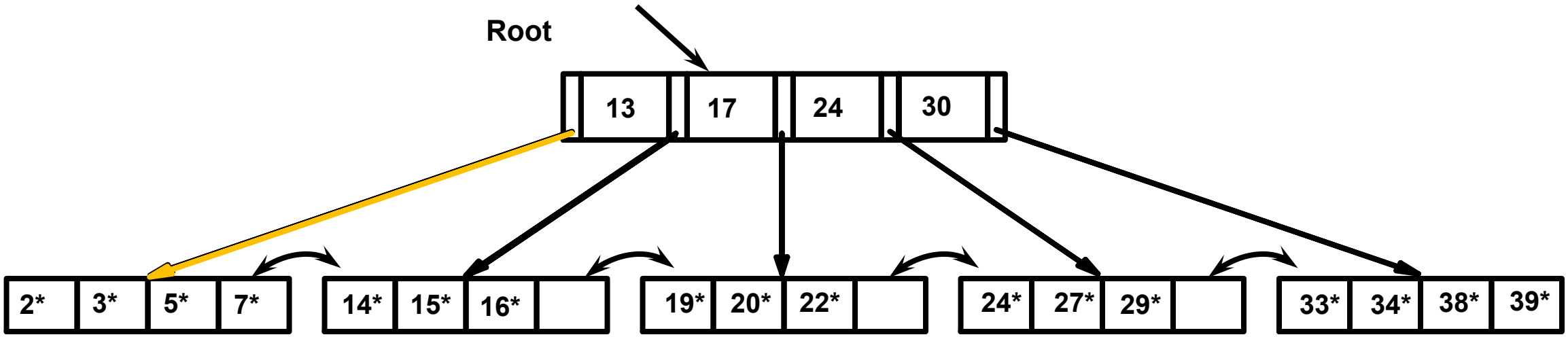
- Inserting 15*



Find the subtree where you would do search for the insertion key.

