Database Applications (15-415)

DBMS Internals- Part III
Lecture 12, February 22, 2015

Mohammad Hammoud

جامعة كارنيجي ميلون في قطر
Carnegie Mellon University Qatar

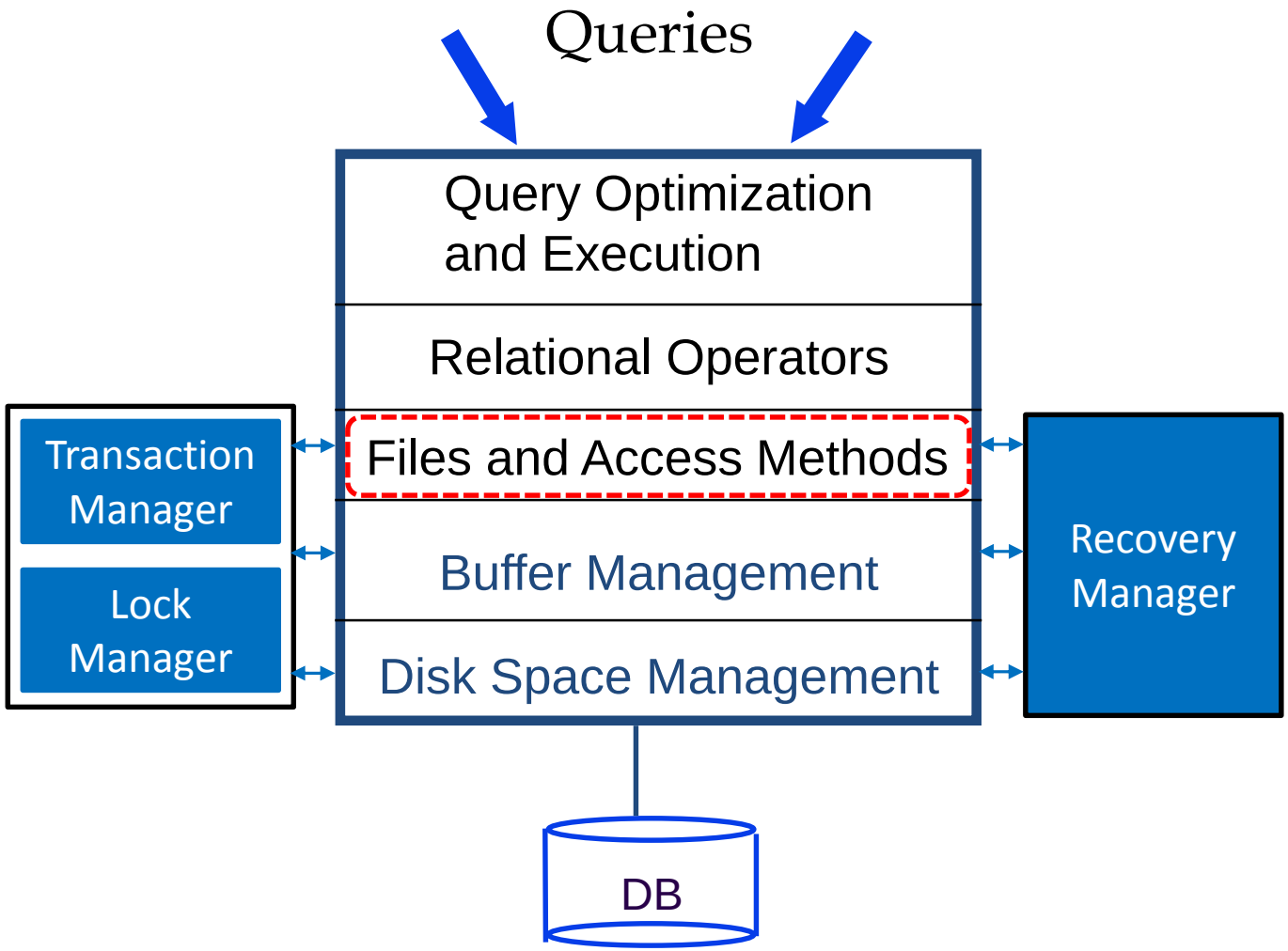
Today…

- Last Session:
  - DBMS Internals- Part II
    - Buffer Management
    - Files and Access Methods (file organizations)

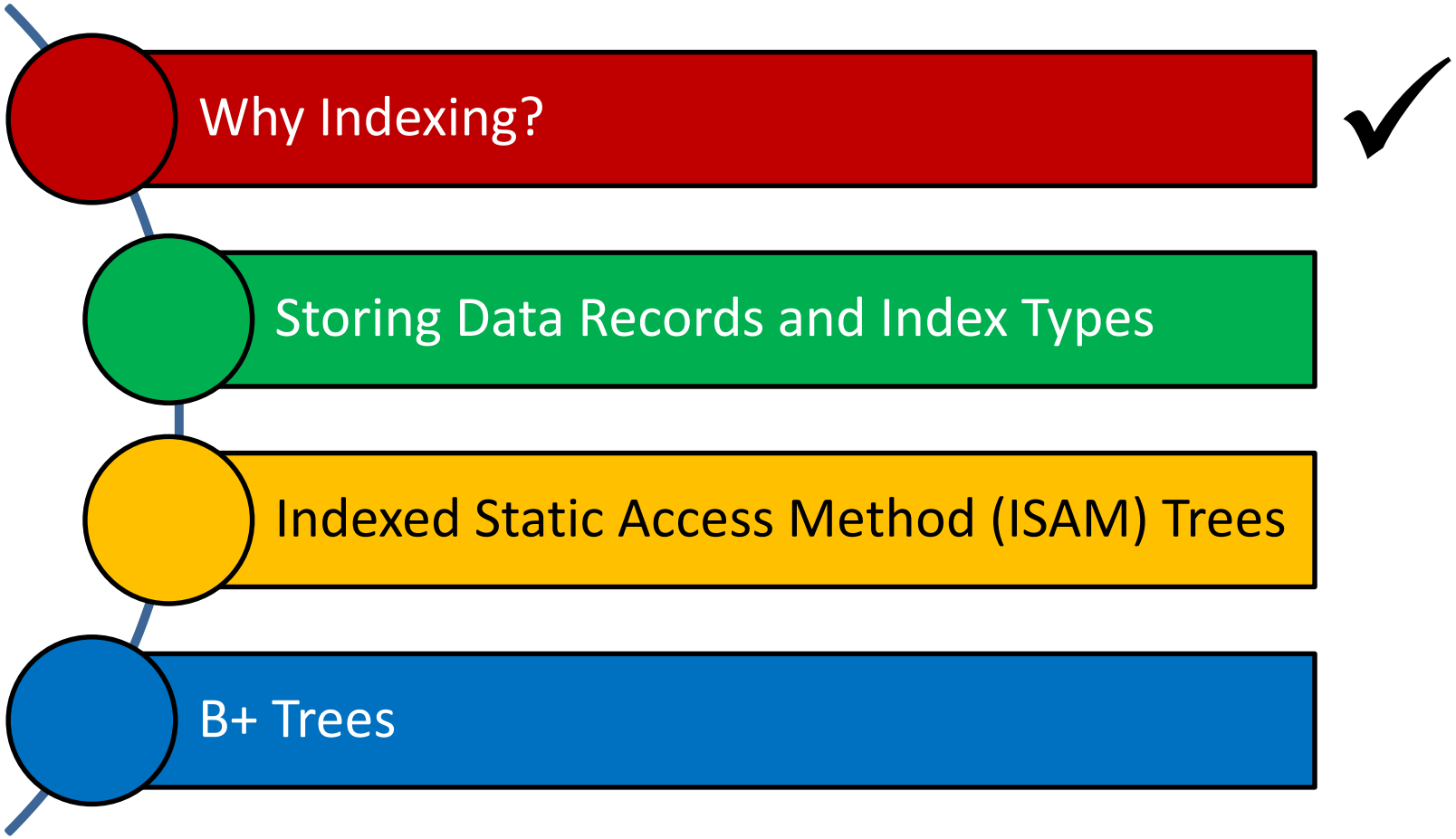
- Today's Session:
  - DBMS Internals- Part III
    - Tree-based indexes: ISAM and B+ trees

- Announcements:
  - PS2 grades are out
  - The midterm exam is on Tuesday Feb 24 (*all* materials are included)
  - Next week is off (Spring break)- classes will resume on March 8
  - P2 is due on March 15
  - PS3 will be out after the Spring break

DBMS Layers


Queries
Query Optimization and Execution
Relational Operators
Files and Access Methods
Buffer Management
Disk Space Management
Transaction Manager
Lock Manager
Recovery Manager
DB

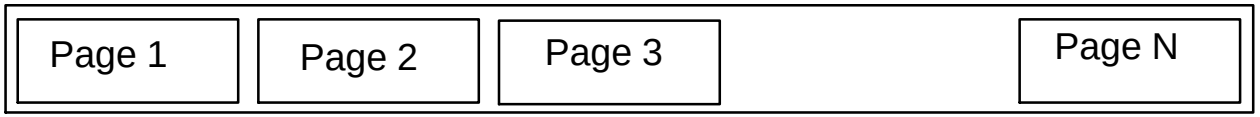
Outline



جامعة كارنيجي ميلون في قطر
Carnegie Mellon University Qatar

Motivation

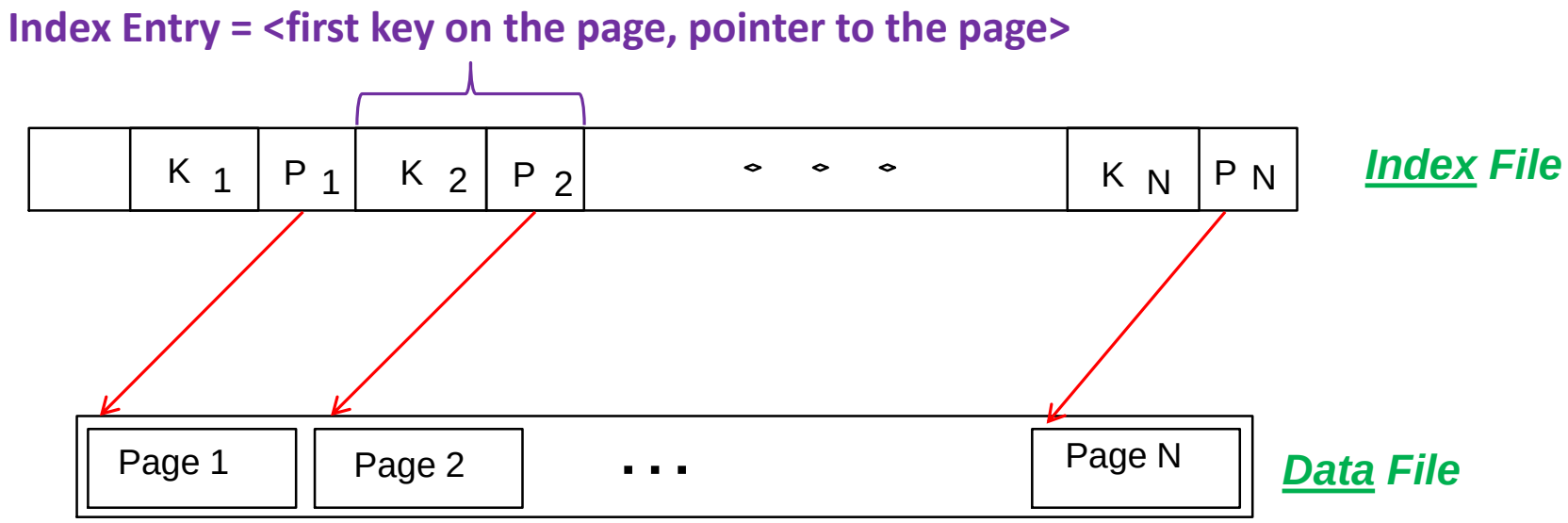
- Consider a file of student records *sorted* by GPA



- How can we answer a *range selection* (E.g., *"Find all students with a GPA higher than 3.0"*)?
  - What about doing a *binary search* followed by a *scan*?
    - Yes, but…
  - What if the file becomes "very" large?
    - Cost is proportional to the number of pages fetched
    - Hence, may become very slow!

Motivation

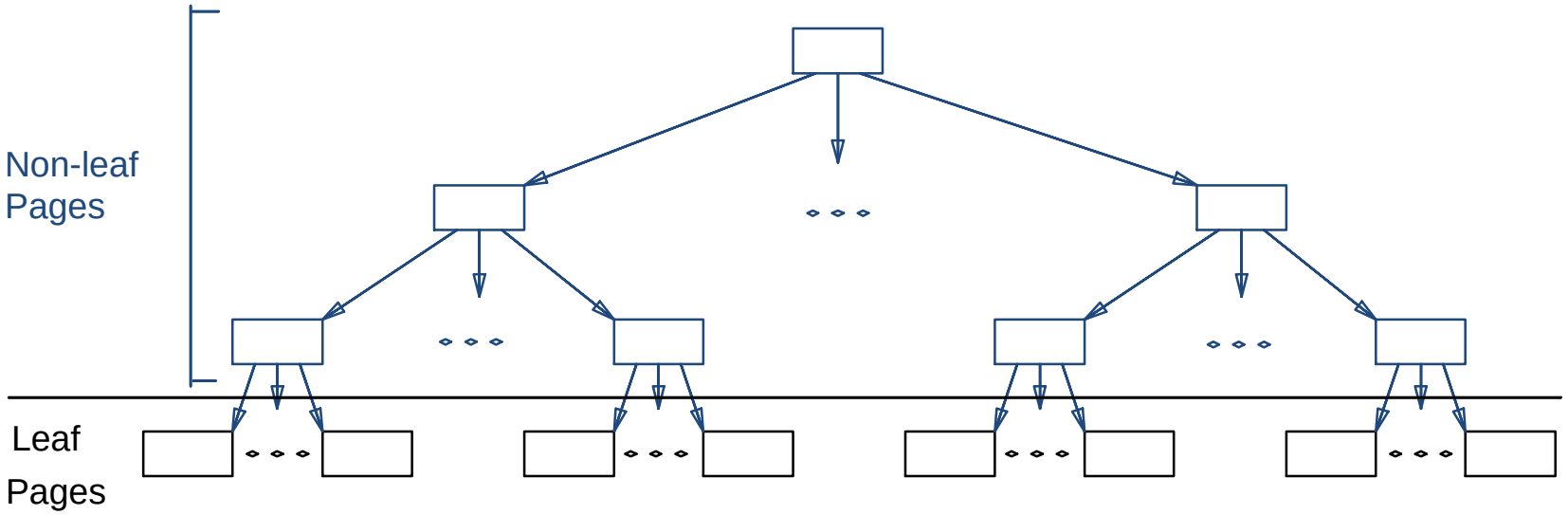
- What about creating an *index file* (with one entry per page) and do binary search there?



