

CSE462/562: Database Systems (Spring 23)

Lecture 8 & 9: Tree Index

2/23/2023 & 2/28/2023



University at Buffalo

Department of Computer Science
and Engineering

School of Engineering and Applied Sciences

Range Searches

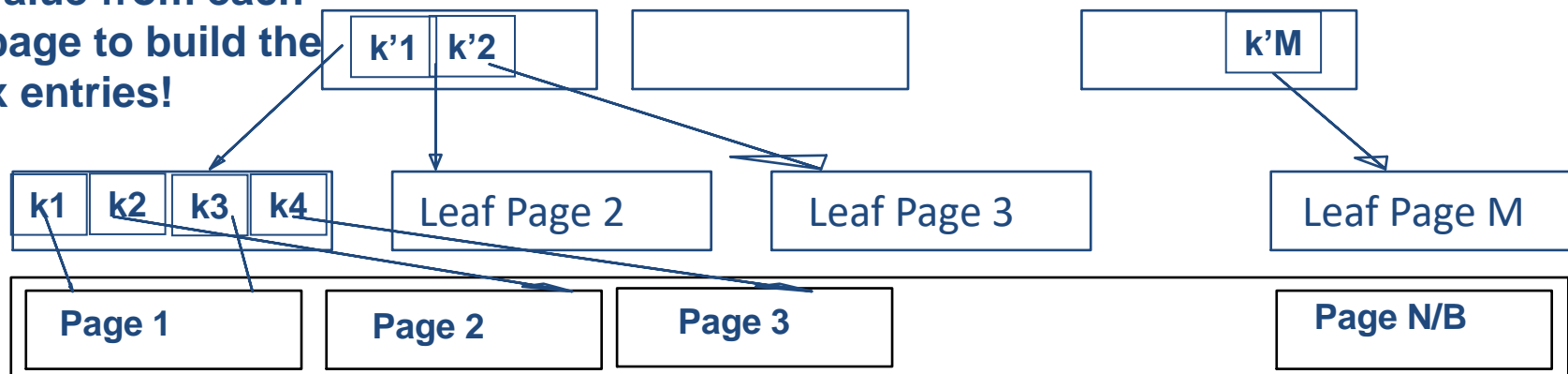
- Find all the students admitted in or after 2020?
 - If data is in sorted file, we can do binary search to find the first; and then scan to find others.
 $O\left(\log_2 \frac{N}{B_h}\right) + \text{scan cost}$ -- N : number of records; B_h : number of records per heap page
 - Cost of binary search can be quite high. Hard to maintain.
- Simple idea: create an index file
 - binary search on the (smaller) index file
 - But the index file could still be quite large
 - Solution: build a new level of indirections

student

sid	name	major	adm_year
100	Alice	CS	2021
101	Bob	CE	2020
102	Charlie	CS	2021
103	David	CS	2020

Internal pages:

Take the smallest search
key value from each
leaf page to build the
index entries!



Leaf Pages with
Data Entries:
1) One data entry
per record!
2) Sort data entries

Data File
With Data Pages

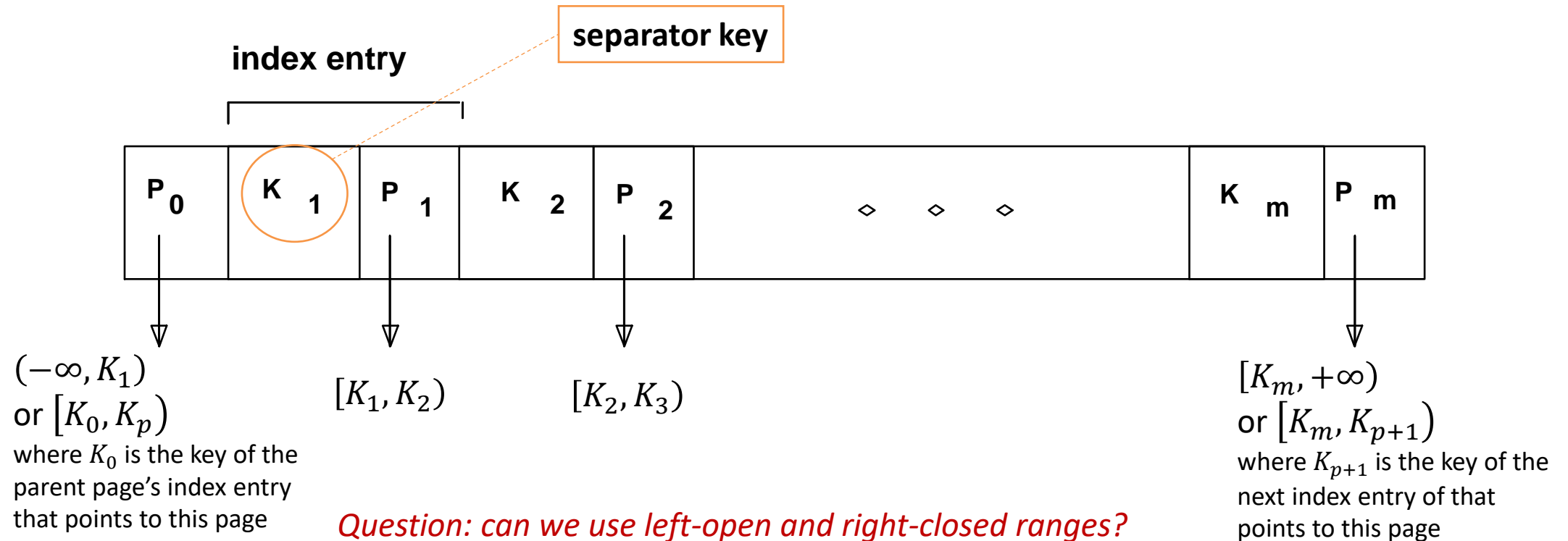
Tree-based Indexes

- *Recall: 3 alternatives for data entries k^* :*
 - Data record with key value k
 - $\langle k, \text{rid of data record with search key value } k \rangle$
 - $\langle k, \text{list of rids of data records with search key } k \rangle$
- Choice is orthogonal to the *indexing technique* used to locate data entries k^* .
- Tree-structured indexing techniques support both *range searches* and *equality searches*.
- ISAM: static structure; B+ tree: dynamic, adjusts gracefully under inserts and deletes.

Index Entries

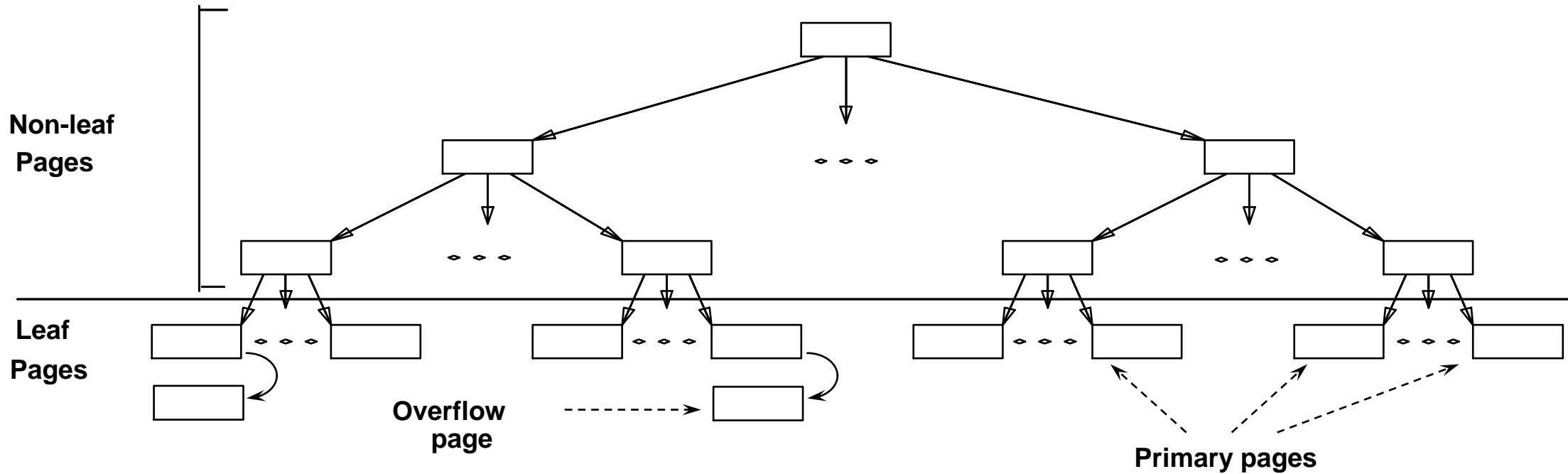
An index entry has the following format: (search key value, page id). The following shows an index page with m index entries (pay attention to the special “left-most pointer”)

Note: entry 0 does not have a key; the range is implicitly defined by left child and K_1



ISAM

- *Static* structure built based on the content of a heap file.
- Supports insert/delete/search.
 - Overflow pages for excessive insertions



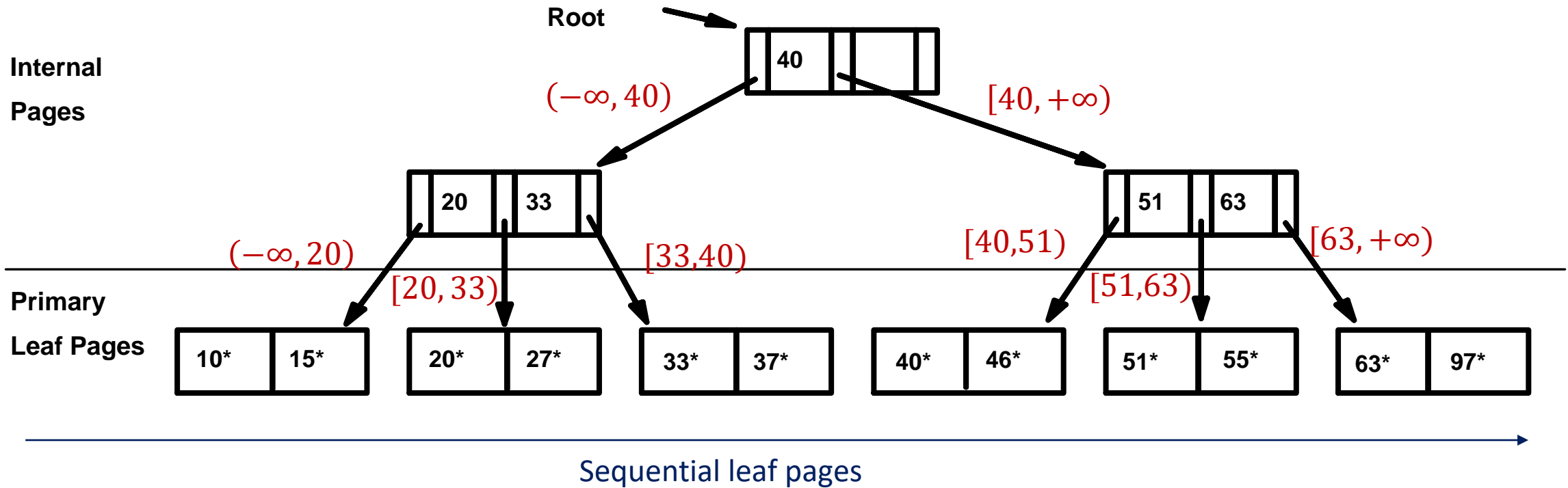
Leaf pages contain data entries.