B-Tree insertion example -- inserting 8*

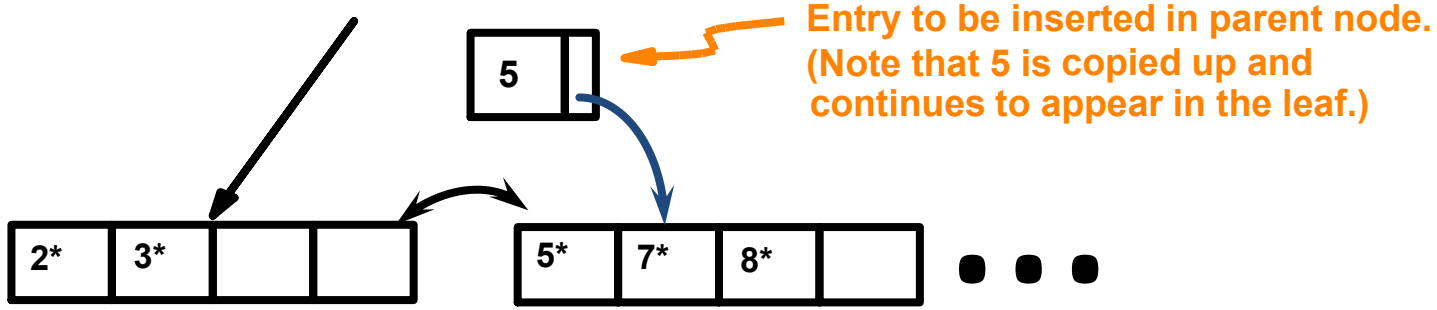
- Inserting 8*

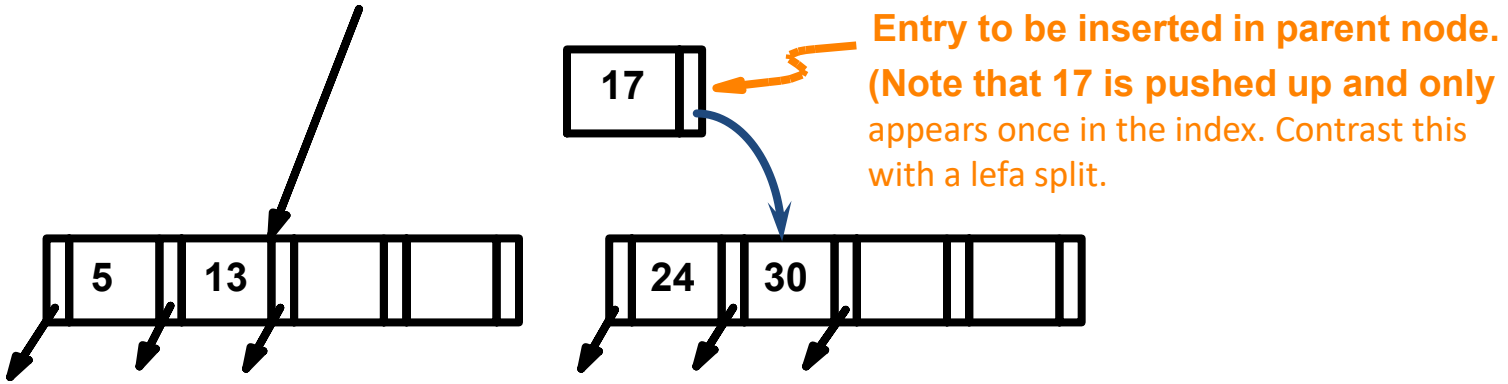


- Leaf page is full, what now? Split the page!
  - After that, the root page also needs to be split because there's no room for a new index entry

B-Tree insertion example -- inserting 8*

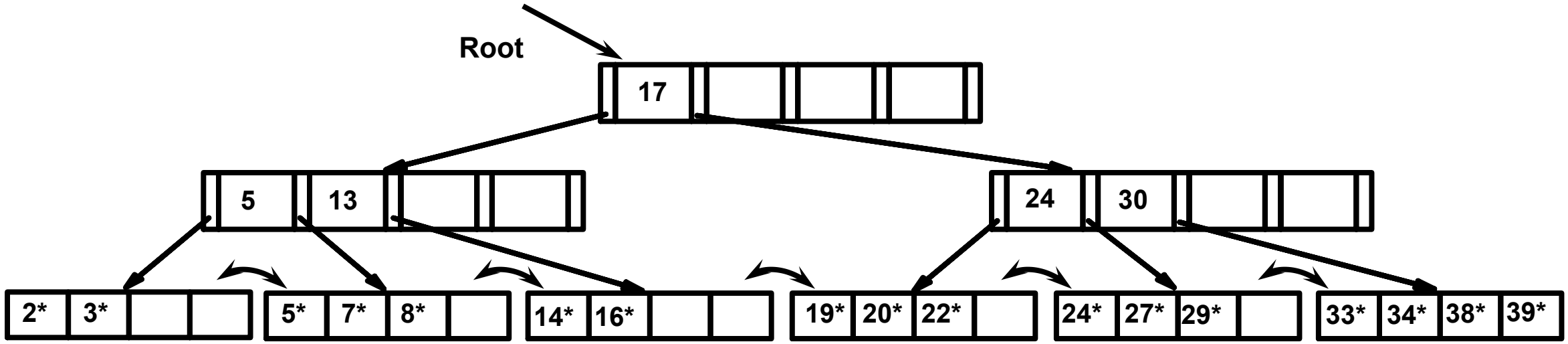
- Observe how minimum occupancy is guaranteed in both leaf and index page splits.
- Note difference between copy-up and push-up; be sure you understand the reasons for this.





B-Tree insertion example -- Inserting 8*

Cost of B-Tree insertion: $h + 1$ to $4h + 2 = O(h)$ I/Os



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.