- But, what if the index file becomes also “very” large?

Motivation

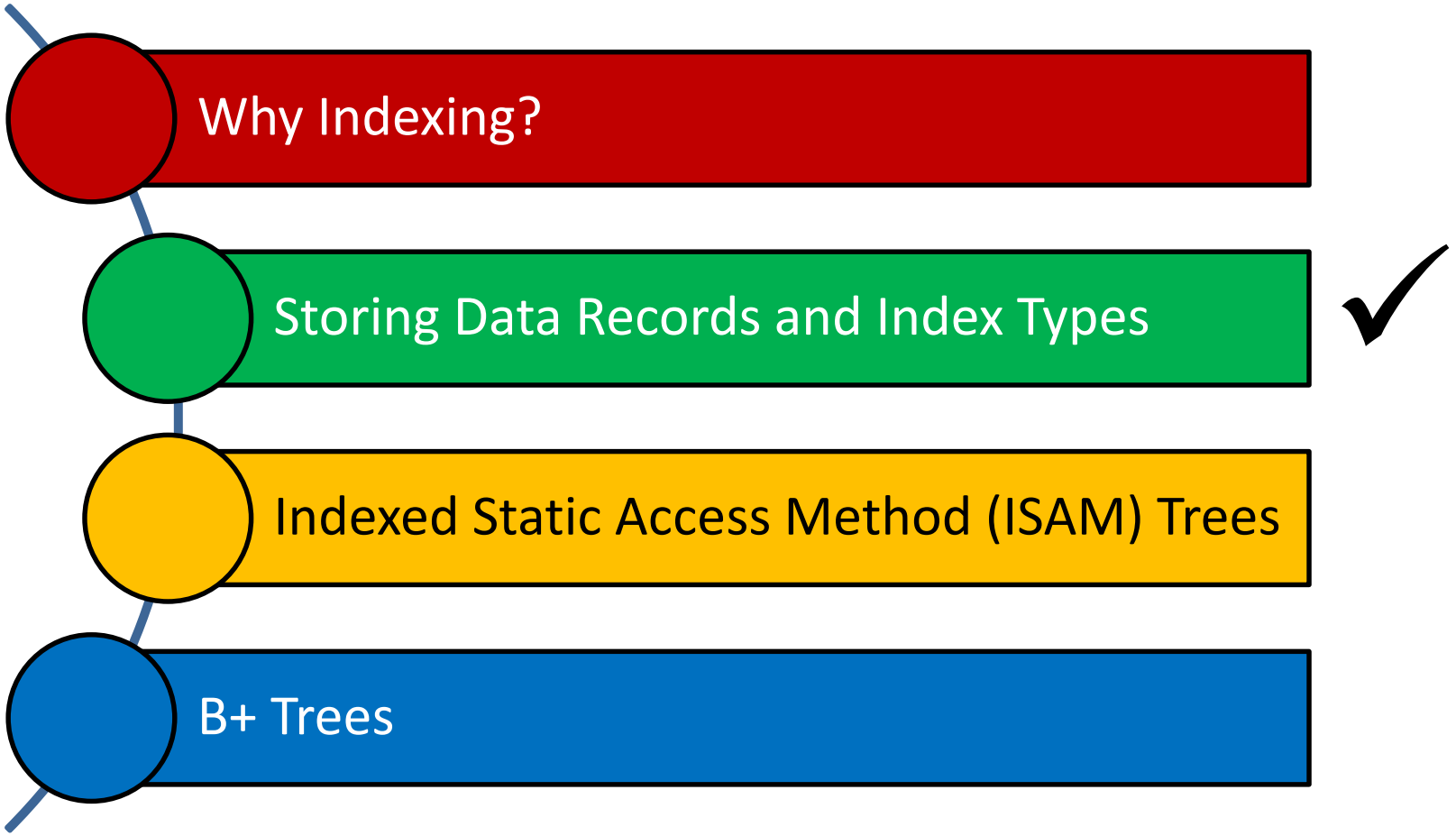
- Repeat recursively!



Each tree page is a disk block and all data records reside (*if chosen to be part of the index*) in ONLY leaf pages

How else data records can be stored?

Outline



جامعة كارنيجي ميلون في قطر
Carnegie Mellon University Qatar

Where to Store Data Records?

- In general, *3 alternatives* for "data records" (each referred to as ***K**)*** can be pursued:
  - Alternative (1): *K** is an actual data record with key ***k***
  - Alternative (2): *K** is a <***k***, **rid>** pair, where rid is the record id of a data record with search key ***k***
  - Alternative (3): *K** is a <***k***, rid-list> pair, where rid-list is a list of rids of data records with search key ***k***

Where to Store Data Records?

- In general, *3 alternatives* for "data records" (each referred to as ***K*)*** can be pursued:

Alternative (1): Leaf pages contain the actual data (i.e., the data records)

Alternative (2): Leaf pages contain the <key, rid> pairs and actual data records are stored in a separate file

Alternative (3): Leaf pages contain the <key, rid-list> pairs and actual data records are stored in a separate file

The choice among these alternatives is orthogonal to the *indexing technique*

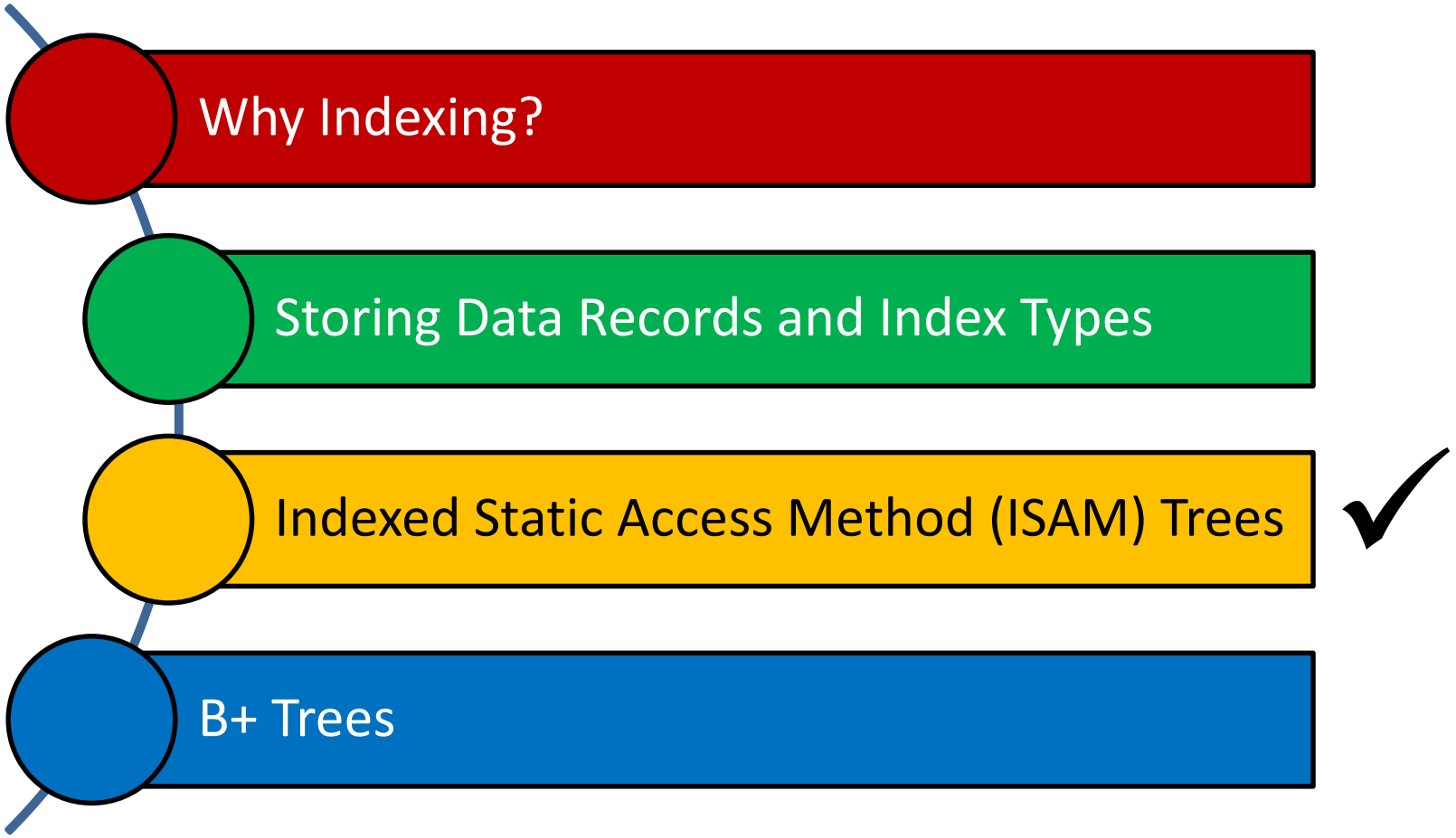
Clustered vs. Un-clustered Indexes

- Indexes can be either clustered or un-clustered
- Clustered Indexes:
  - When the ordering of data records is the same as (or close to) the ordering of entries in some index
- Un-clustered Indexes:
  - When the ordering of data records differs from the ordering of entries in some index

Clustered vs. Un-clustered Indexes

- Is an index that uses Alternative (1) clustered or un-clustered?
  - Clustered
- Is an index that uses Alternative (2) or (3) clustered or un-clustered?
  - Clustered “only” if data records are sorted on the search key field
- In practice:
  - A clustered index is an index that uses Alternative (1)
  - Indexes that use Alternatives (2) or (3) are un-clustered

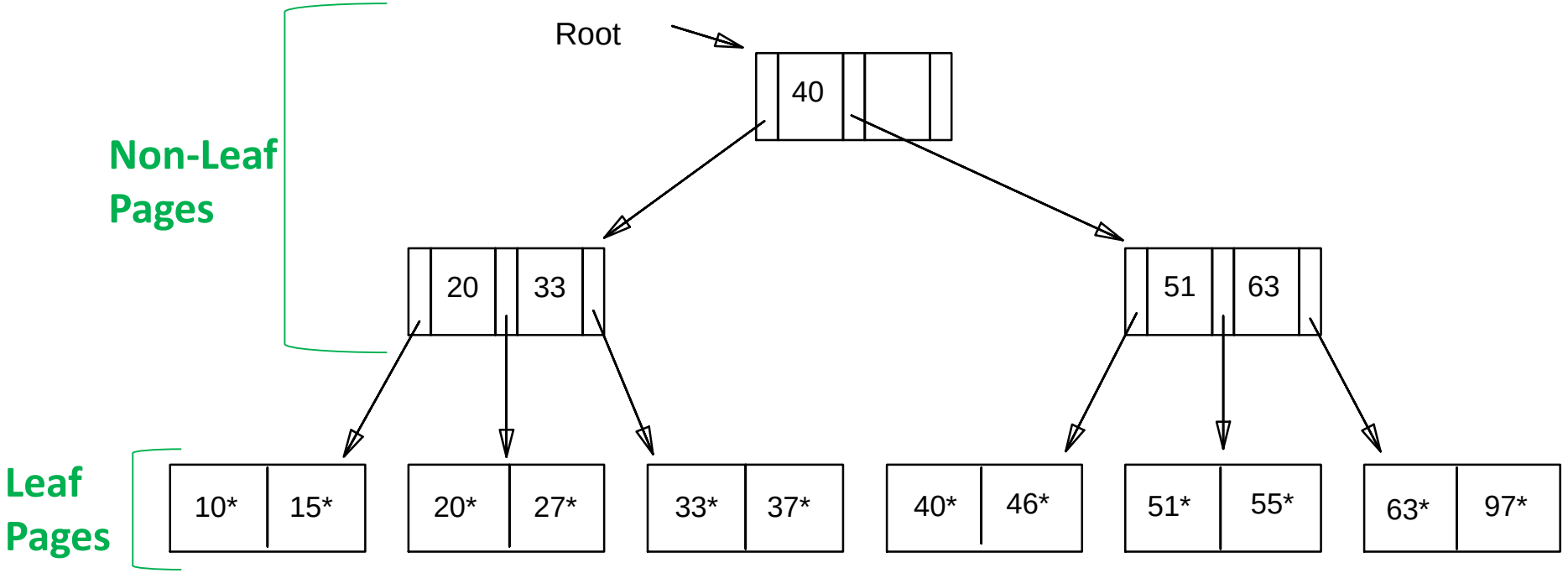
Outline



جامعة كارنيجي ميلون في قطر
Carnegie Mellon University Qatar

ISAM Trees

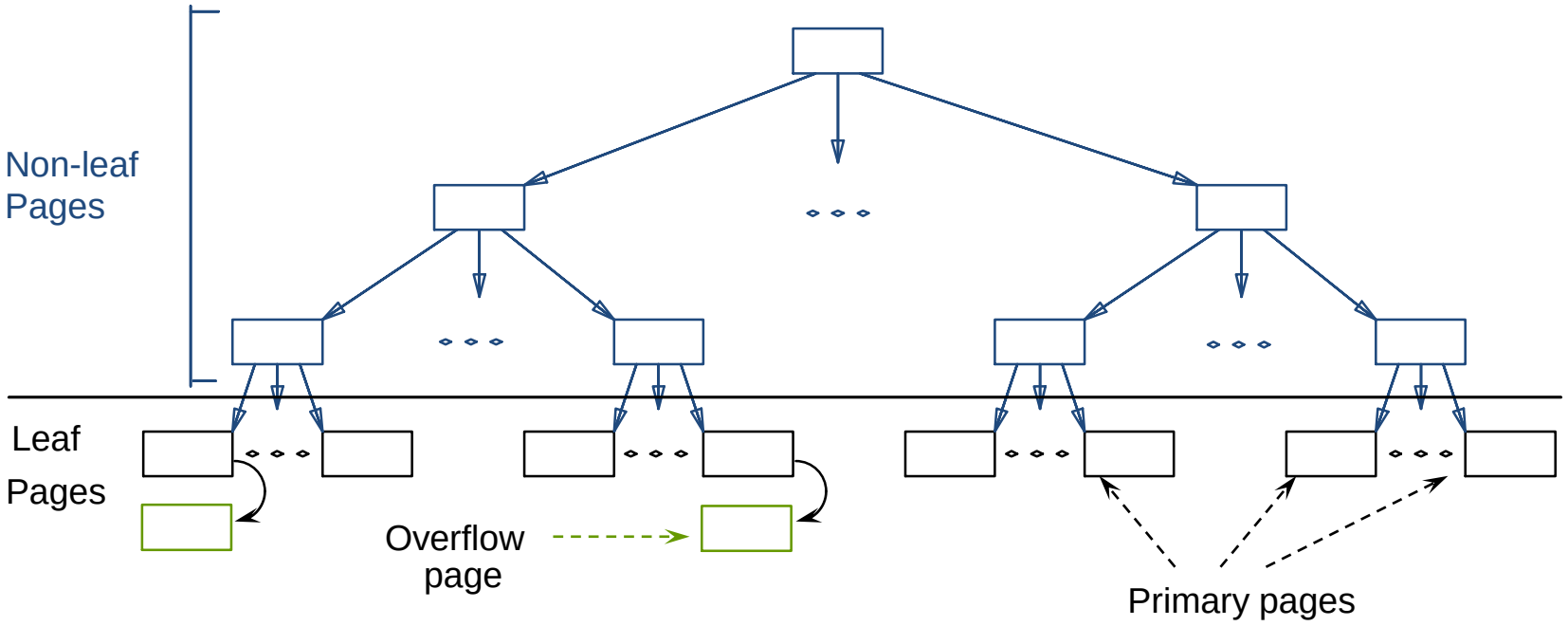
- Indexed Sequential Access Method (ISAM) trees are *static*



E.g., 2 Entries Per Page

ISAM Trees: Page Overflows

- What if there are a lot of insertions after creating the tree?