ISAM Details

- *File creation*: With data pages in a heap file loaded.
Leaf (data) pages allocated sequentially, and data entries sorted by search key;
Then index pages allocated.
Then space for overflow pages.
 - *Index entries*: <search key value, page id>; they 'direct' search for *data entries*, which are in leaf pages.
- *Search*: Start at root; use key comparisons to go to leaf.
I/O cost: $O\left(\log_F \frac{N}{B_0}\right)$
F = fan-out, i.e., # entries per index page, N = # data entries, B_0 = # data entries / leaf page
- *Insert*: Find leaf where data entry belongs, put it there.
(Could be on an overflow page).
- *Delete*: Find and remove from leaf; if empty overflow page, de-allocate.
- **Static tree structure**: *inserts/deletes affect only leaf pages.*
 - Not good for files with a lot of insertions/deletions
 - Could have skews/long overflow chains
- No support for variable-length records in the original ISAM design
 - MyISAM supports variable-length records, but no transaction support, no foreign-key integrity constraint support
 - In any case, you should not use ISAM in practice. But it is a good starting point for learning tree indexes.

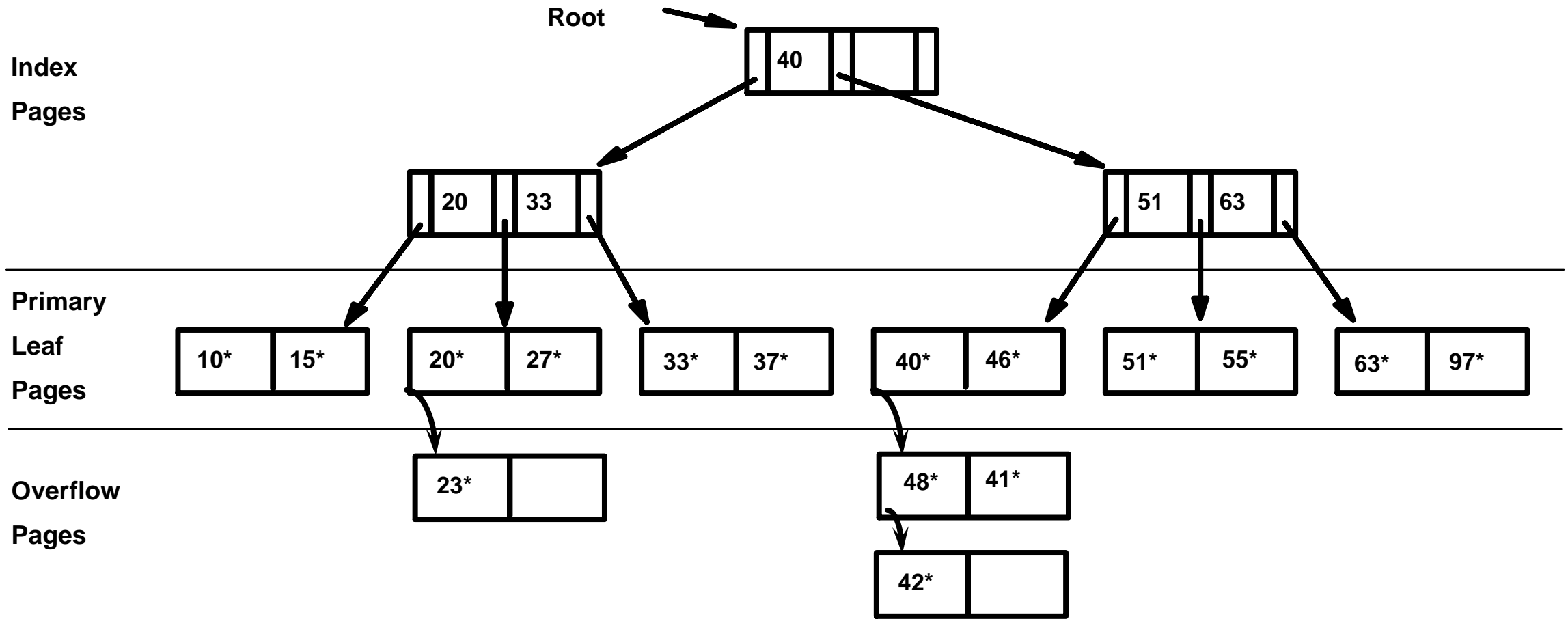
Example ISAM

- e.g., each node can hold 2 data entries or 1 + 2 index entries
 - no need for 'next-leaf-page' pointers. (Why?)