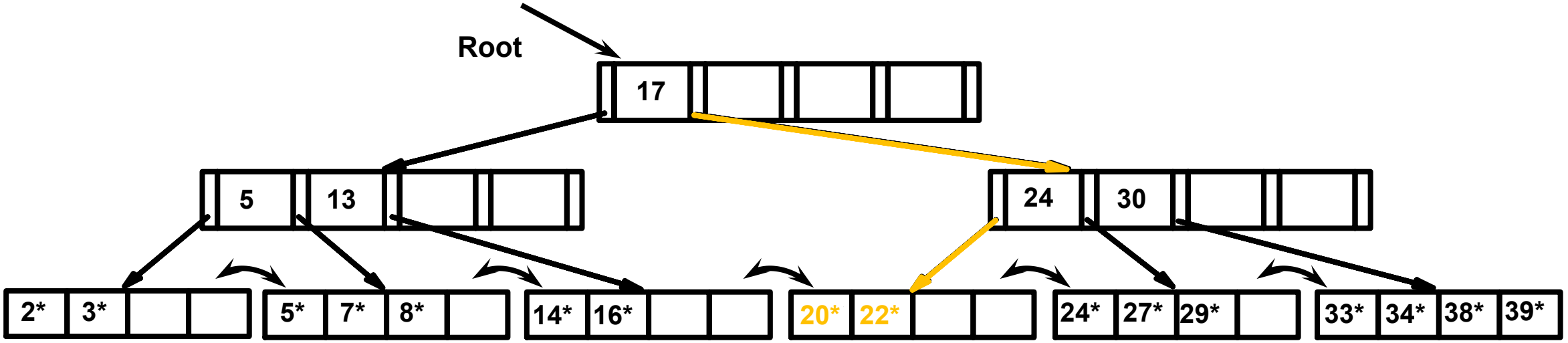
B-Tree deletion

- Start at root, find leaf *L* where entry belongs.
- Remove the entry.
  - If L is at least half-full, *done!*
  - If L has less than half full,
    - Try to *merge* *L* and a sibling sharing a common parent.
      - Pull down the key in the parent if this is an internal page
    - Or *redistribute* keys (i.e., rebalance) between L and a sibling sharing a common parent
      - Need to update the key in the parent after rebalancing
      - Rebalancing is rarely implemented in practice, why?
- If merge occurred, must delete an index entry from parent of *L*. Which one?
  - The one on the right.
- If redistribute occurs, must update the index entry from parent of L. Which one?
  - Still the one on the right.
- Merge could propagate to root, decreasing height.

B-Tree deletion example -- deleting 19*

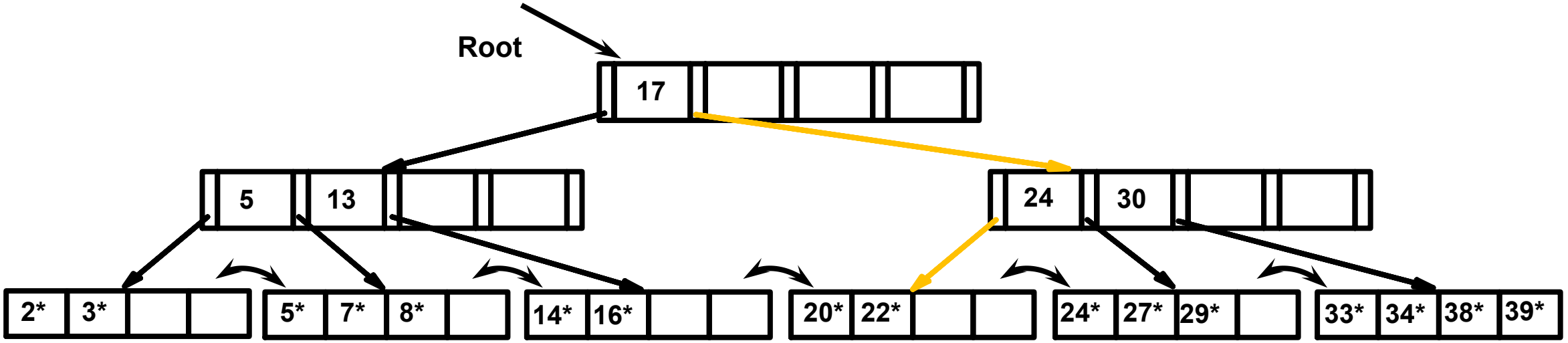
- Deleting 19* is easy.