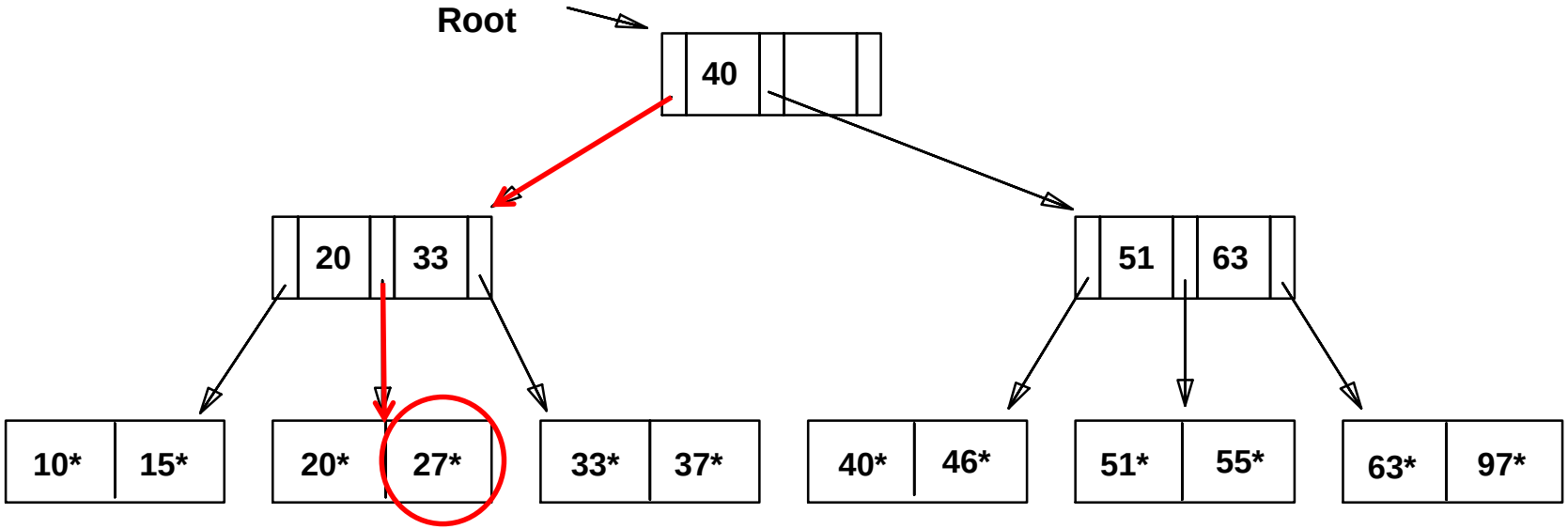
ISAM File Creation

- How to create an ISAM file?
  - All leaf pages are allocated *sequentially* and *sorted* on the search key value
  - If Alternative (2) or (3) is used, the data records are created and *sorted* before allocating leaf pages
  - The non-leaf pages are subsequently allocated

ISAM: Searching for Entries

- Search begins at root, and key comparisons direct it to a leaf

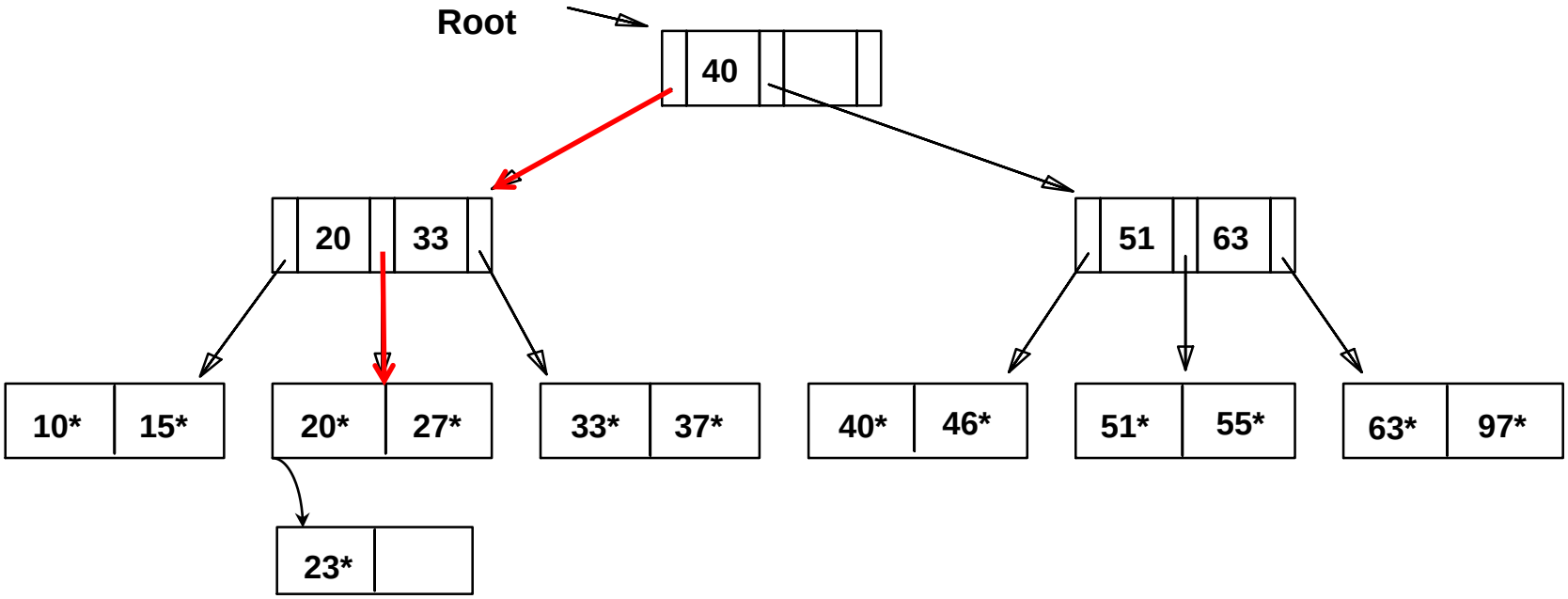
- Search for 27*



ISAM: Inserting Entries

- The appropriate page is determined as for a search, and the entry is inserted (with overflow pages added if necessary)

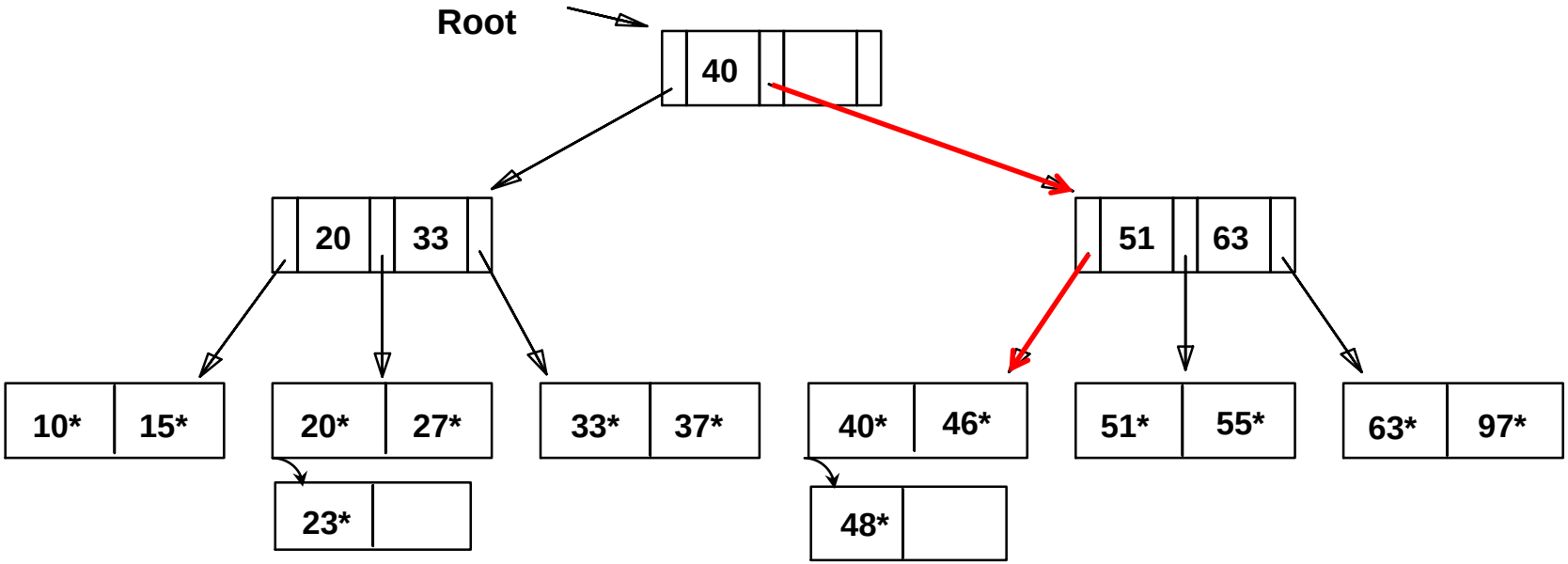
- Insert 23*



ISAM: Inserting Entries

- The appropriate page is determined as for a search, and the entry is inserted (with overflow pages added if necessary)

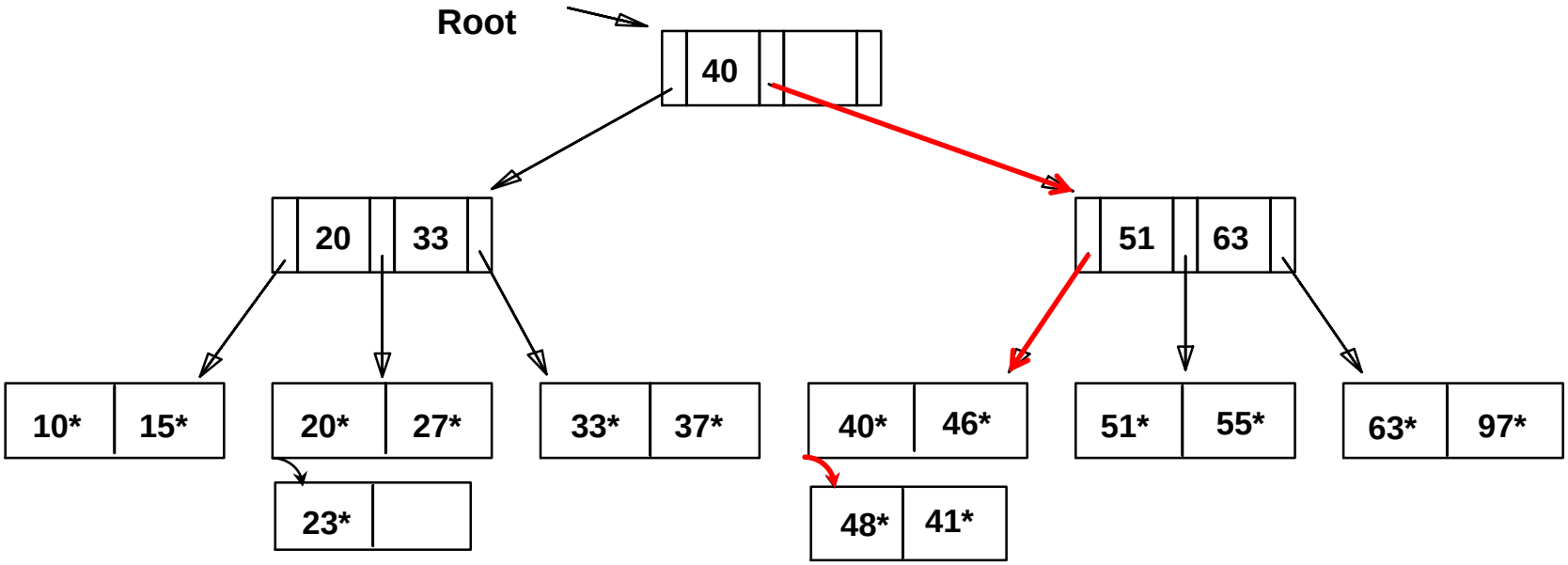
- Insert 48*



ISAM: Inserting Entries

- The appropriate page is determined as for a search, and the entry is inserted (with overflow pages added if necessary)