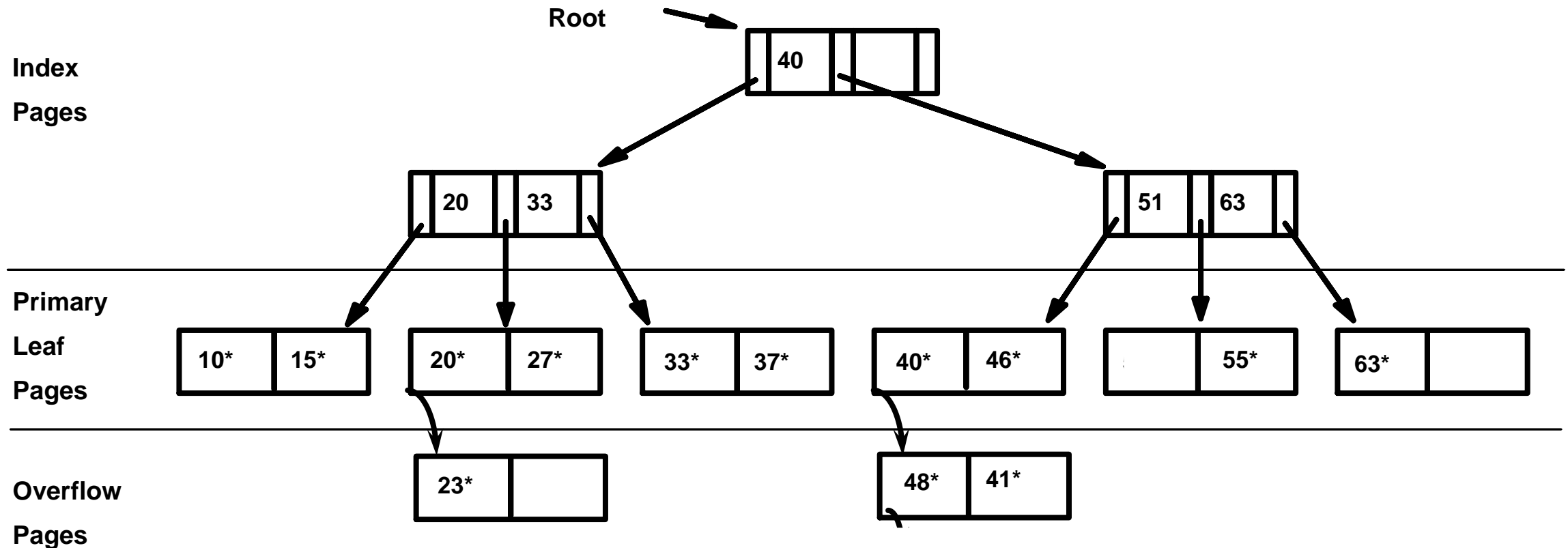
ISAM Insertion Examples

- Inserting 23*, 48*, 41*, 42*



ISAM Deletion Examples

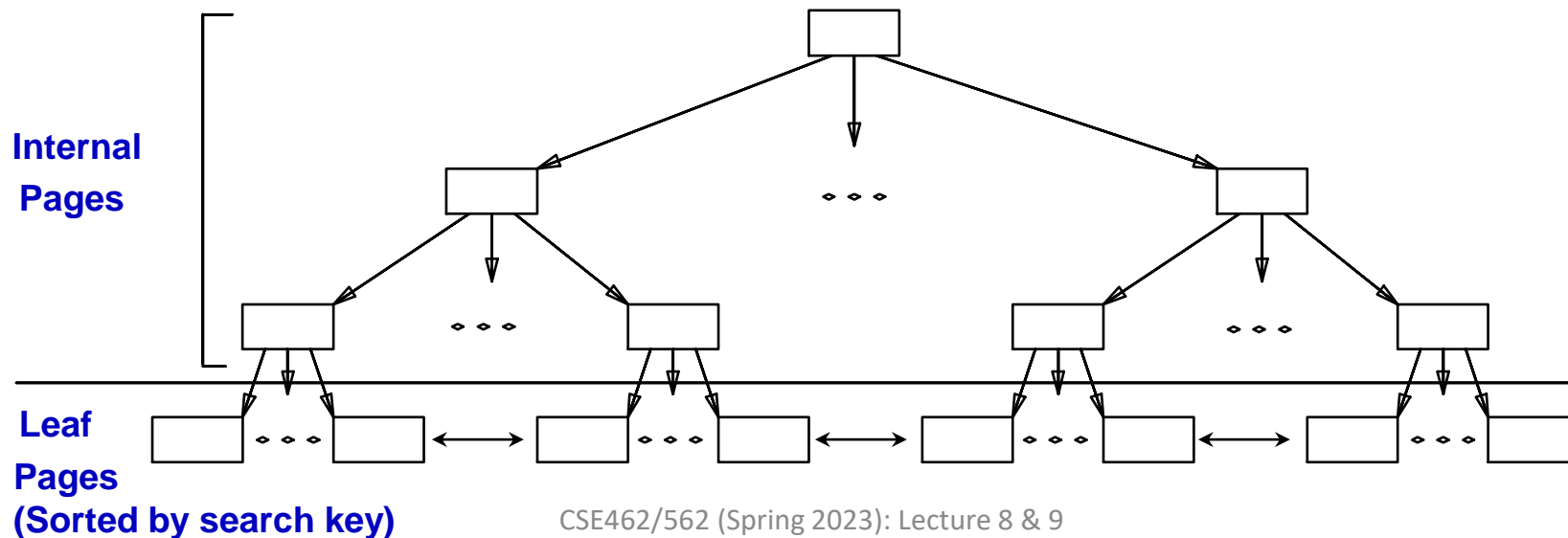
- Deleting 42*, 51*, 97*



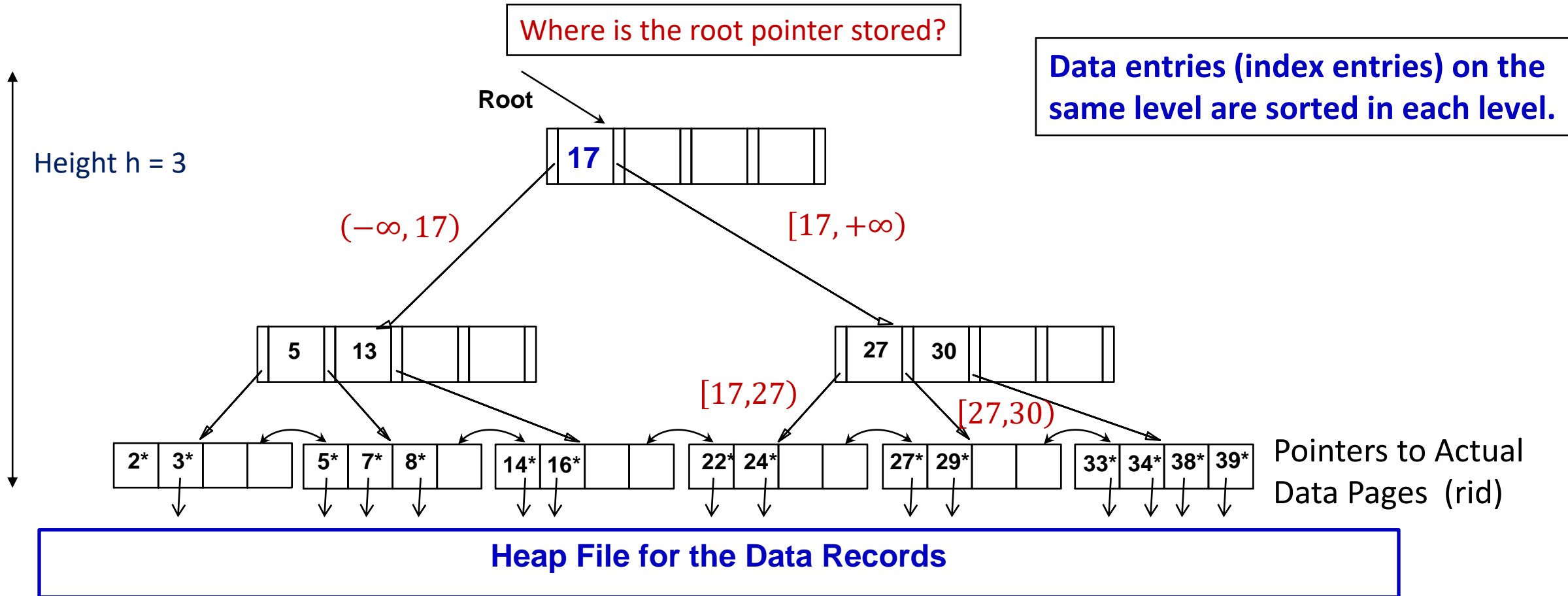
Note that 51 appears in index levels, but 51 not in leaf!*

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 \equiv B-Tree internal/leaf page
- Actually, it's a B+-Tree

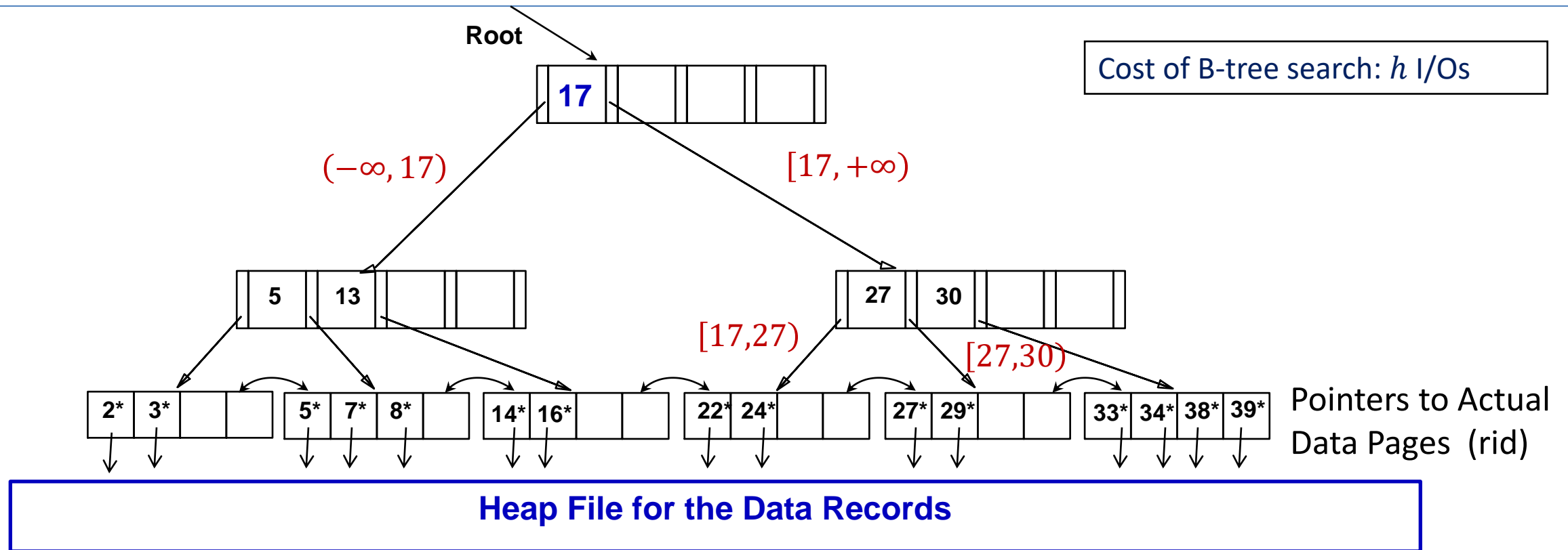


B-Tree example



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

B-Tree search



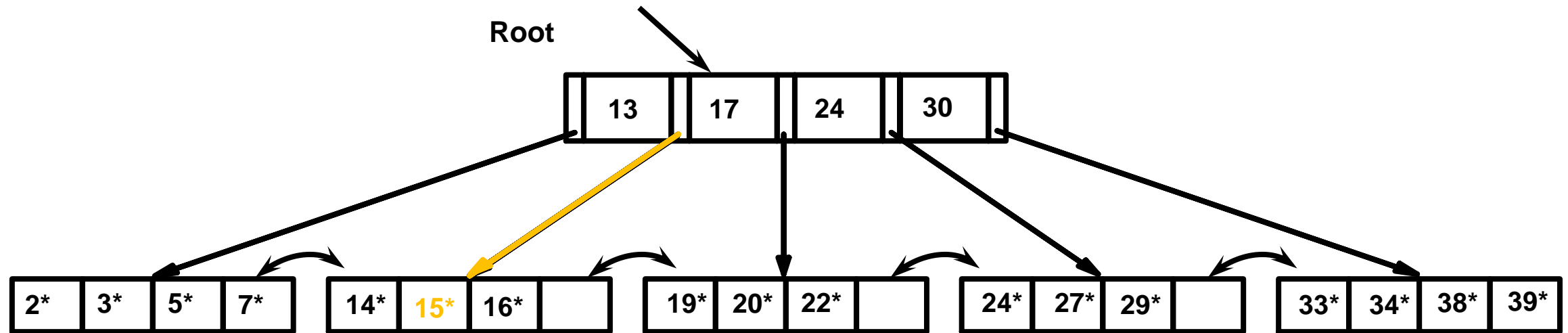
- Find 28*? 29*? All $> 15^*$ and $< 30^*$
 - Starting from root and use key comparison to follow the correct pointers until reaching leaf.
 - To scan a range
 - Locate the lower bound of the key range
 - move right on the data entries until there're no left or you find one that's out of range
 - Can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

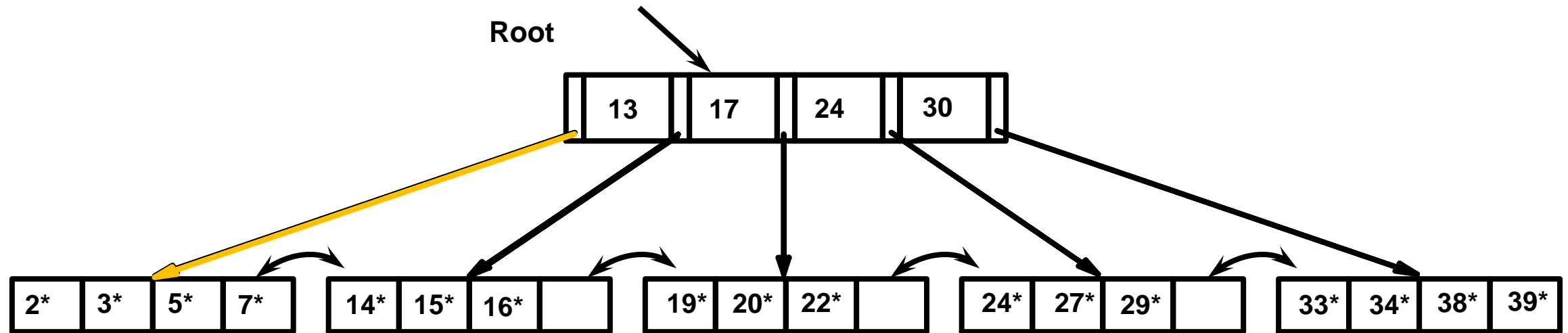
- Inserting 15*



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

B-Tree insertion example -- inserting 8*

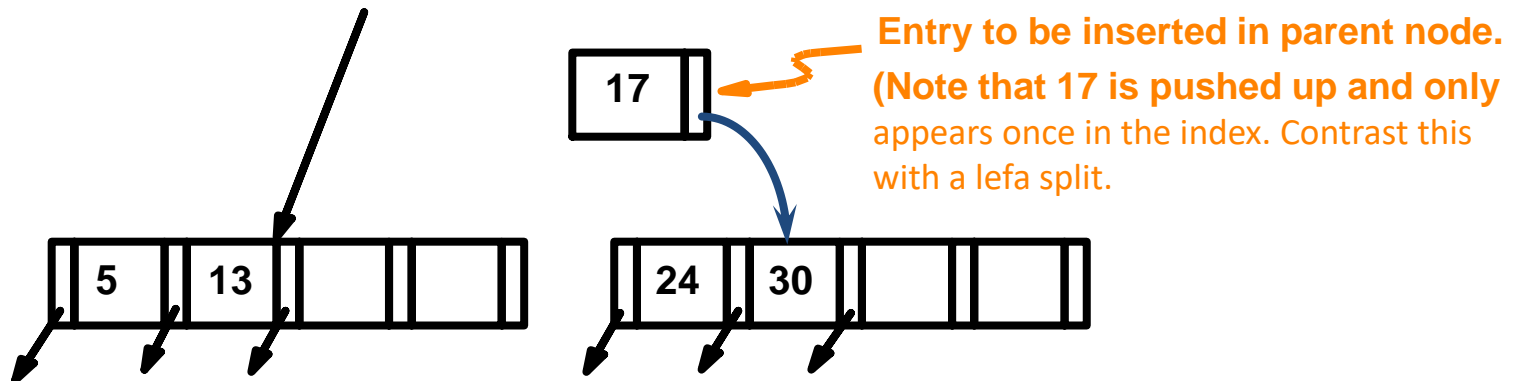
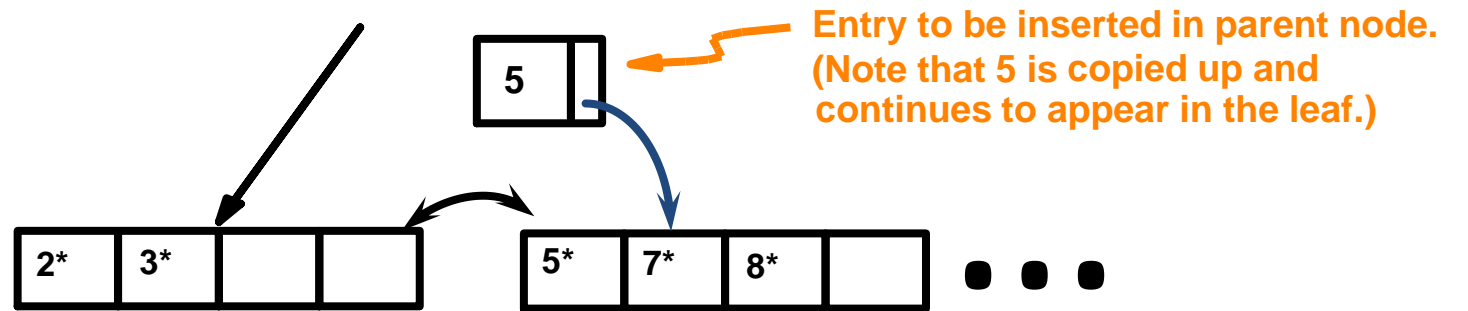
- Inserting 8*