Cost = $h + 1$ I/Os.



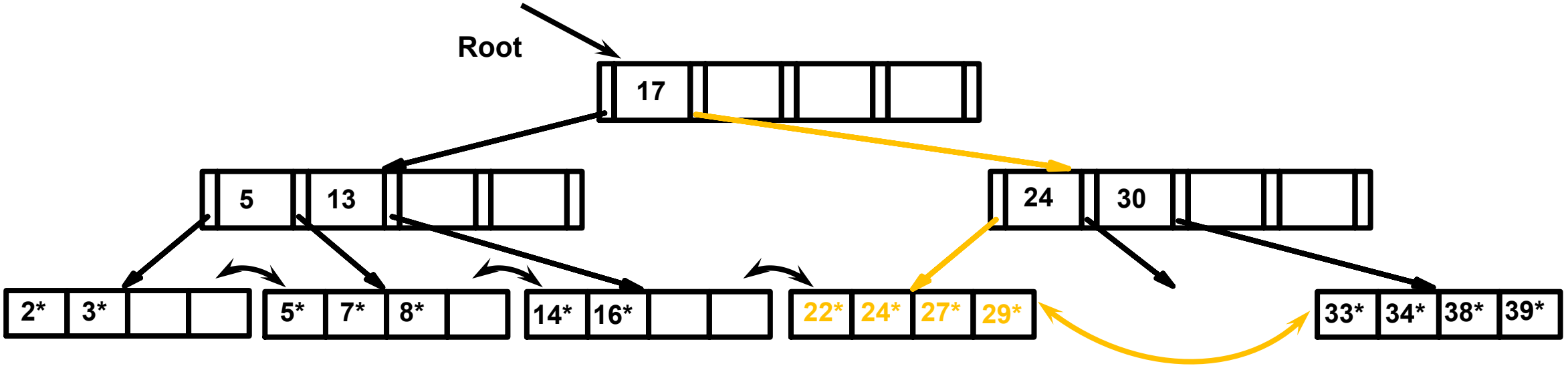
B-Tree deletion example -- deleting 20* with merging

- Deleting 20* with merging. Index entry pointing the right sibling is deleted.



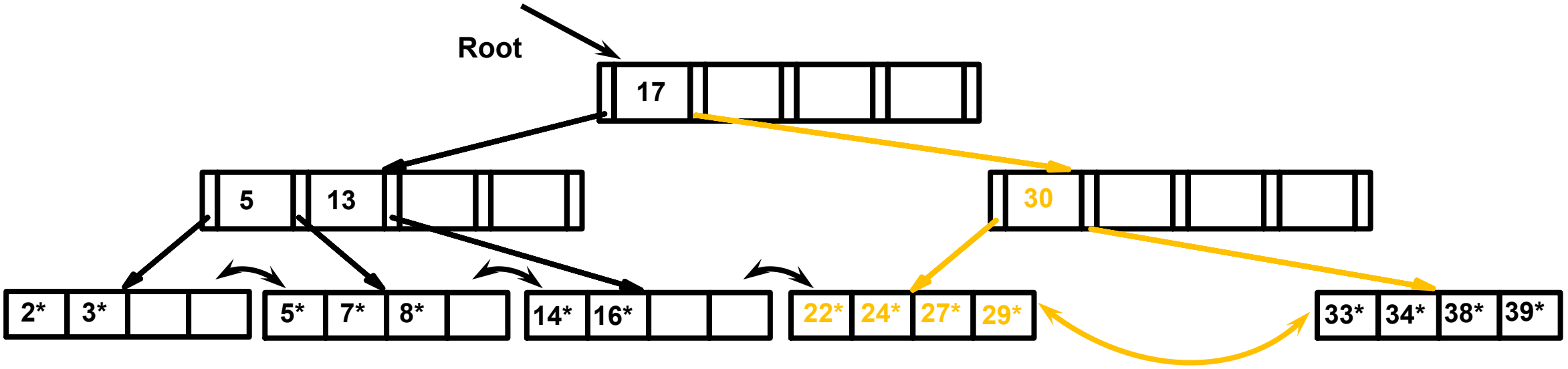
B-Tree deletion example -- deleting 20* with merging