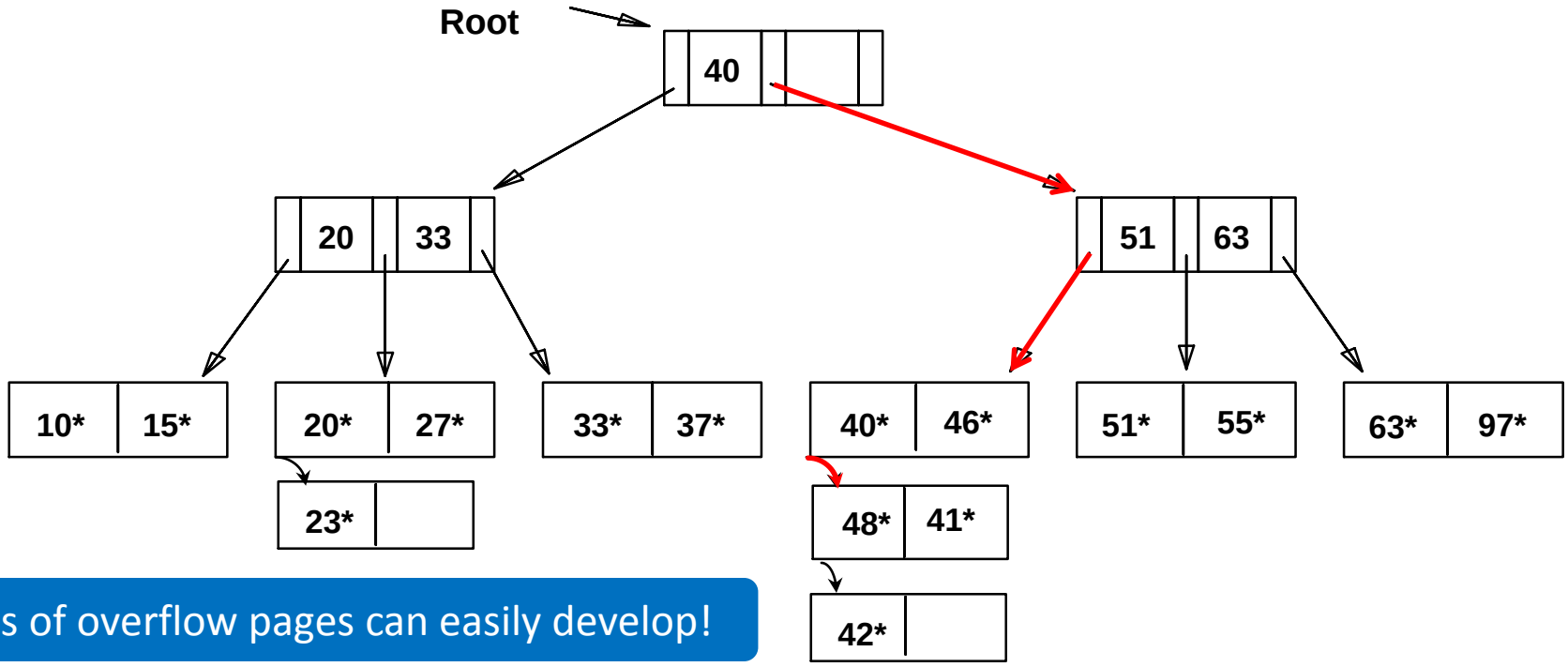
- Insert 41*



ISAM: Inserting Entries

- The appropriate page is determined as for a search, and the entry is inserted (with overflow pages added if necessary)

- Insert 42*

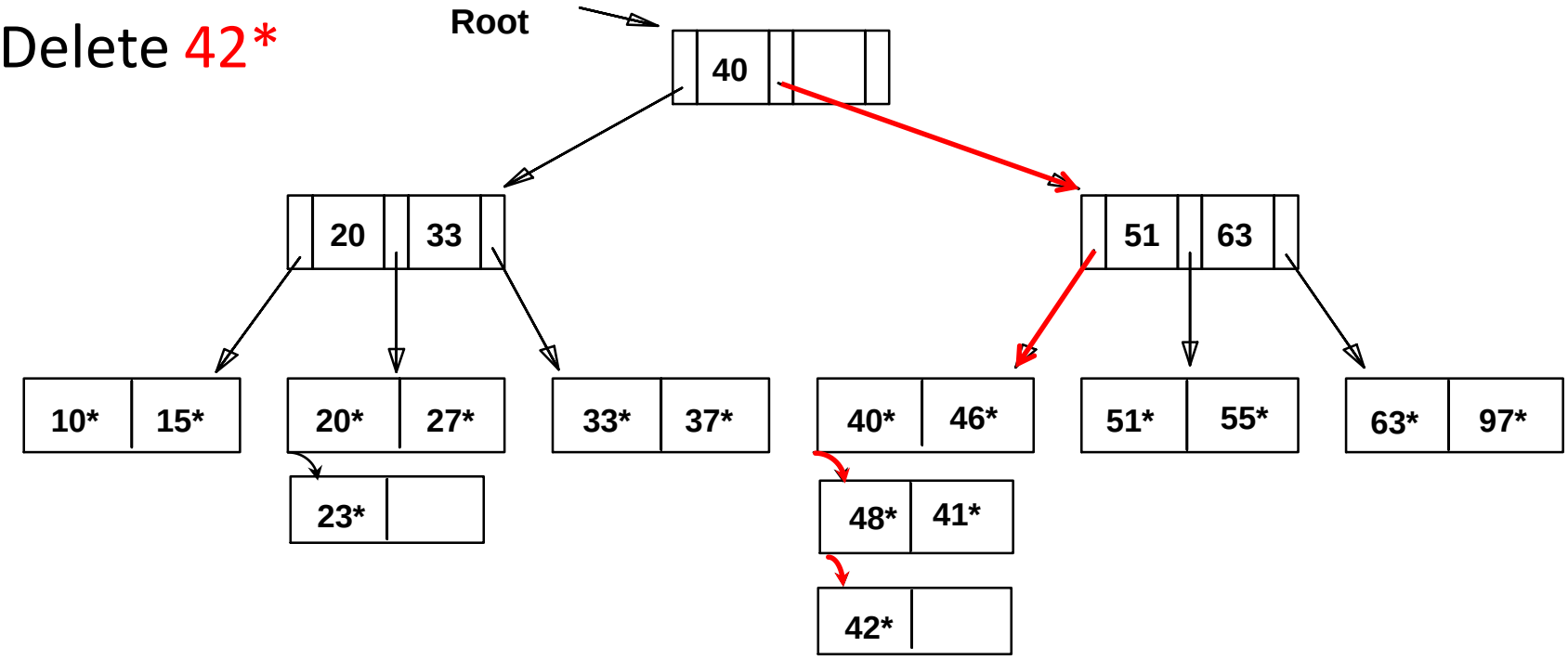


Chains of overflow pages can easily develop!

ISAM: Deleting Entries

- The appropriate page is determined as for a search, and the entry is deleted (*with ONLY overflow pages removed when becoming empty*)

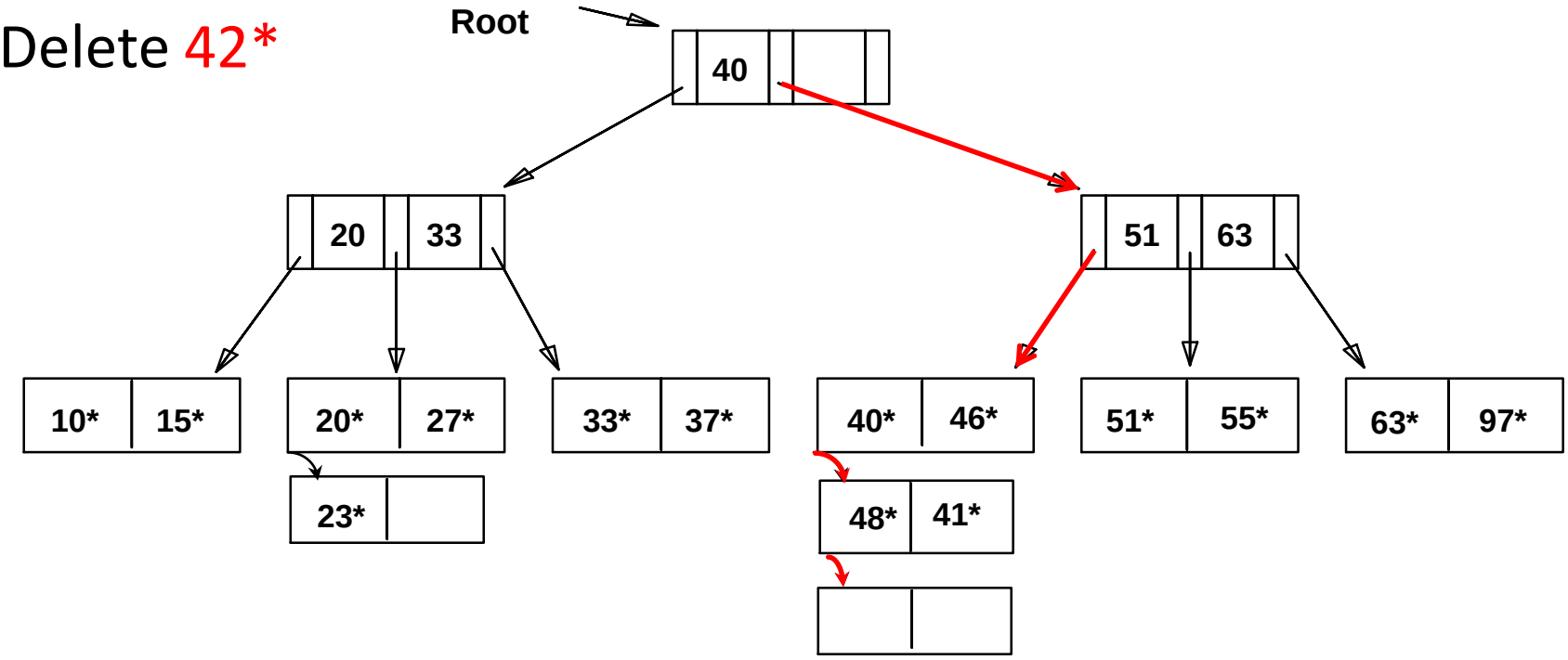
- Delete 42*



ISAM: Deleting Entries

- The appropriate page is determined as for a search, and the entry is deleted (*with ONLY overflow pages removed when becoming empty*)

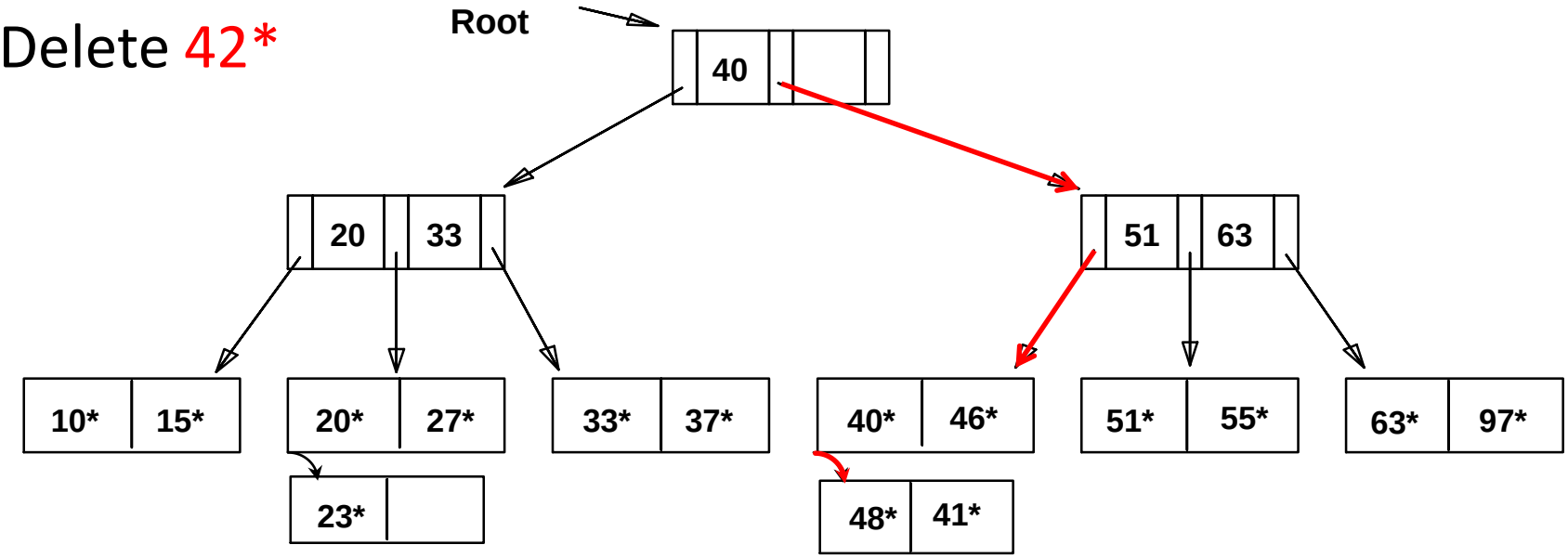
- Delete 42*



ISAM: Deleting Entries

- The appropriate page is determined as for a search, and the entry is deleted (*with ONLY overflow pages removed when becoming empty*)

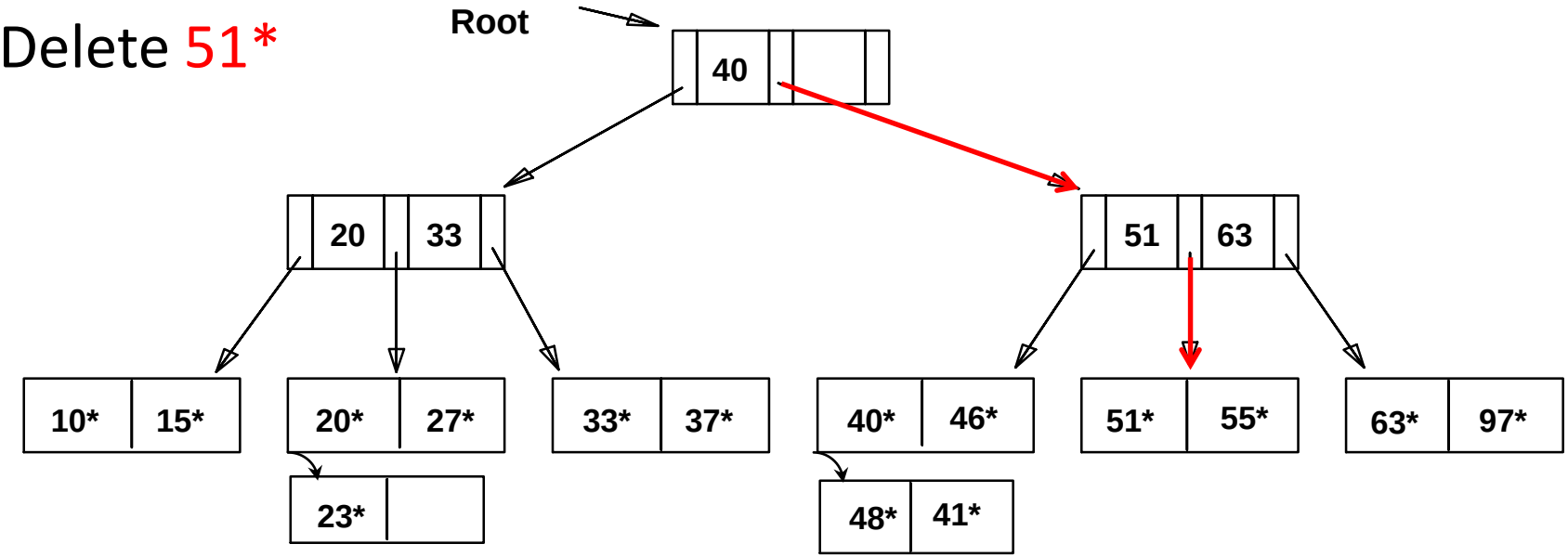
- Delete 42*



ISAM: Deleting Entries

- The appropriate page is determined as for a search, and the entry is deleted (*with ONLY overflow pages removed when becoming empty*)