- 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

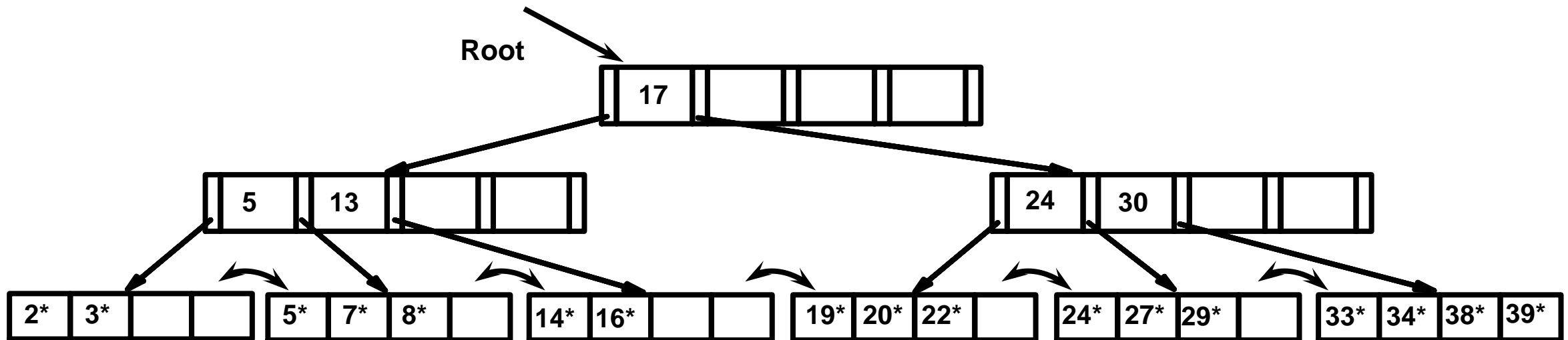
B-Tree insertion example -- inserting 8*

- 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 $4h + 1 = O(h)$ 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.

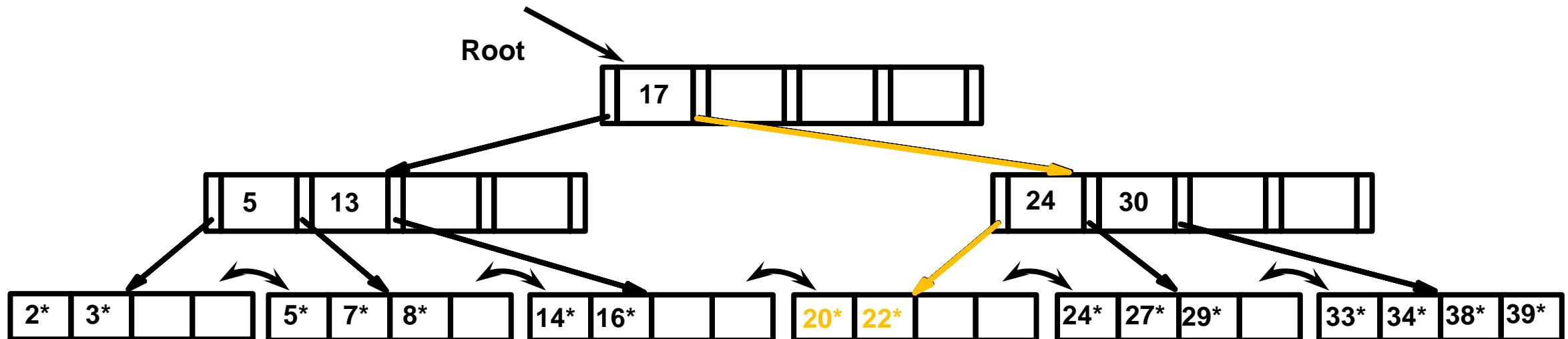
B-Tree deletion

- 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*

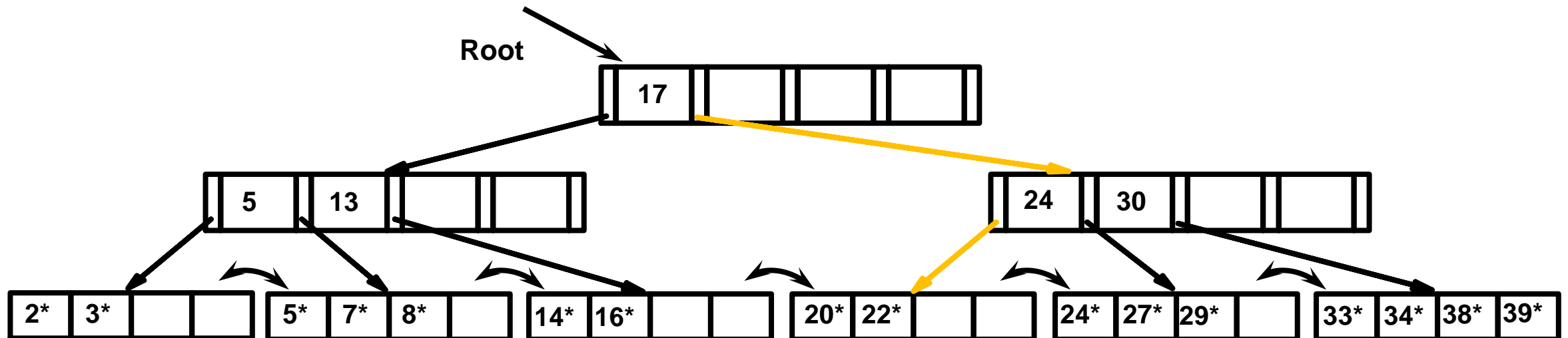
- Deleting 19* is easy.

Cost = h + 1 I/Os.



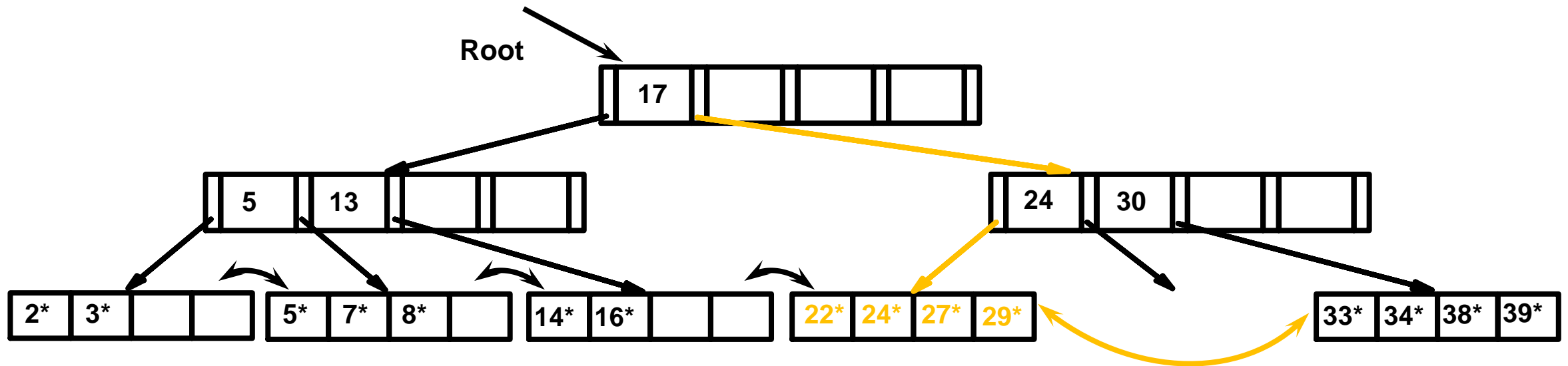
B-Tree deletion example -- deleting 20* with merging

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



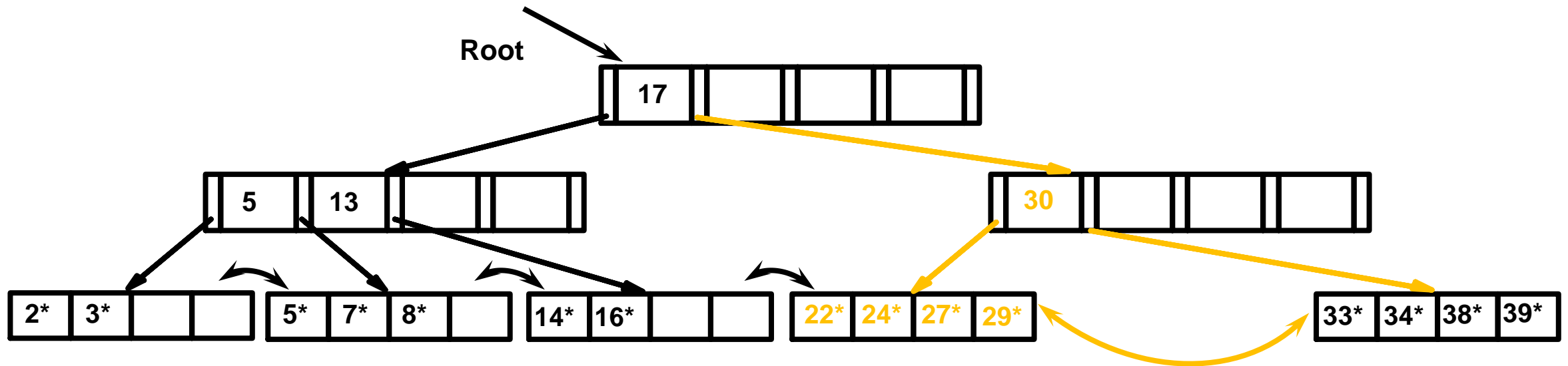
B-Tree deletion example -- deleting 20* with merging

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



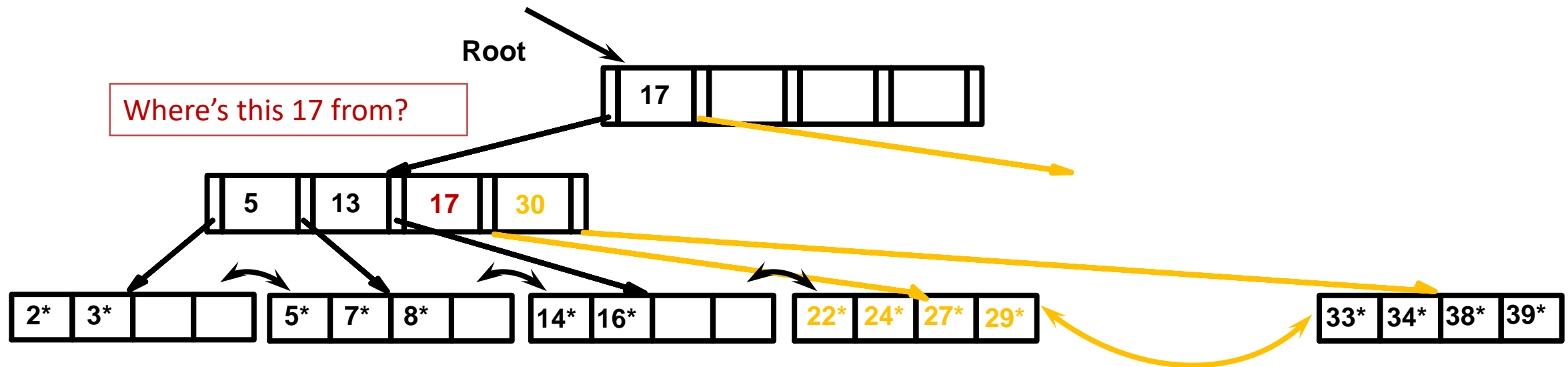
B-Tree deletion example -- deleting 20* with merging