- Deleting 20* with merging. Index entry pointing the right sibling is deleted.



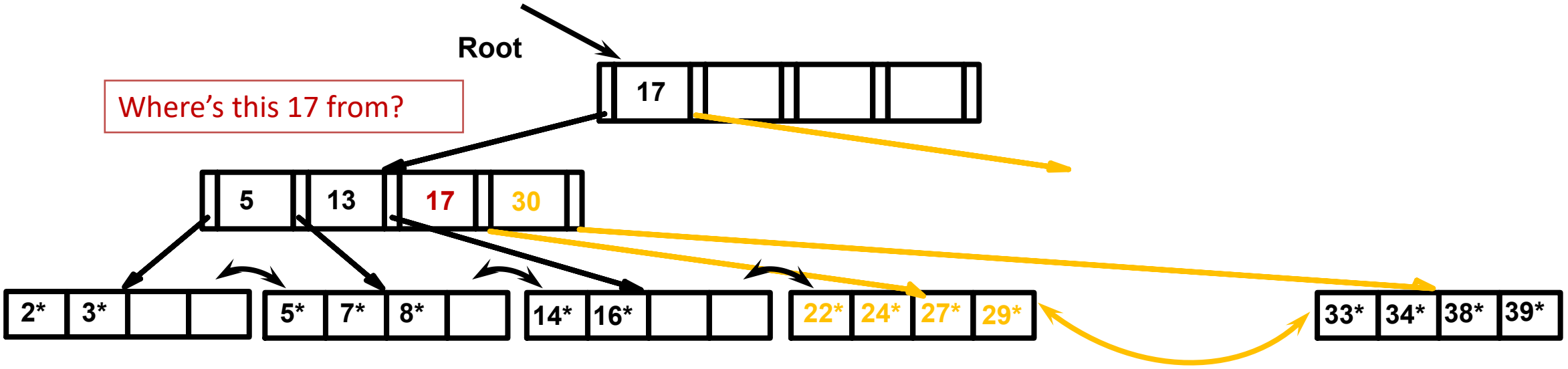
B-Tree deletion example -- deleting 20* with merging

- Deleting 20* with merging. Index entry pointing the right sibling is deleted.
  - Internal page is also under-utilized at this point, merge it with sibling.