- Delete 51*

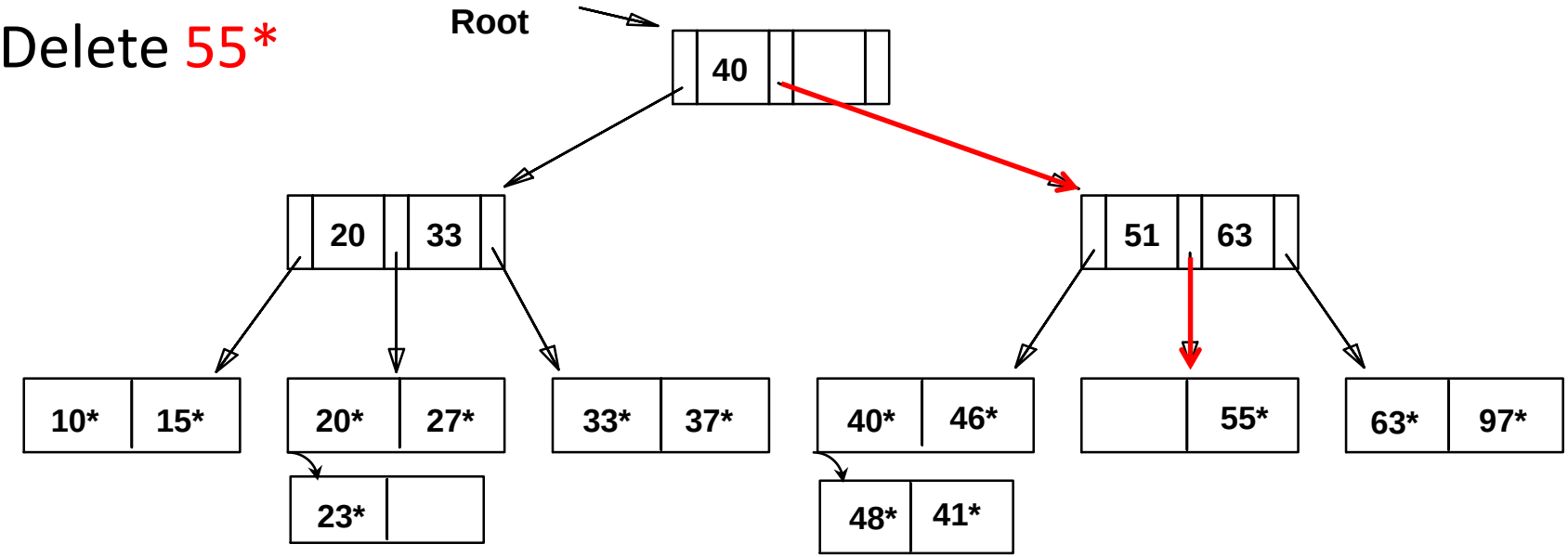


Note that 51 still appears in an index entry, but not in the leaf!

ISAM: Deleting Entries

- The appropriate page is determined as for a search, and the entry is deleted (*with ONLY overflow pages removed when becoming empty*)

- Delete 55*



Primary pages are NOT removed, even if they become empty!

ISAM: Some Issues

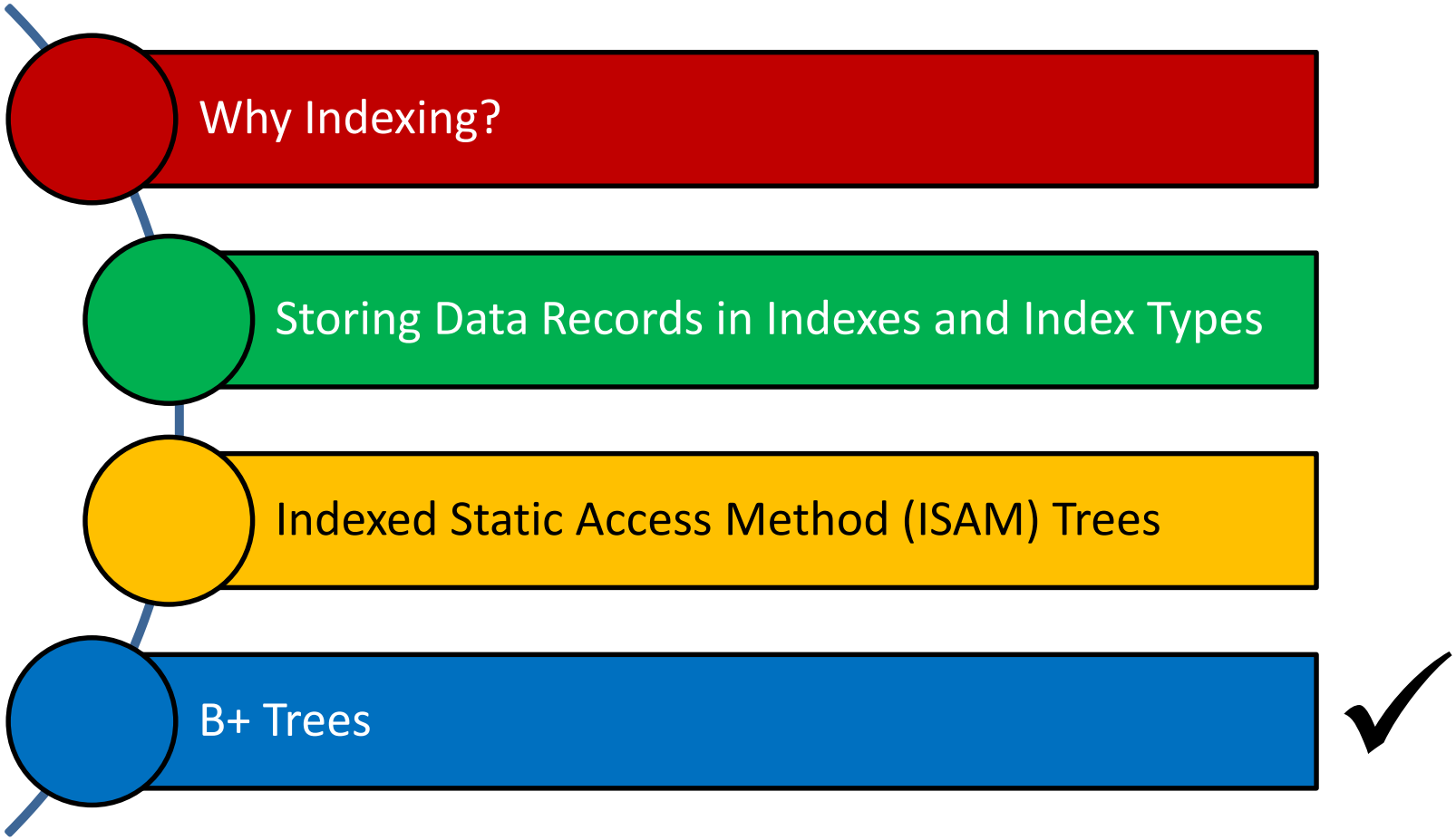
- Once an ISAM file is created, insertions and deletions affect only the contents of leaf pages (i.e., *ISAM is a static structure*!)

- Since index-level pages are *never* modified, there is no need to *lock* them during insertions/deletions
  - Critical for concurrency!

- Long overflow chains can develop easily
  - The tree can be initially set so that ~20% of each page is free

- If the data distribution and size are relatively static, ISAM might be a good choice to pursue!

Outline



جامعة كارنيجي ميلون في قطر
Carnegie Mellon University Qatar

Dynamic Trees

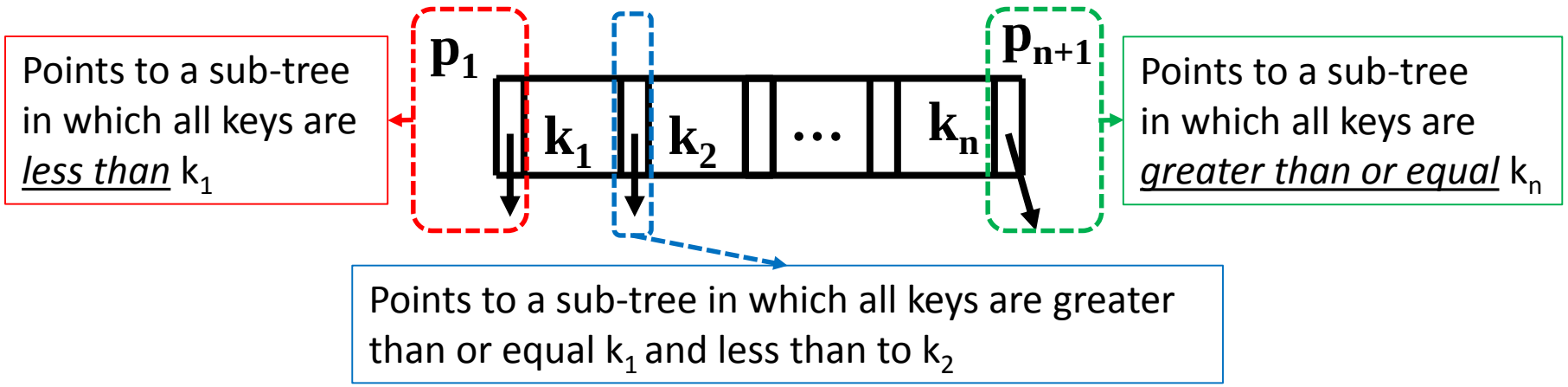
- ISAM indices are static
  - Long overflow chains can develop as the file grows, leading to poor performance

- This calls for more flexible, *dynamic* indices that adjust gracefully to insertions and deletions
  - No need to allocate the leaf pages sequentially as in ISAM

- Among the **most successful** dynamic index schemes is the B+ tree

B+ Tree Properties

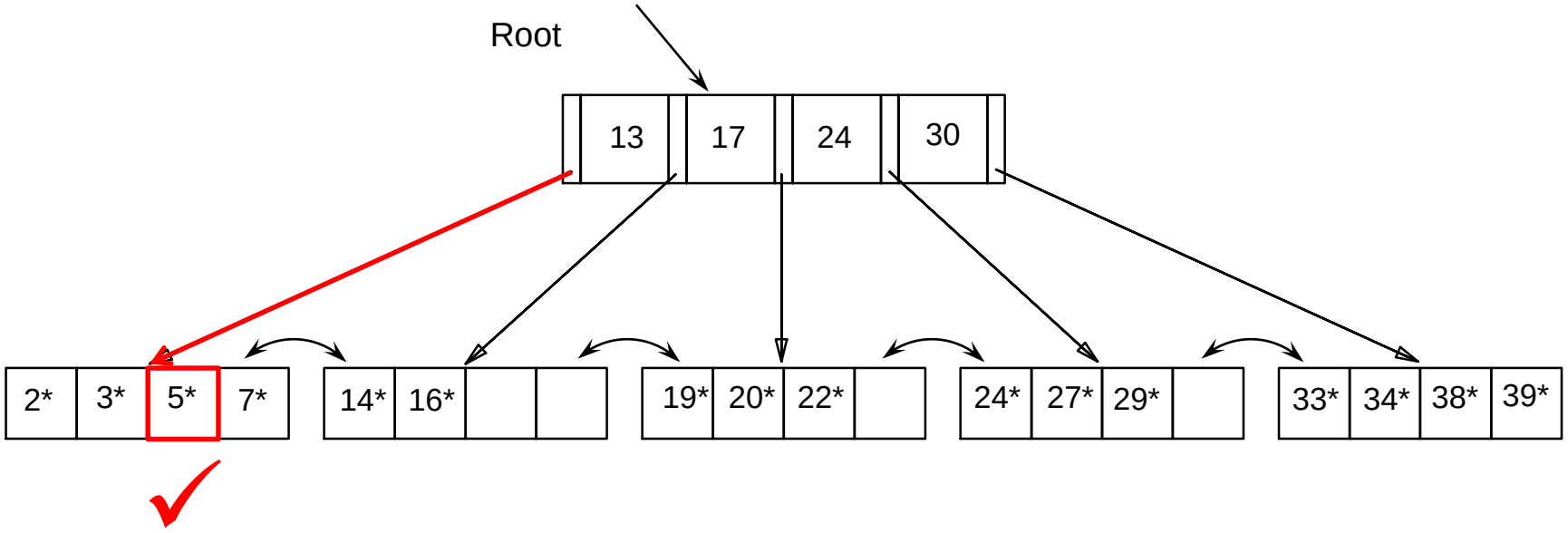
- Each node in a B+ tree of order ***d*** (this is a measure of the capacity of a tree):
  - Has at most ***2d*** keys
  - Has at least ***d*** keys (except the root, which may have just 1 key)
  - All leaves are on the same level
  - Has exactly ***n-1*** keys if the number of pointers is ***n***



B+ Tree: Searching for Entries

- Search begins at root, and key comparisons direct it to a leaf (as in ISAM)