- 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

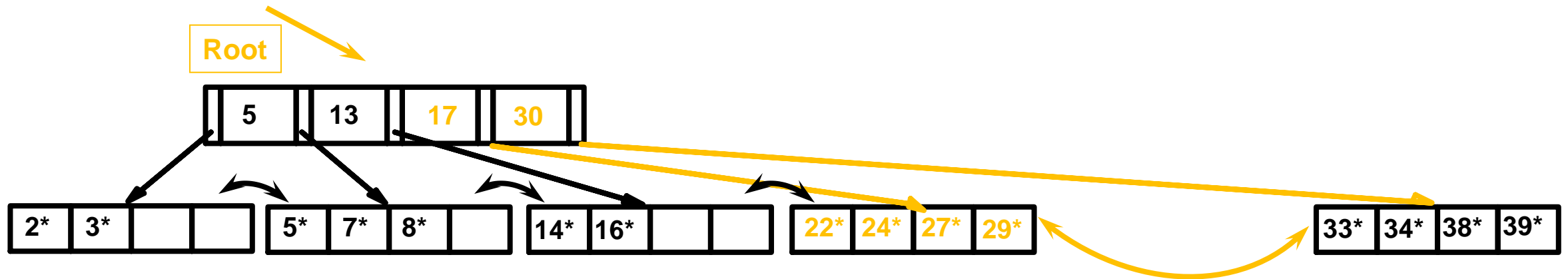
- Deleting 20* with merging. Index entry pointing the right sibling is deleted.
 - Internal page is also under-utilized at this point, merge it with sibling.
 - Root would have only one pointer at this point if we remove the index entry to the right sibling
 - need to remove the root page at this point



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

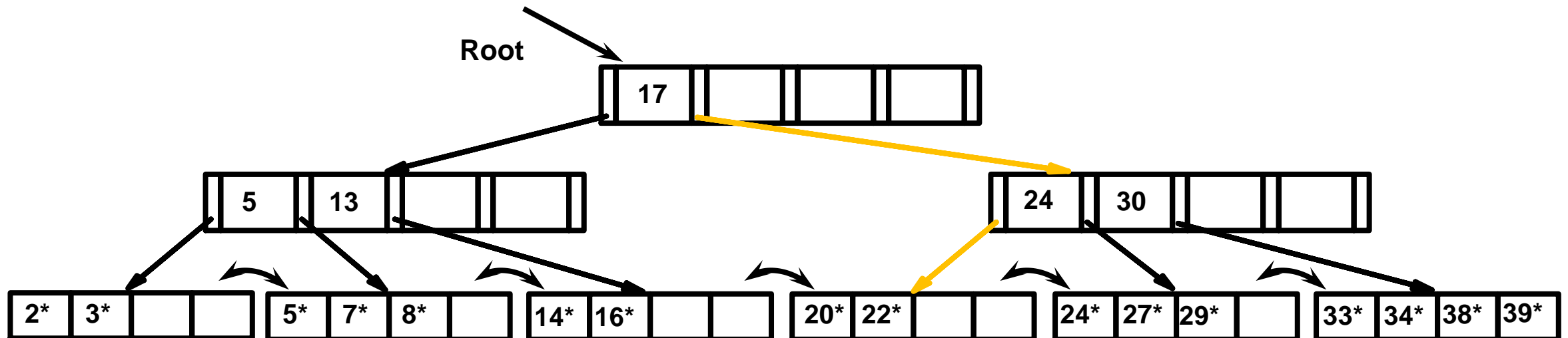
- Deleting 20* with merging. Index entry pointing the right sibling is deleted.
 - Internal page is also under-utilized at this point, merge it with sibling.
 - Root would have only one pointer at this point if we remove the index entry to the right sibling
 - need to remove the root page at this point

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



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

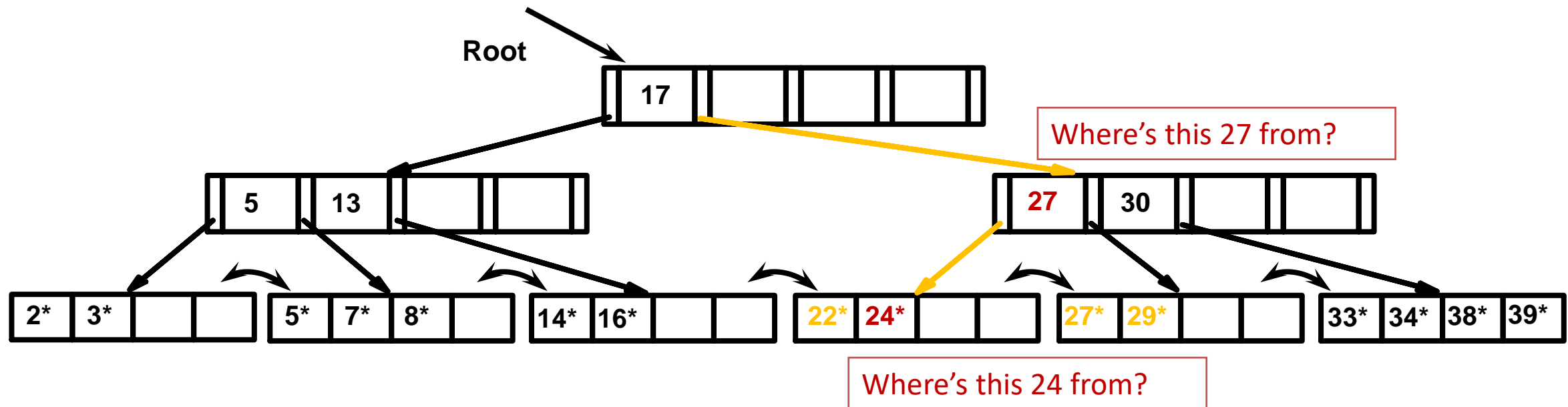
- Deleting 20* with rebalancing. Index entry pointing the right sibling is updated.
 - Copy up of the smallest key on the right page



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

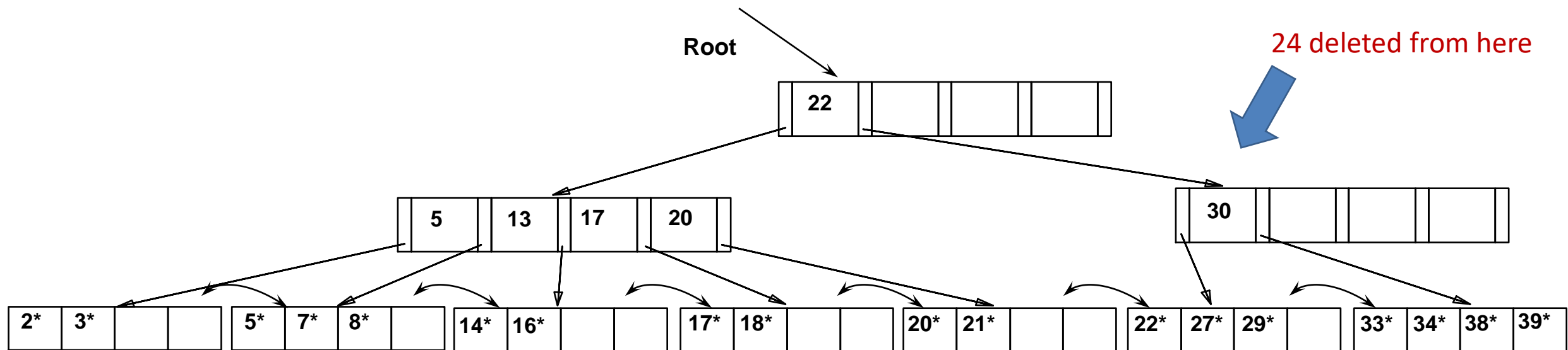
- Deleting 20* with merging. Index entry pointing the right sibling is updated.
 - Copy up of the smallest key on the right page

Cost = $h + 5$ I/Os.



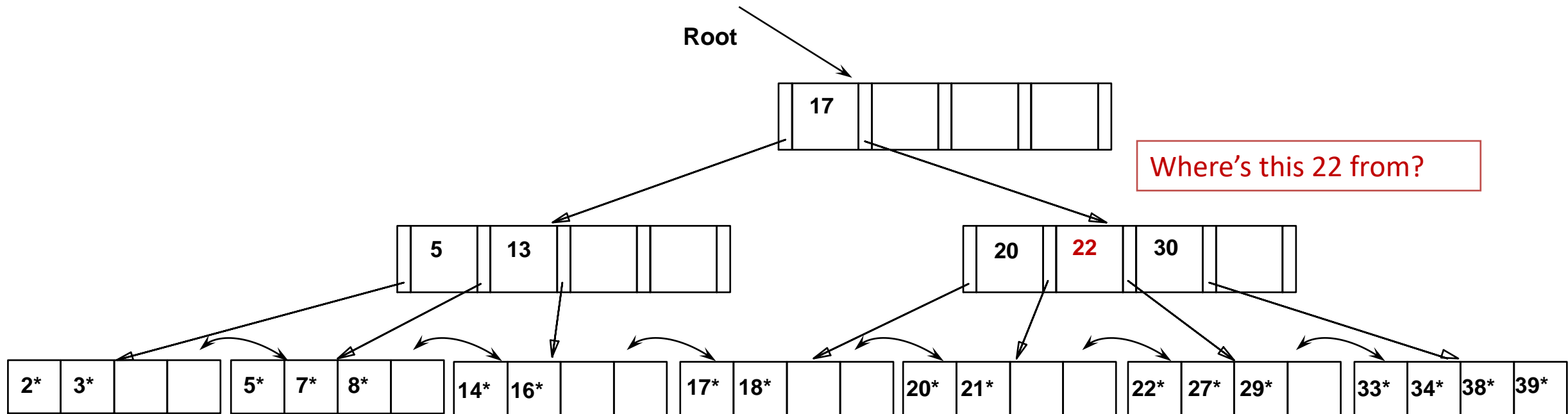
B-Tree example of non-leaf rebalancing

- 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



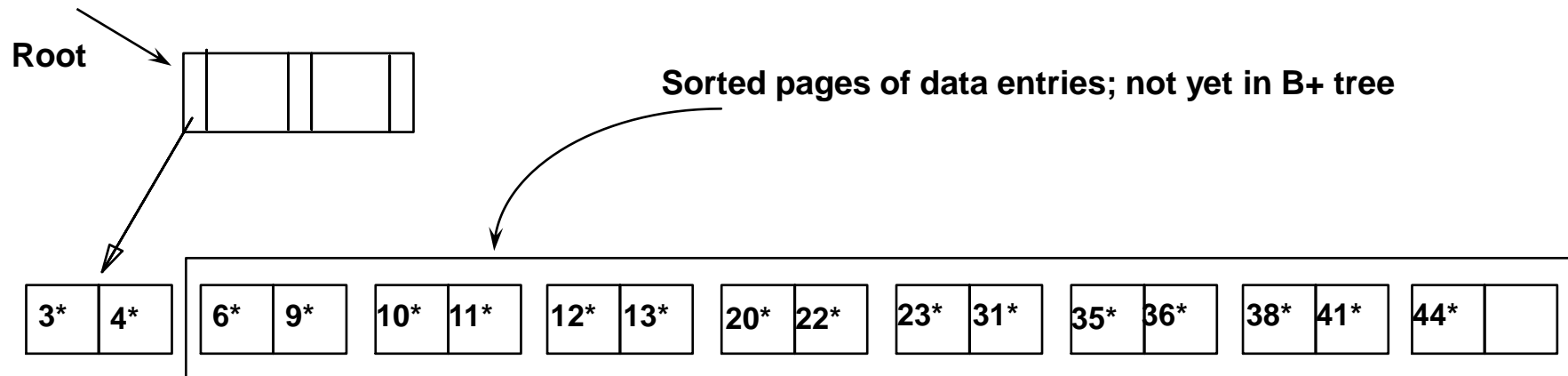
B-Tree example of non-leaf rebalancing (cont'd)