B-Tree deletion example -- deleting 20* with merging

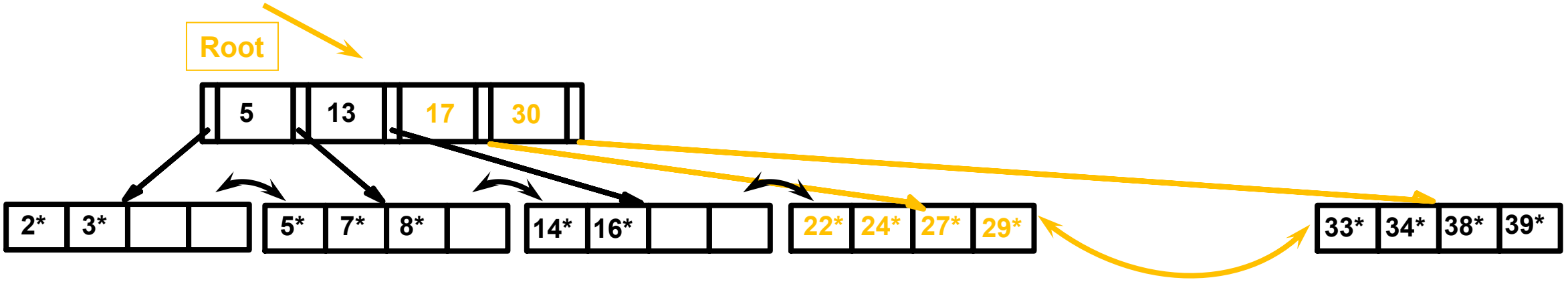
- Deleting 20* with merging. Index entry pointing the right sibling is deleted.
  - Internal page is also under-utilized at this point, merge it with sibling.
  - Root would have only one pointer at this point if we remove the index entry to the right sibling
    - need to remove the root page at this point



B-Tree deletion example -- deleting 20* with merging

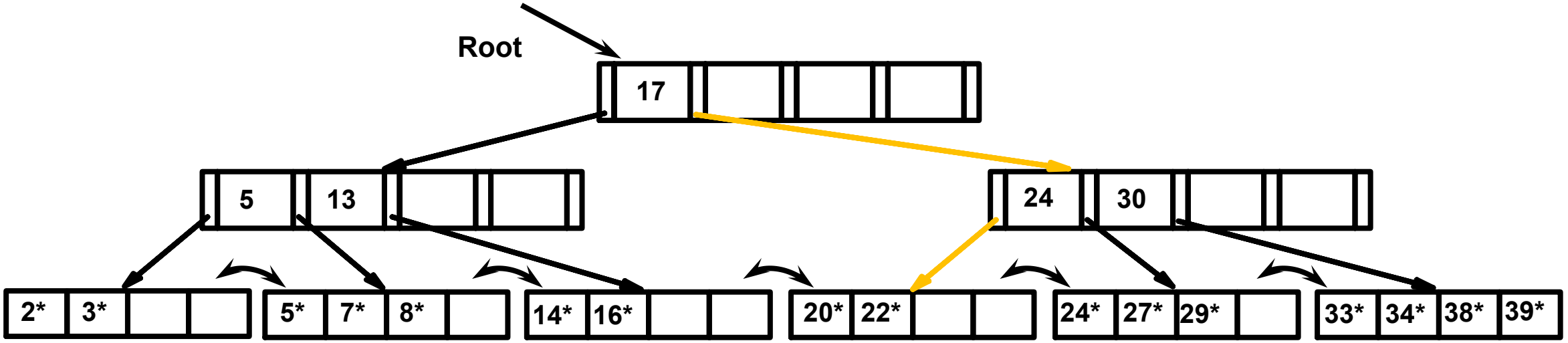
- Deleting 20* with merging. Index entry pointing the right sibling is deleted.
  - Internal page is also under-utilized at this point, merge it with sibling.
  - Root would have only one pointer at this point if we remove the index entry to the right sibling
    - need to remove the root page at this point

Cost = up to $4h$ I/Os.