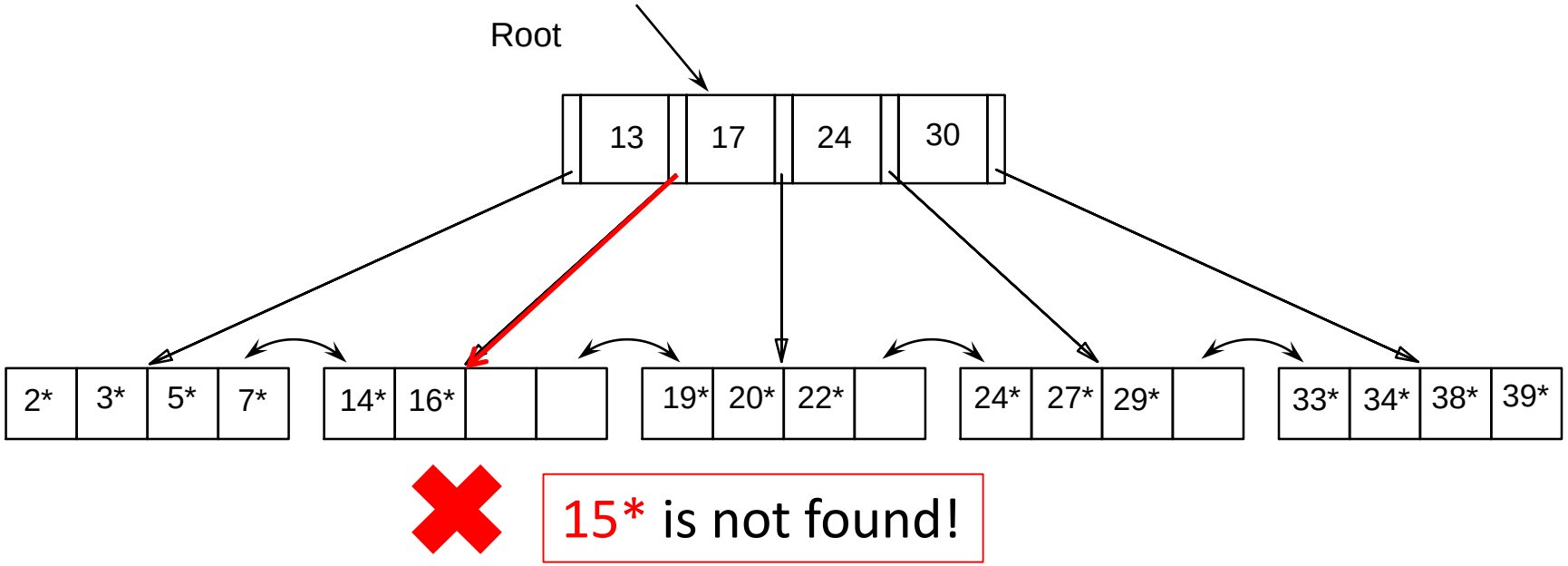
- Example 1: Search for entry 5*



B+ Tree: Searching for Entries

- Search begins at root, and key comparisons direct it to a leaf (as in ISAM)

- Example 2: Search for entry 15*

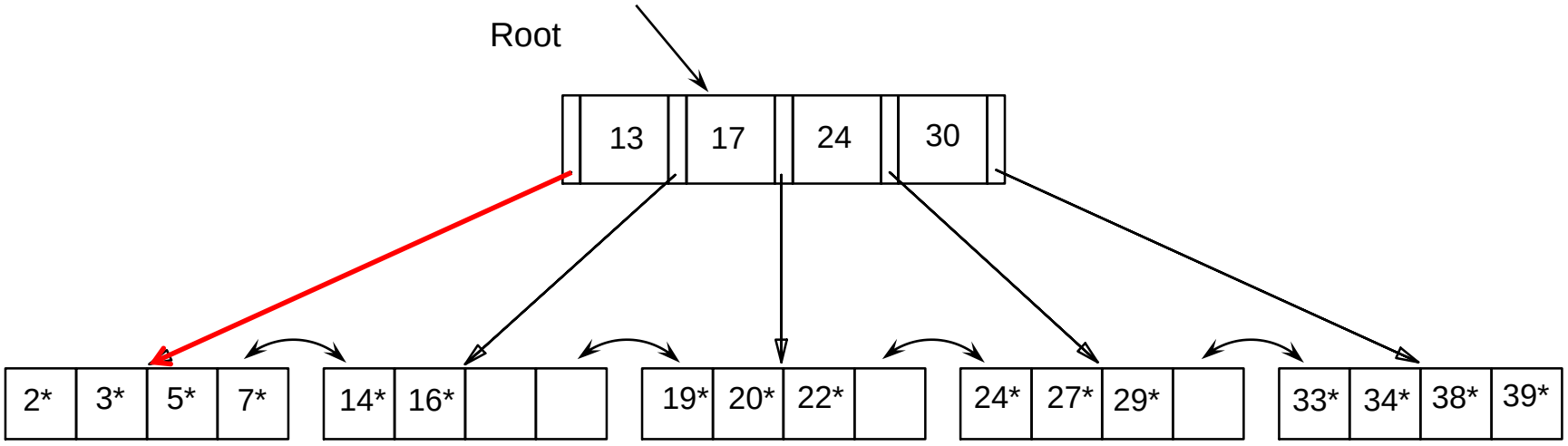


B+ Trees: Inserting Entries

- Find correct leaf ***L***
- Put data entry onto ***L***
  - If ***L*** has enough space, *done*!
  - Else, *split* ***L*** *into* ***L*** *and a new node* L_2
    - Re-partition entries *evenly*, *copying up* the middle key
- Parent node may *overflow*
  - *Push up* middle key (splits "grow" trees; a root split increases the height of the tree)

B+ Tree: Examples of Insertions

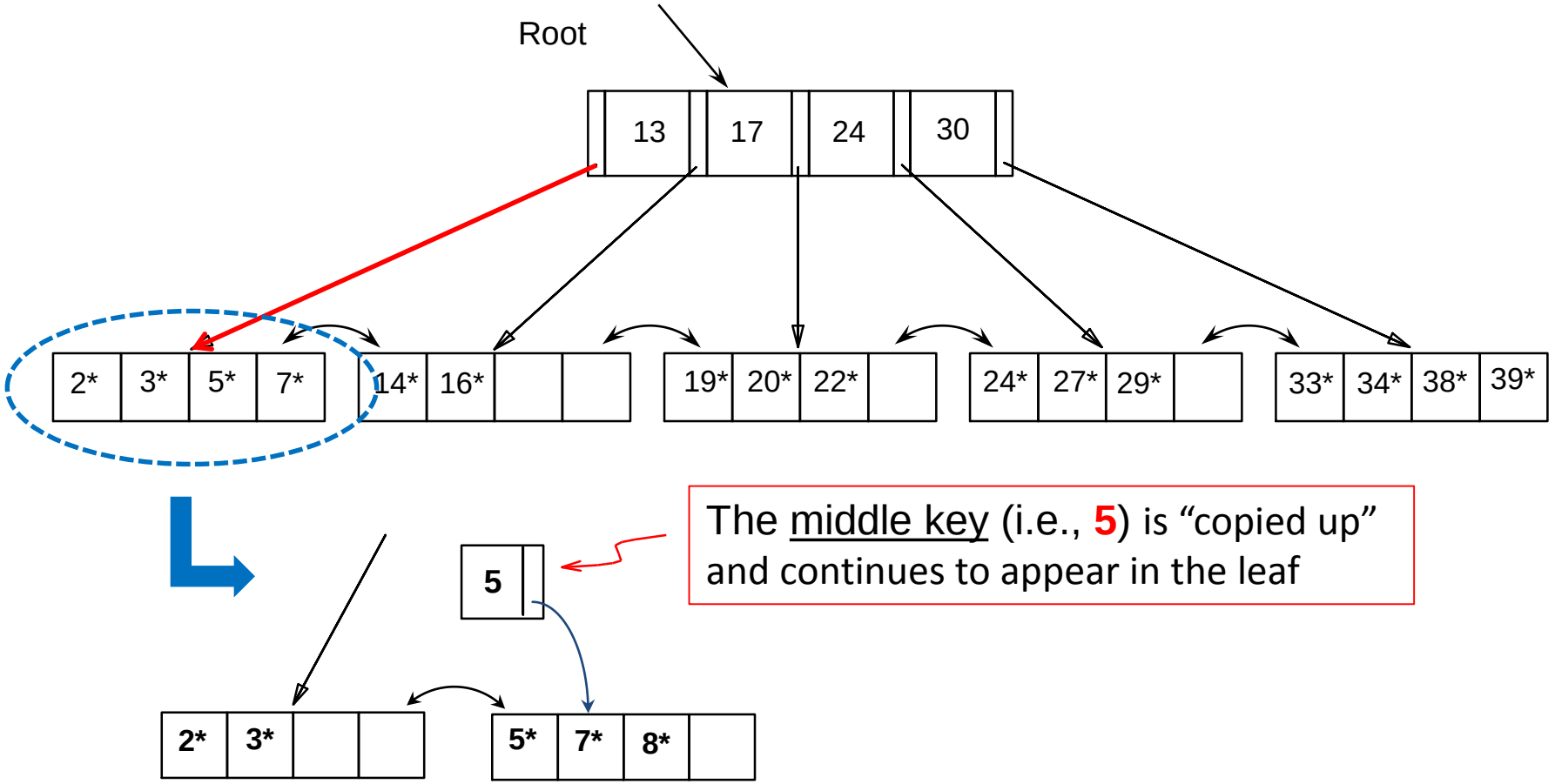
- Insert entry 8*



Leaf is ***full***; hence, split!

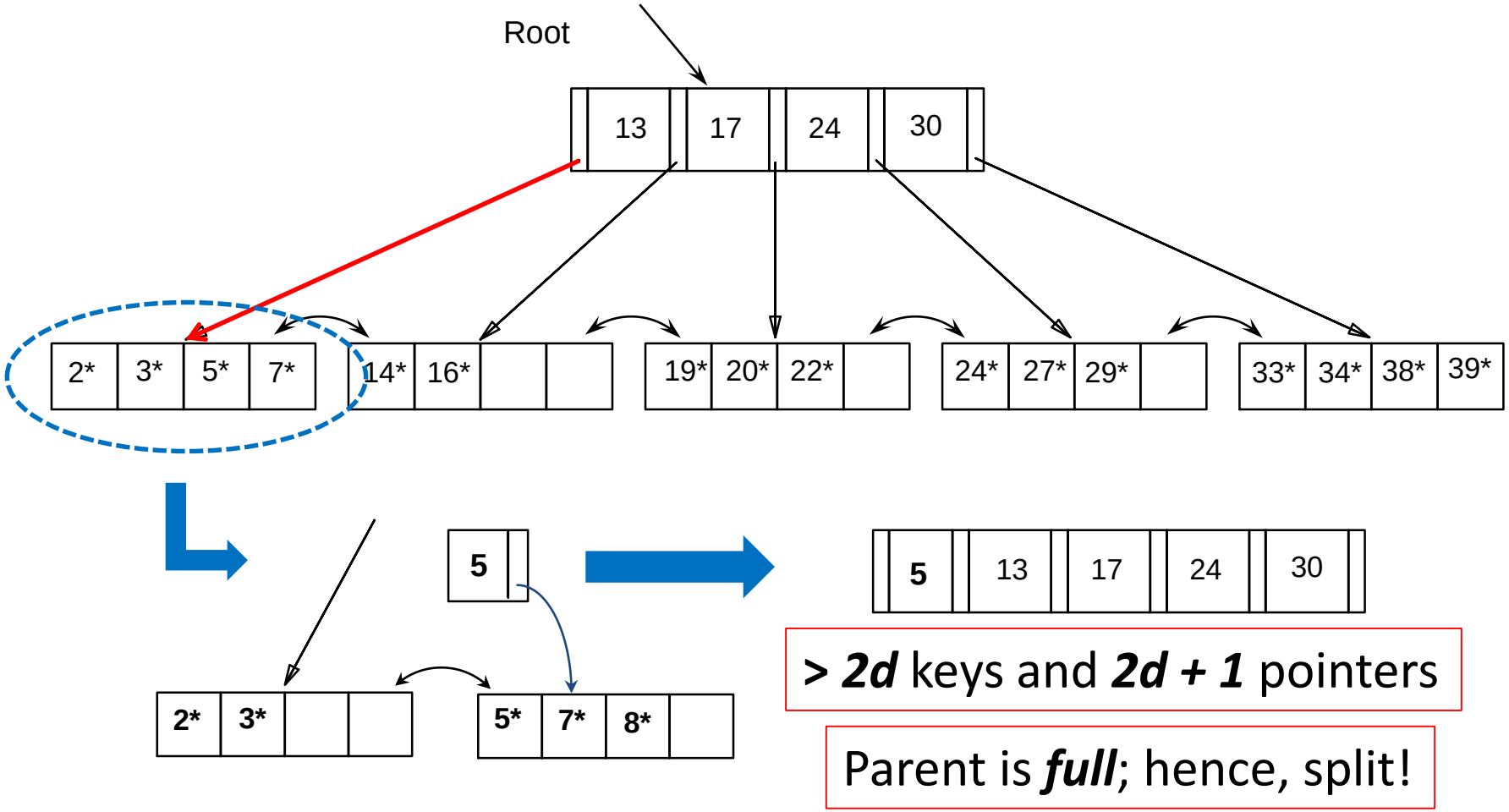
B+ Tree: Examples of Insertions

- Insert entry 8*



B+ Tree: Examples of Insertions

- Insert entry 8*

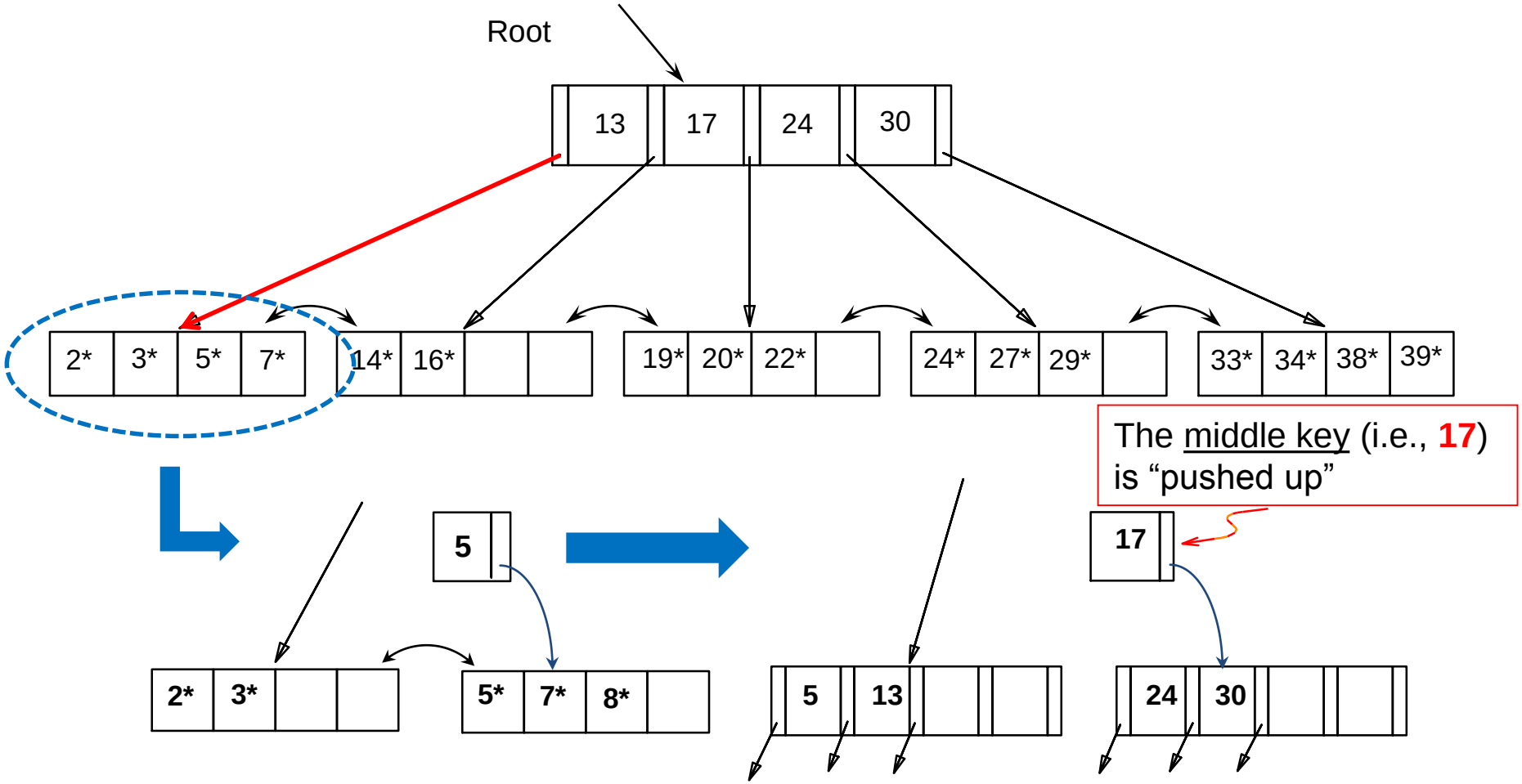


> ***2d*** keys and ***2d + 1*** pointers

Parent is ***full***; hence, split!