- 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

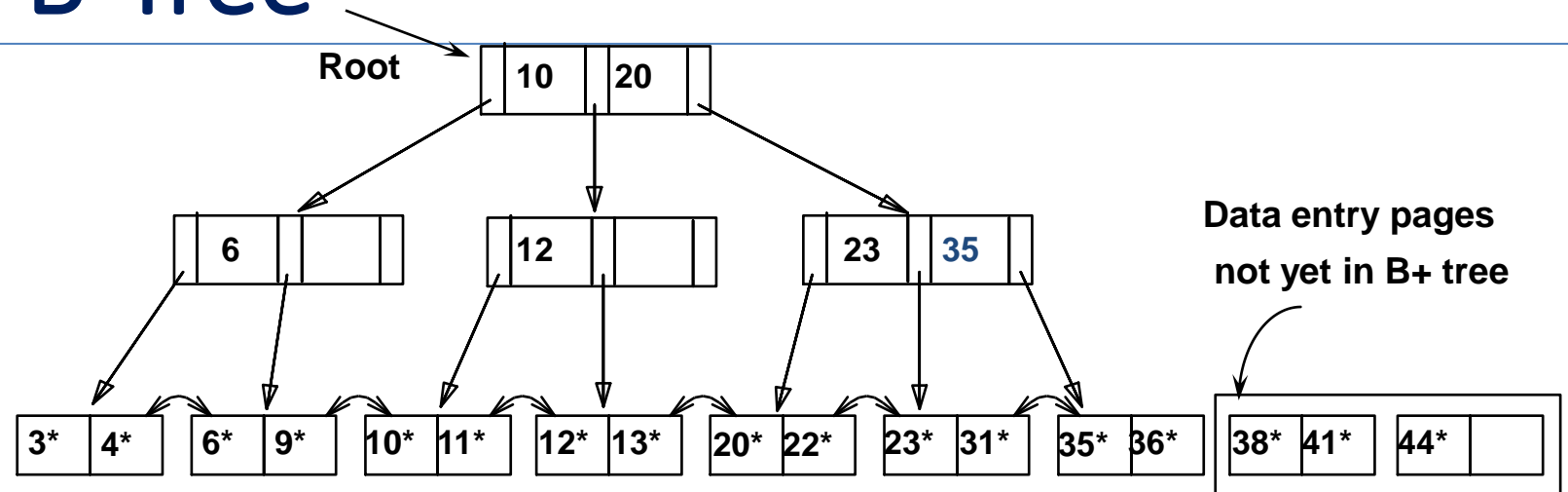


Bulk loading of a B-Tree

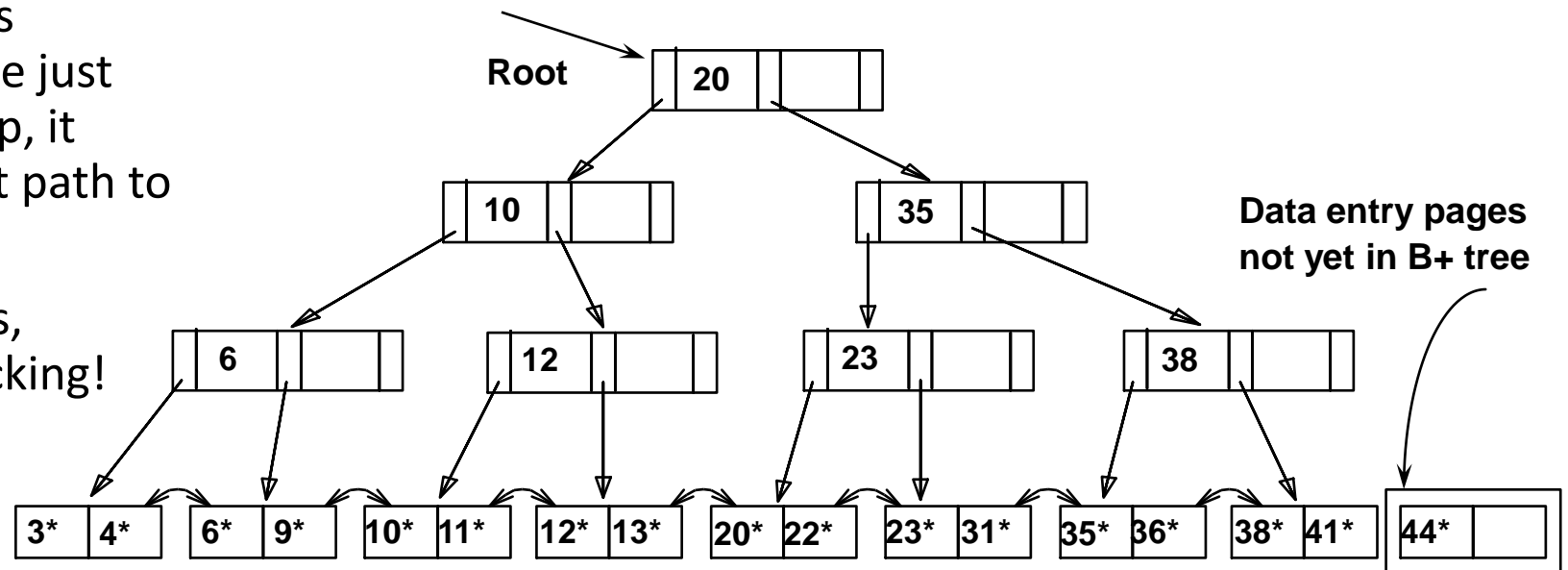
- If we have a large collection of records, and we want to create a B+ tree on some field, doing so by repeatedly inserting records is very slow.
 - Also leads to minimal leaf utilization --- why?
- Bulk loading can be done much more efficiently.
 - fill factor: the default utilization ratio for leaf and internal pages (may vary for leaf and internal pages)
typical values: 70%/80%
- *Initialization*: Sort all data entries, insert pointer to first (leaf) page in a new (root) page.



Bulk loading of a B-Tree



- 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, especially when one considers locking!



Analysis of B-Tree storage cost

- Suppose the usable page size is P (bytes), each record is r (bytes), the index key is k bytes, record ID or page number is q bytes, and N records in total in the heap file.
- Assume we use alternative 2 for the data entries.
- Bottom-up analysis:
 - Number of pages in the heap file: $M = \lceil \frac{N}{\lfloor P/r \rfloor} \rceil$.
 - Number of data entries: N (one per record)
 - Size of a data entry: $k + q$ bytes (without considering alignments)
 - Number of pages in leaf level:
 - $N' = \lceil \frac{N}{\lfloor P/(k+q) \rfloor} \rceil$
 - If the average leaf page utilization ratio is u :
$$N' = \lceil \frac{N}{\lfloor P * u / (k + q) \rfloor} \rceil$$
 - Let B be the number of data entries per leaf page
 - $B = \lfloor P * u / (k + q) \rfloor$

Analysis of B-Tree storage cost

fill factor: the default utilization ratio
when bulk loading the tree

- Internal levels:

- Fan-out/number of index entries per page

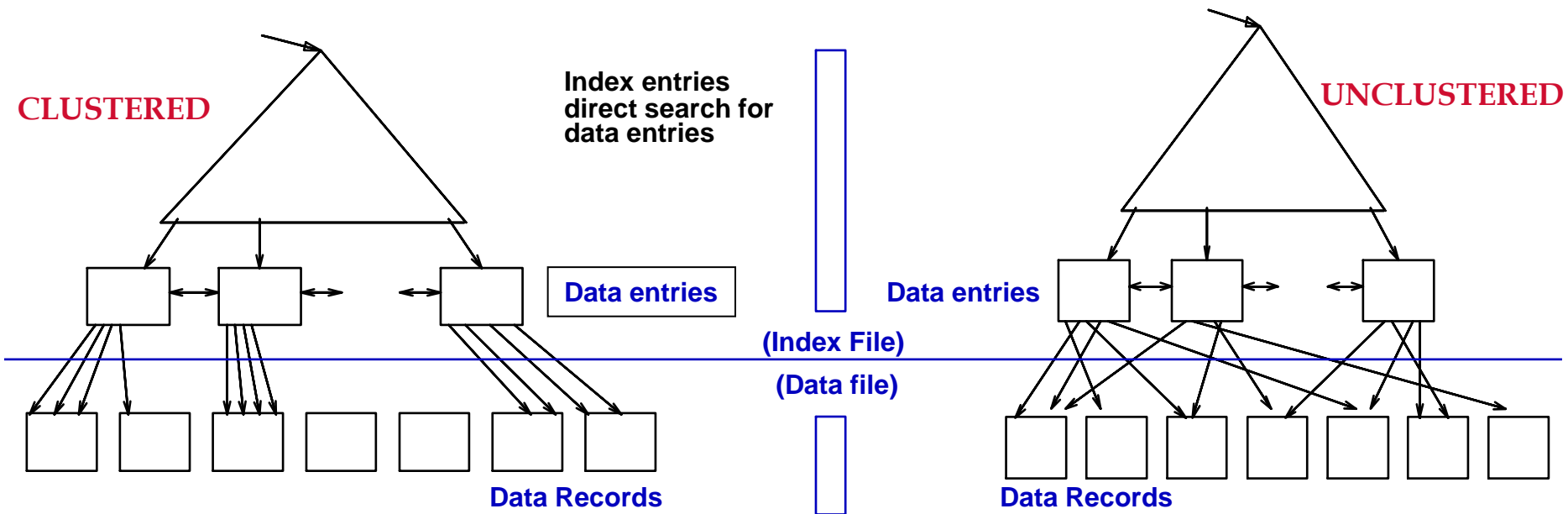
$$f = \left\lfloor \frac{P \times u - q}{k + q} \right\rfloor + 1 \quad (u \text{ is the average utilization ratio: } [0.5, 1))$$

- Number of entries in the index level right above the leaf level: N' (one entry per leaf-level page)
- Number of pages required in this level: N' / f
- Number of entries in the level above: N' / f
- Number of pages in the level above: N' / f^2
- Recursively pages in each level:
 - $N', N'/f, N'/f^2, N'/f^3, \dots, 1 = N'/f^{h-1}$
 - So $h = \lceil \log_f N' \rceil + 1 = \left\lceil \log_f \left\lceil \frac{N}{B} \right\rceil \right\rceil + 1$
 - total number of internal pages $1 + f + \dots + f^{h-1} = \frac{f^h - 1}{f - 1} = O(N') = O(N/B)$