B-Tree deletion example -- deleting 20* with rebalancing

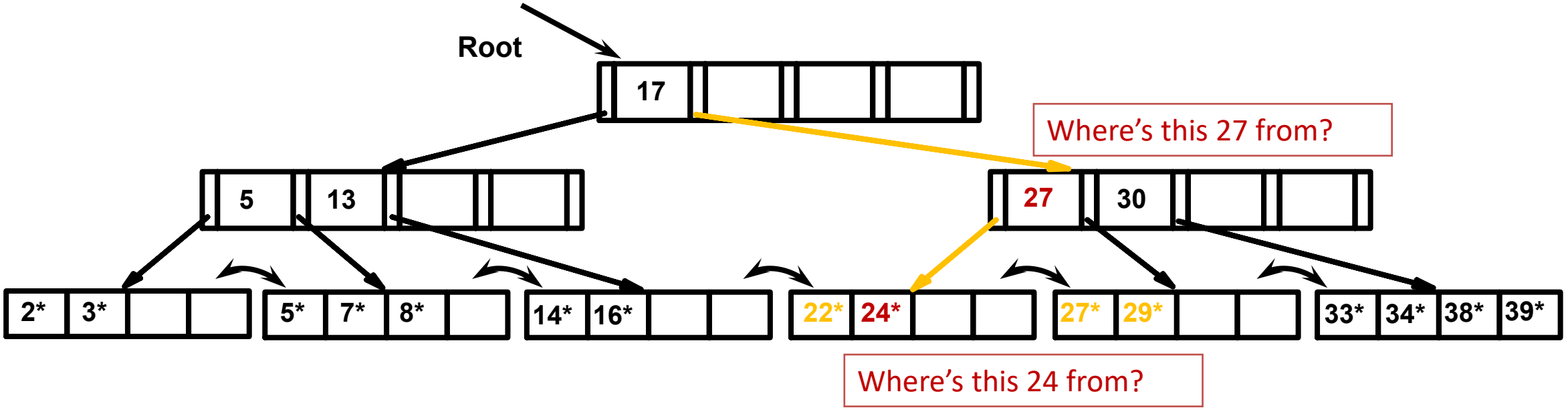
- Deleting 20* with rebalancing. Index entry pointing the right sibling is updated.
  - Copy up of the smallest key on the right page



B-Tree deletion example -- deleting 20* with rebalancing

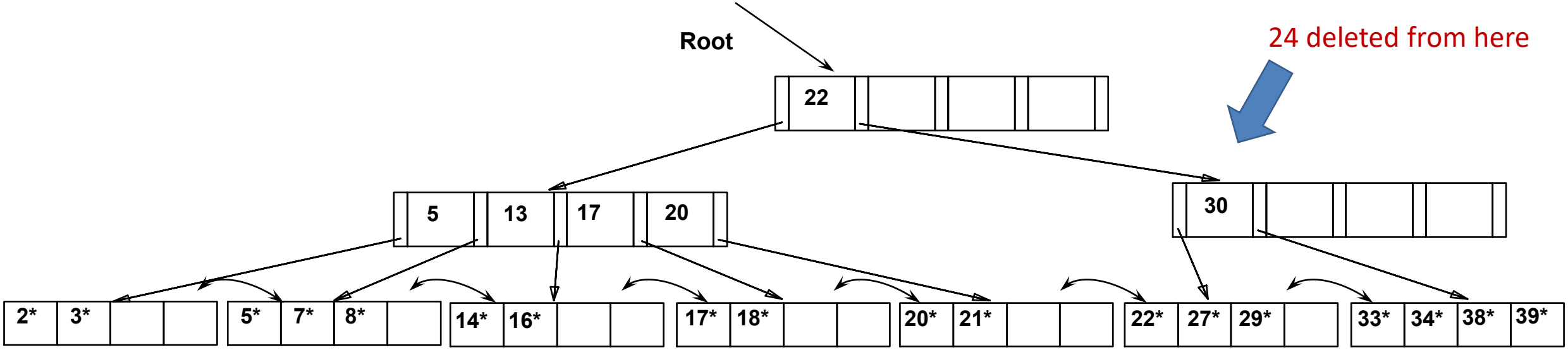
- Deleting 20* with merging. Index entry pointing the right sibling is updated.
  - Copy up of the smallest key on the right page

Cost = h + 5 I/Os.



B-Tree example of non-leaf rebalancing

- Suppose this is the tree we have and we just deleted 24* from the tree
  - which caused a deletion of an index entry on an internal page



B-Tree example of non-leaf rebalacing (cont'd)

- Intuitively, entries are re-distributed by `pushing through' the splitting entry in the parent
- Two choices: either keep 3 or 4 entries on the left page