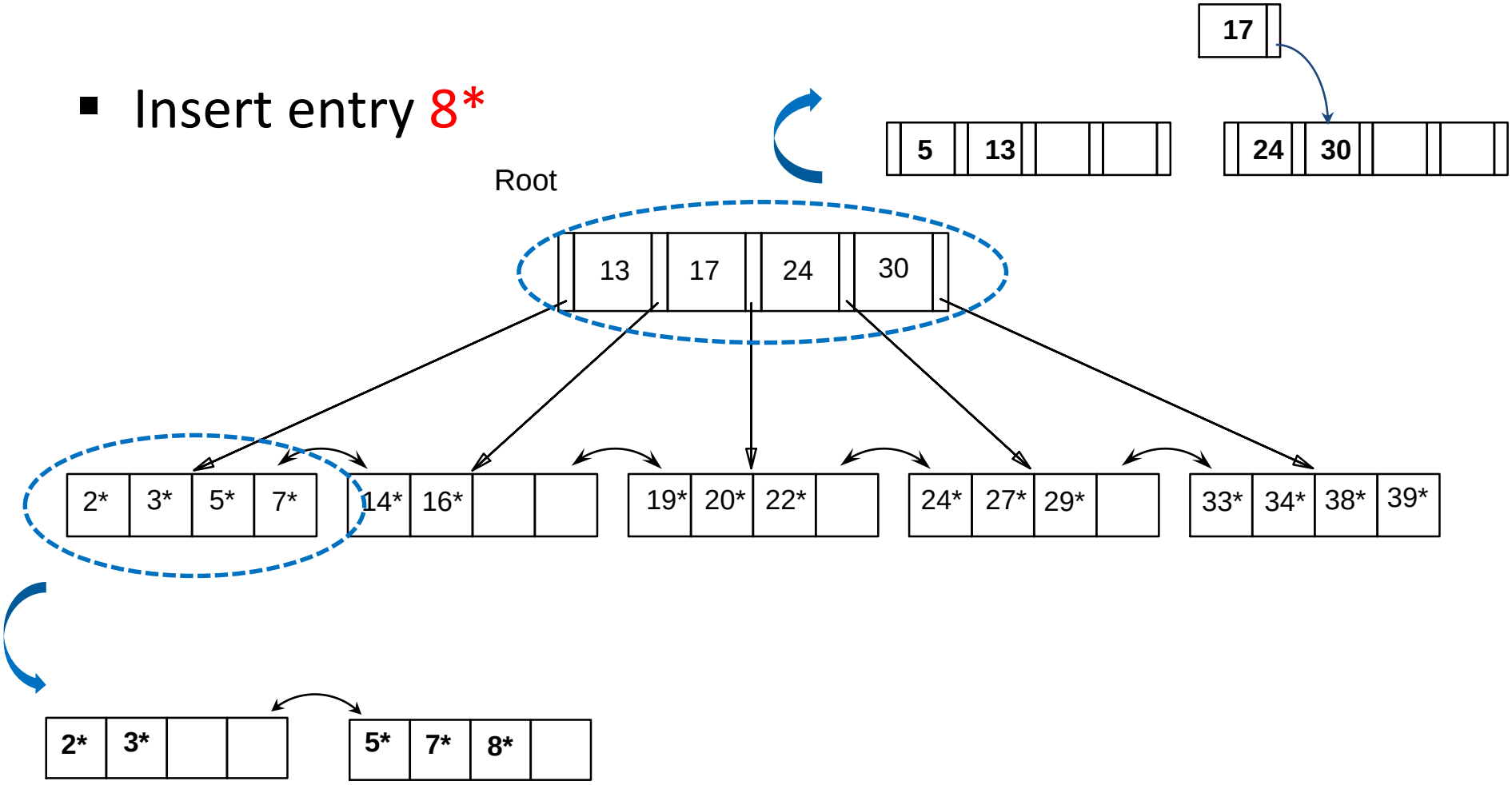
B+ Tree: Examples of Insertions

- Insert entry 8*



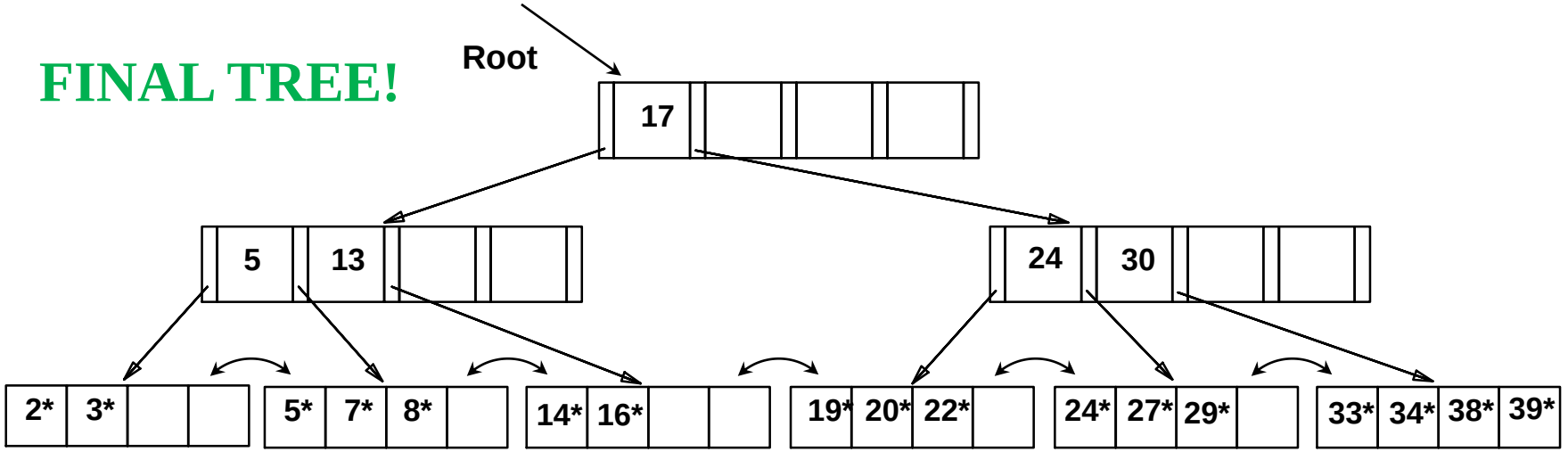
B+ Tree: Examples of Insertions

- Insert entry 8*



B+ Tree: Examples of Insertions

- Insert entry 8*

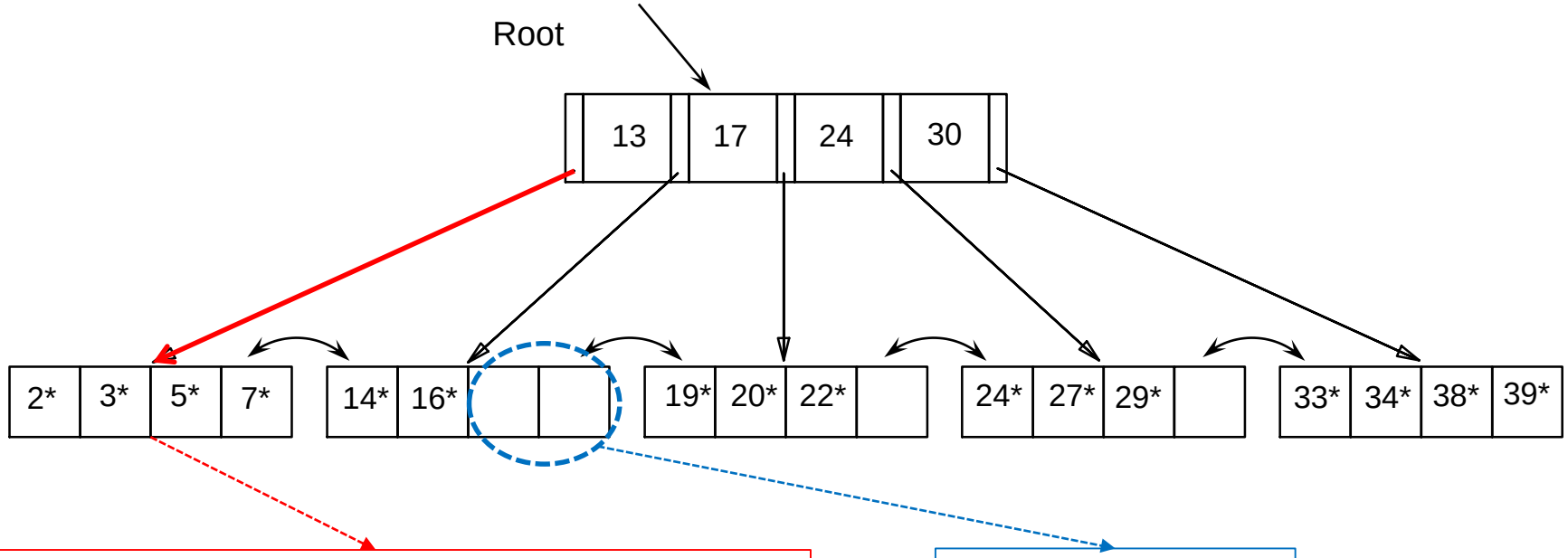


Splitting the root lead to an increase of height by 1!

What about *re-distributing* entries instead of *splitting* nodes?

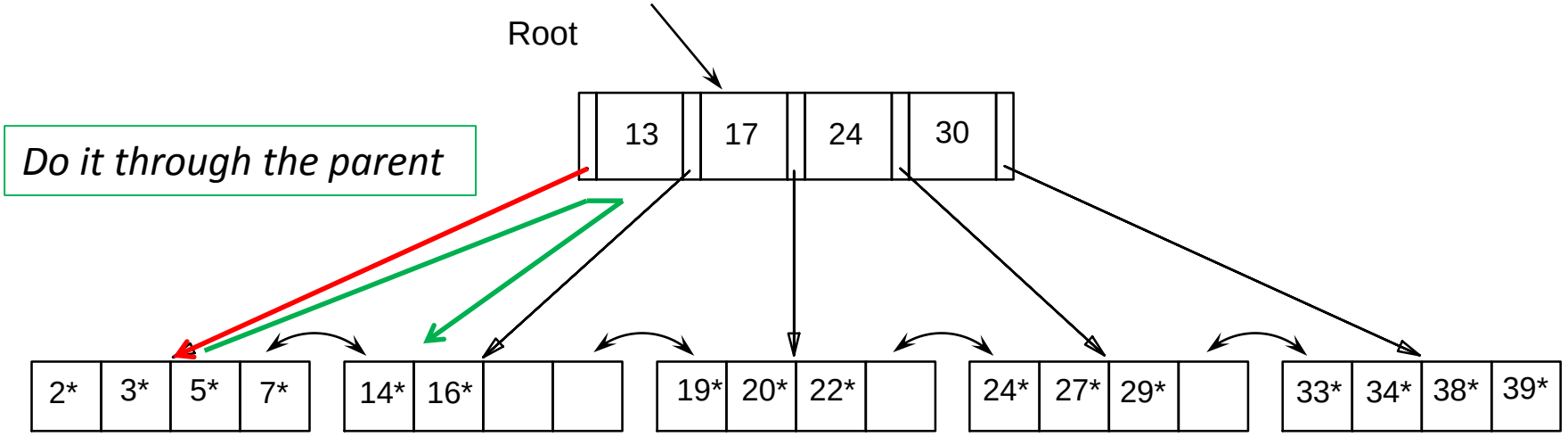
B+ Tree: Examples of Insertions

- Insert entry 8*



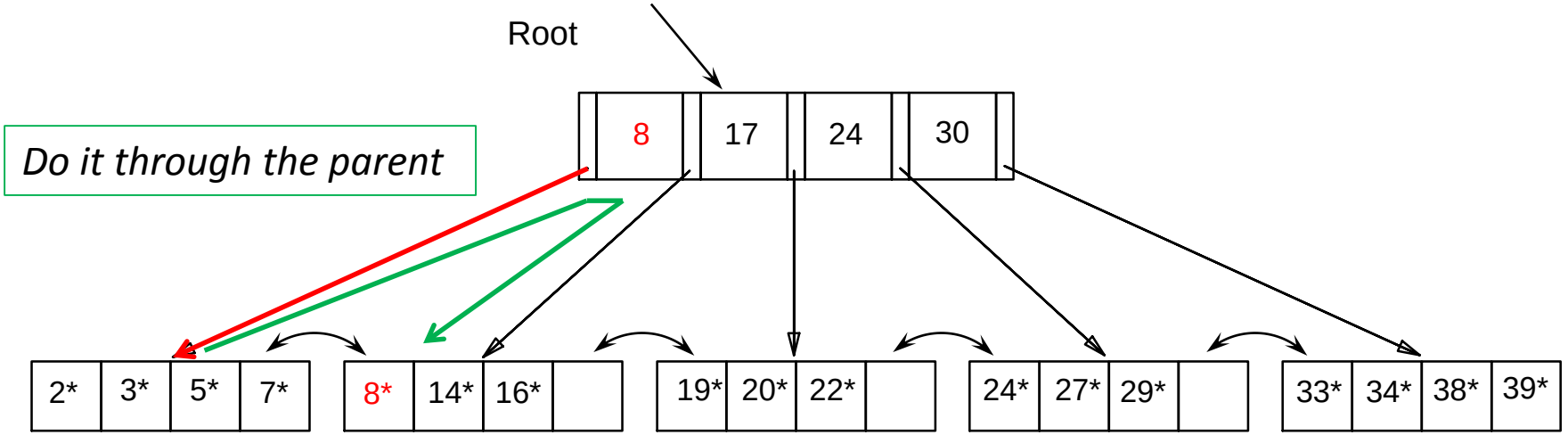
B+ Tree: Examples of Insertions

- Insert entry 8*



B+ Tree: Examples of Insertions

- Insert entry 8*



But, when to *redistribute* and when to *split*?