- Total number of pages in a B-Tree: $O(N') = O\left(\frac{N}{B}\right)$

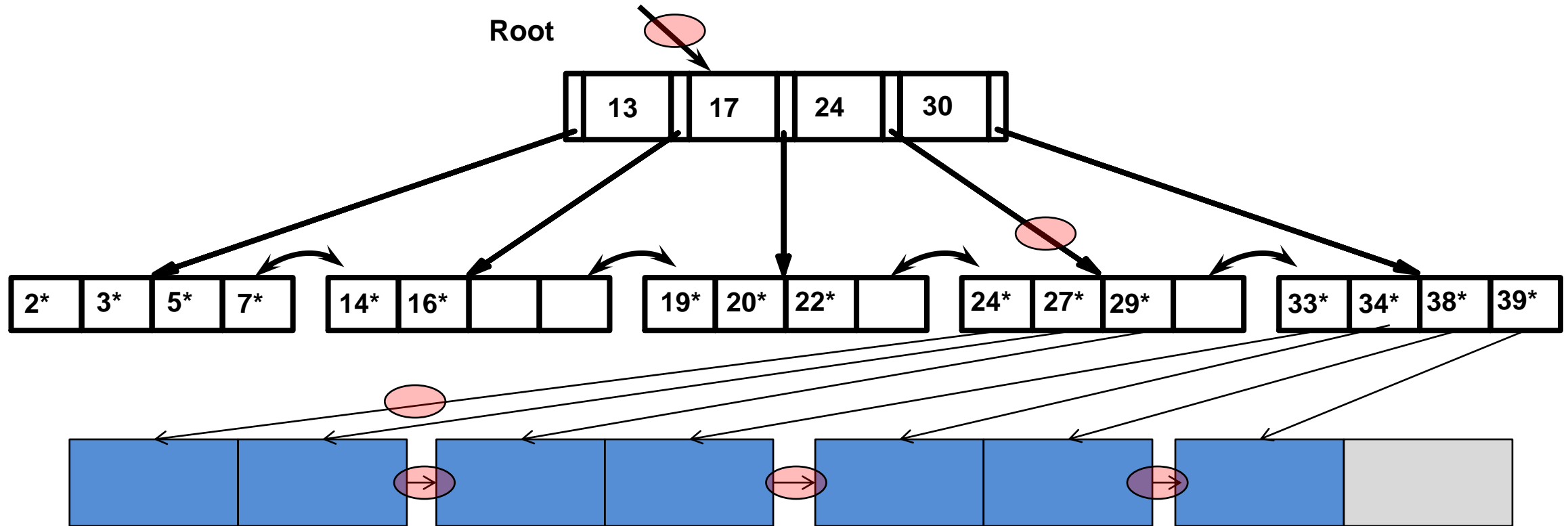
Data access cost using B-Tree

- Recall **clustered vs. unclustered**: if order of data records is the same as, or 'close to', order of index data entries, then called clustered index.
 - Cost of using B-Tree to access records varies a lot depending on whether it is clustered or not*



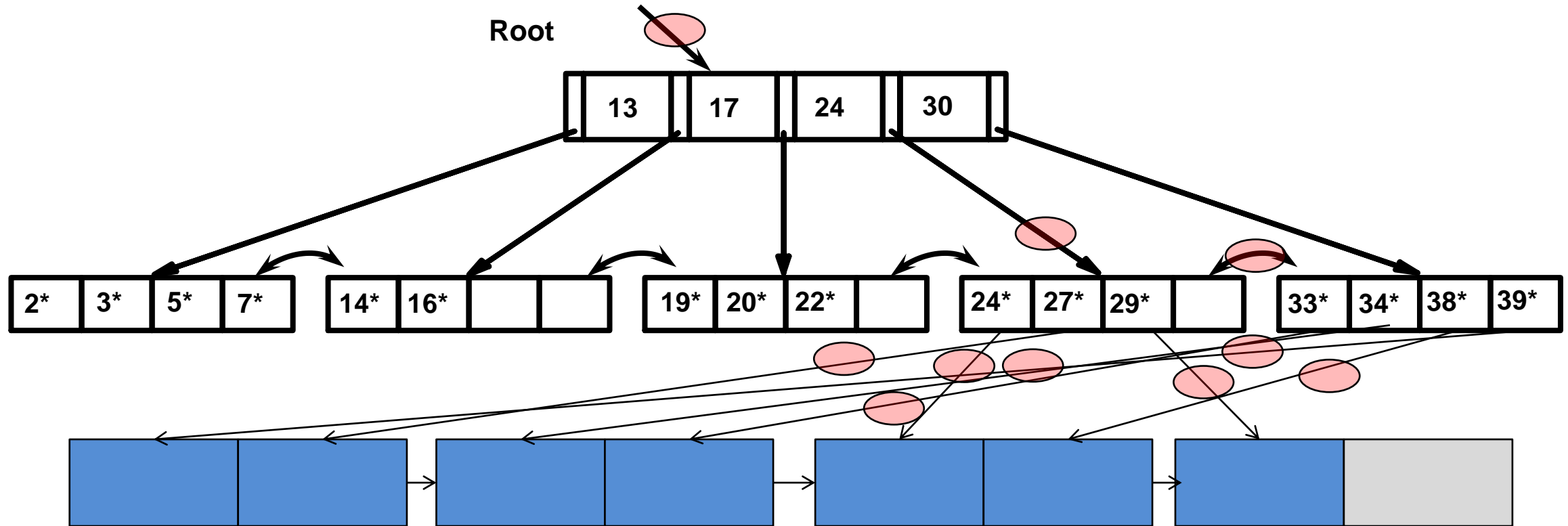
Cost of range scan with clustered B-Tree index

- All records with key ≥ 24 . Clustered index with alternative 2.
 - 6 I/Os
 - 2 random I/O
 - 4 sequential I/O if heap file is laid out sequentially



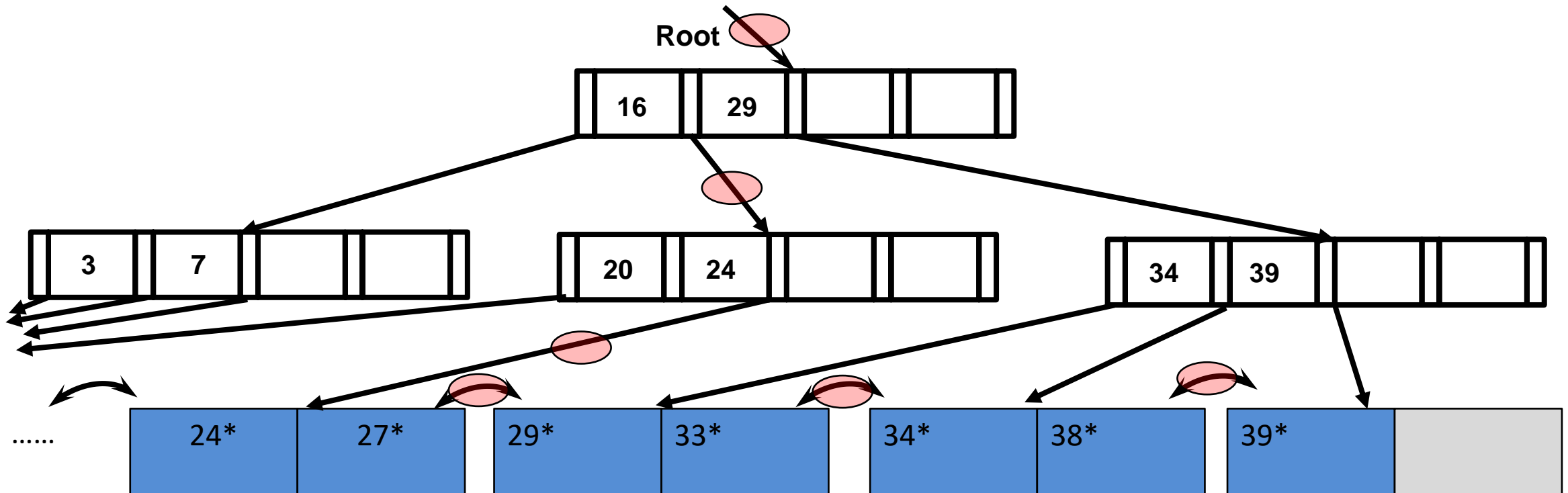
Cost of range scan with unclustered B-Tree index

- All records with key ≥ 24 . Unclustered index with alternative 2.
 - 10 I/Os
 - All random I/Os



Cost of range scan with clustered B-Tree file

- All records with key ≥ 24 . Clustered index with alternative 1.
 - 6 I/Os
 - 3 Random I/O
 - 3 Sequential I/O if the leaf level is sequential in the file

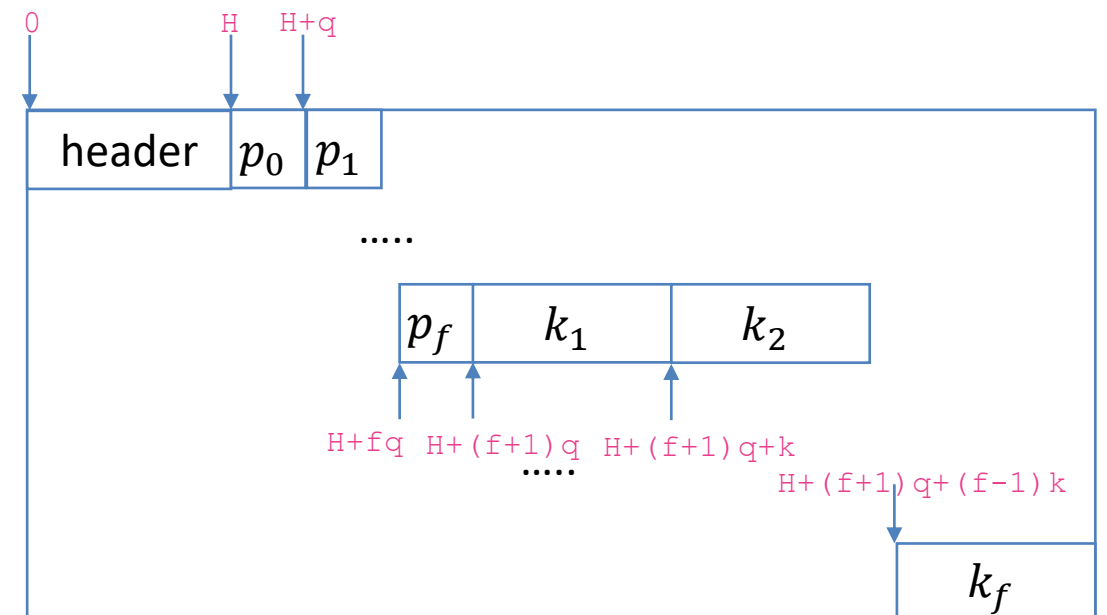
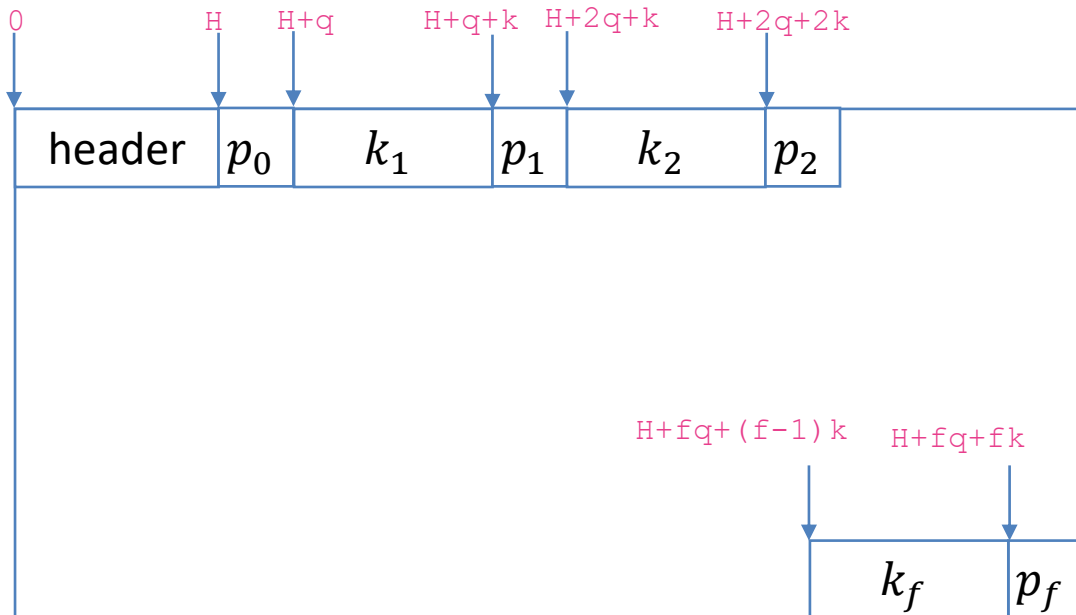


Trade-offs with B-Tree

- Clustered B-Tree
 - One per table
 - Both are good for large range scans, small range scans and point lookups
 - Alternative 2/3 (clustered index)
 - A bit easier to maintain – can be lax on the heap record order (“close to” the data entry order)
 - Alternative 1 (clustered file)
 - Harder to maintain – strictly clustered
 - Need to reorganize the leaf level to make sure they are sequential
 - Save space on data entries (no duplication of keys)
 - Might have larger tree height
- Unclustered B-Tree
 - Usually alternative 2/3
 - Easiest to maintain
 - Not very efficient when range scan covers too many records
 - Rule of thumb: Scan no more than a tiny fraction of rows
e.g., 0.01% on 7200 rpm HDD, 0.1% on consumer-level Nand SSD
(empirical value, it may vary depending on your DBMS and storage device)

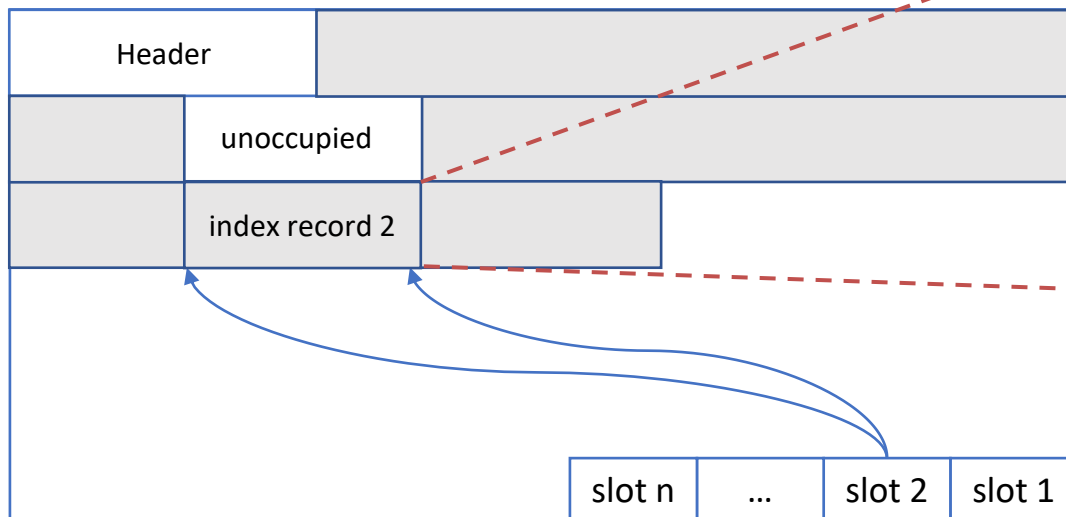
B-Tree in practice: page and record layout

- So far, we considered fixed-length keys => fixed-fanout
 - Easy to define page occupancy in terms of number of slots
 - Easy to implement leaf and internal nodes
 - Option 1: alternating pointers and keys
 - Option 2: two arrays for pointers and keys
 - Both with fixed offsets!



B-Tree in practice: page and record layout

- But, we could have variable-length keys
 - Nullable columns, string keys
- How do you organize the B-tree nodes?
 - Use slotted data page



Index entry

child_pid	key payload
-----------	-------------

Data entry (alternative 2)

recid	index key
-------	-----------

B-Tree in practice: structural modification

- How do you define page utilization?
 - How many bytes are used? How many slots there are?
 - Issues?
- Page split – that's usually ok
- Page merge
 - Leaf page merge – no problem
 - Internal page merge -- the key to pull down from the parent page may not fit!
- Page rebalance
 - Leaf or internal page rebalance
 - the key to copy/push up may not fit in the parent page!
 - Internal page rebalance:
 - the key to pull down from the parent page may not fit here!
 - Rarely implemented -- also makes concurrency control hard

B-Tree in practice: multi-field keys

- Multi-field keys are totally ordered in the lexicographical order (aka dictionary order)
 - e.g., (a, b, c), order by a first, then b, finally c
- Multi-field keys in B-Tree is very useful
 - You can answer certain queries with predicates of a prefix of the keys
 - For instance, with a B-Tree over (age, gpa) , it may be used for answering the following queries:
 - $age \geq 20 \wedge age \leq 25$
 - $age = 20 \wedge gpa \geq 3.0$
 - What about $age \geq 20 \wedge gpa \geq 3.0$?
 - Strategy 1: using B-Tree to locate the first data entries with $(age = 20 \wedge gpa \geq 3.0) \vee age > 20$ then scan all data entries starting from that
 - Strategy 2: for each of the distinct $age \geq 20$, locate the first data entry with $gpa \geq 3.0$ then scan data entries starting from these first data entries separately (aka index skip scan (e.g., Oracle) /jump scan (e.g., DB2) in various systems)
Strategy 2 only works when there are few distinct values in the prefix column

B-tree in practice: NULL values

- We need to index NULL values in B-tree indexes
 - because indexed columns may have NULLs
- Caveat: SQL 3-value logic
 - *NULL < anything is unknown!*
 - B-tree requires a total order of the key
- Solution: don't use the SQL 3-value logic
 - *For instance, define NULL = NULL, NULL < any non-NULL value*
 - *Alternatively, NULL = NULL, NULL > any non-NULL value*
 - Some systems support both
 - In the course project Taco-DB, we assume NULL < any non-NULL value for indexing

B-Tree in practice: non-unique keys

- So far, we assumed unique keys, but
 - we might create indexes over non-unique columns (e.g., name)
- B-Tree can be modified to support duplicate keys, but
 - How do you find the data entry for a specific record for update?
- What if we still want to uniquely identify keys in the tree?
 - Include record ID as the last column
 - record IDs are always unique
 - Then a search with key in B-Tree only becomes prefix search:
 - e.g., key = (age, gpa), actual key = (age, gpa, record id)
 - Query: $age = 22 \wedge gpa = 3.7$?
 - Locate the first data entry such that $(age = 22 \wedge gpa \geq 3.7) \vee age > 22$
 - Then scan the data entries until it falls out of range
 - To uniquely locate a data entry for a record: use the full search key

B-Tree in practice: unique constraints

- B-Tree are often used for enforcing UNIQUE constraints
 - e.g., `sid SERIAL PRIMARY KEY`
 - e.g., `login VARCHAR(20) UNIQUE`
- Build unique B-tree index
 - Reject insertion of a data entry whose key already exists in another data entry in the index
 - even if the record id does not match
- However, what about NULLs?
 - Nullable unique column is allowed to contain multiple NULLs (because they are unknown values)
 - Reality: some allow and some don't
 - Some DBMS disallows inserting multiple NULLs into unique B-Tree index
 - non-conformant to SQL, but easier to implement (no special case handling)
 - Some do allow that
 - SQL-conforming, but need special handling logic for that

B-Tree in practice: handling concurrency

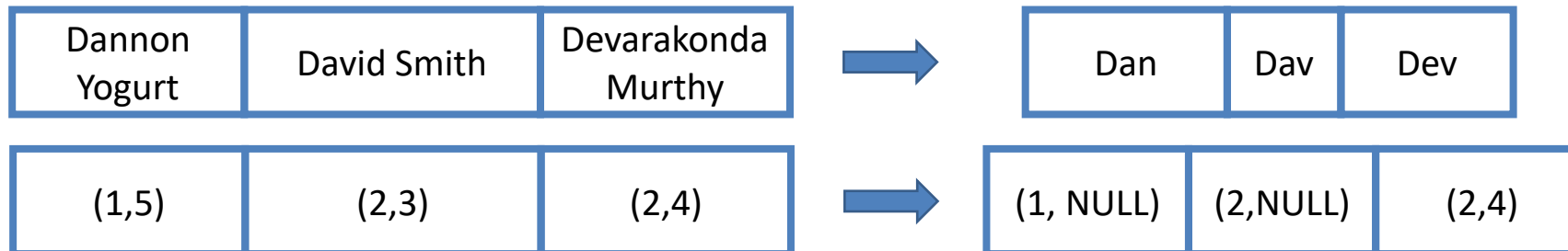
- Lock-based (e.g., reader-writer lock, in DBMS jargon: latches)
 - Many issues:
 - Should lock at most c pages at a time (c usually is 1/2/3)
 - Lock coupling order (deadlock avoidance)
 - Insertion:
 - Split will cause key space shift (how does concurrent search handle this?)
 - Root split? How to install the new root with concurrent readers?
 - Deletion (harder):
 - Page merge/reducing tree height: also causes key space changes
 - Some design avoids them by deleting a page only when it's completely empty
 - Some design use mini transactions to handle SMO
 - File space management:
 - What if a page is deleted but a concurrent reader reaches the deleted page?
 - Recovery: what if crashes and we have to roll back a half completed B-tree update?
- Lock-free
 - Using CAS and additional indirection ([J. Levandoski, D. Lomet, S. Sengupta. ICDE '13](#))
 - Other considerations?

B-Tree in practice: key compression

- We want high fan-out \rightarrow low tree height \rightarrow faster query/update
 - But string keys are often quite long (tens of bytes vs 4 bytes/8 bytes)
- Prefix key compression: extract the common prefix and only store the unique suffix
 - Sorted keys tend to have a short common prefix



- Suffice truncation: store only the prefix that is enough for differentiating the subtree range
 - Works for both string/multi-field keys



Summary

- These lectures:
 - ISAM
 - B-Tree index
 - How to search and scan/insert/delete in B-Tree
 - Analysis of B-Tree index/file
 - B-Tree in practice
- Next lecture:
 - Indexing and cost analysis