Splitting vs. Redistributing

- Leaf Nodes
  - Previous and next-neighbor pointers must be updated upon insertions (*if splitting is to be pursued*)
  - Hence, checking whether redistribution is possible does not increase I/O
  - Therefore, if a sibling can spare an entry, re-distribute
- Non-Leaf Nodes
  - Checking whether redistribution is possible *usually* increases I/O
  - Splitting non-leaf nodes typically pays off!

B+ Insertions: Keep in Mind

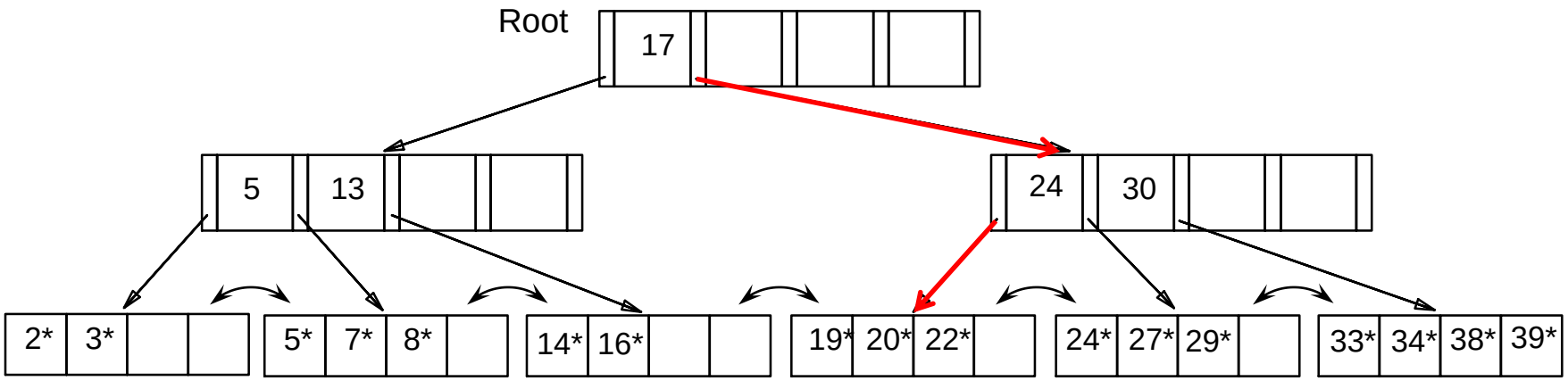
- Every data entry must appear in a leaf node; hence, “copy up” the middle key upon splitting
- When splitting index entries, simply “push up” the middle key
- Apply splitting and/or redistribution on leaf nodes
- Apply only splitting on non-leaf nodes

B+ Trees: Deleting Entries

- Start at root, find leaf ***L*** where entry belongs
- Remove the entry
  - If ***L*** is at least half-full, *done!*
  - If ***L*** *underflows*
    - Try to re-distribute (i.e., borrow from a "rich sibling" and "copy up" its *lowest key*)
    - If re-distribution fails, *merge* ***L*** and a "poor sibling"
      - Update parent
      - And possibly merge, recursively

B+ Tree: Examples of Deletions

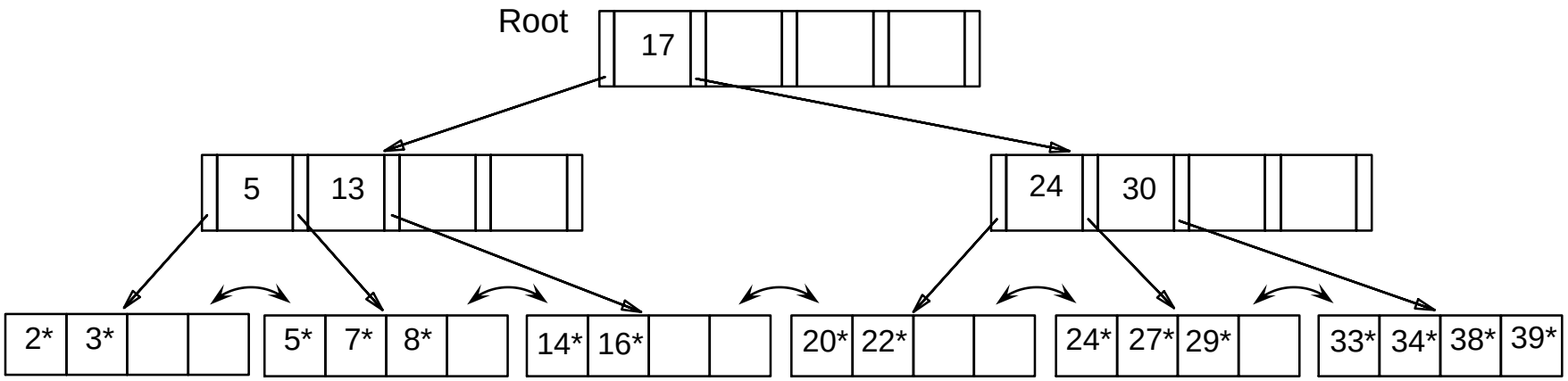
- Delete 19*



Removing 19* does not cause an underflow

B+ Tree: Examples of Deletions

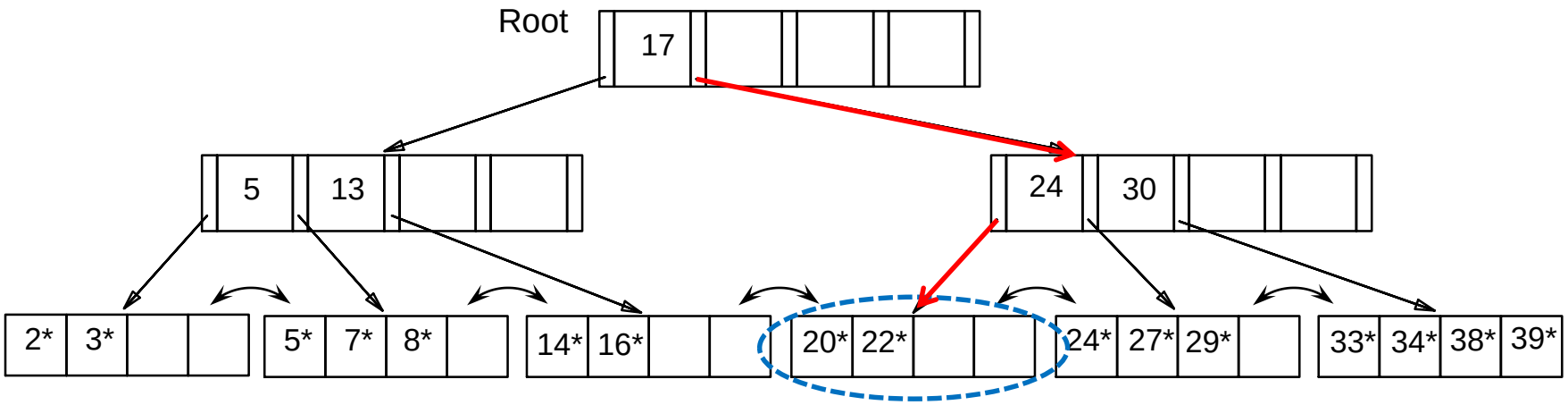
- Delete 19*



FINAL TREE!

B+ Tree: Examples of Deletions

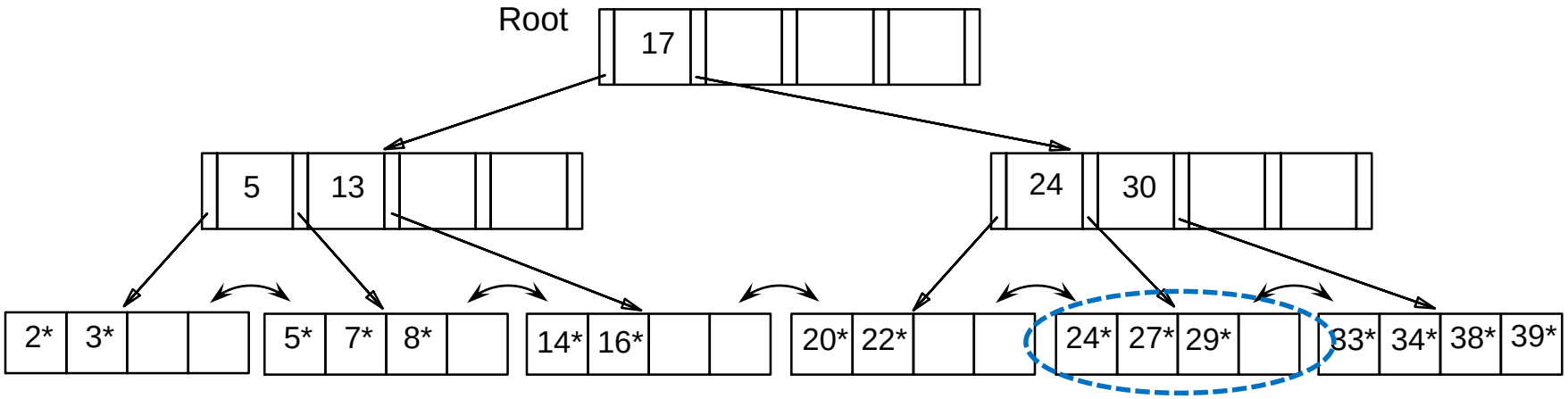
- Delete 20*



Deleting 20* causes an underflow; hence, check a sibling for redistribution

B+ Tree: Examples of Deletions

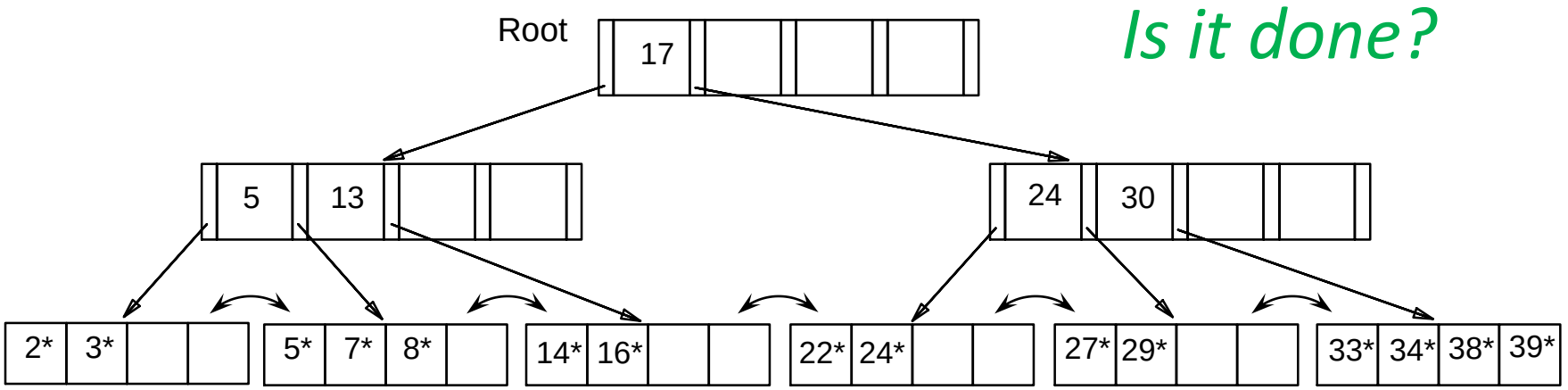
- Delete 20*



The sibling is ‘rich’ (i.e., can lend an entry); hence, remove 20* and redistribute!

B+ Tree: Examples of Deletions

- Delete 20*

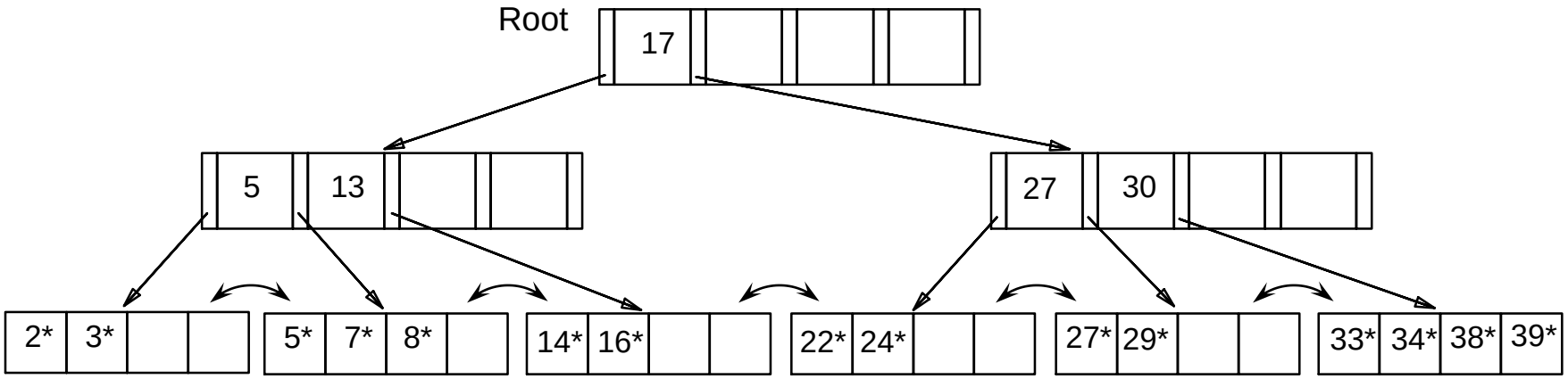


Is it done?

"Copy up" 27*, the lowest value in the leaf from which we borrowed 24*

B+ Tree: Examples of Deletions

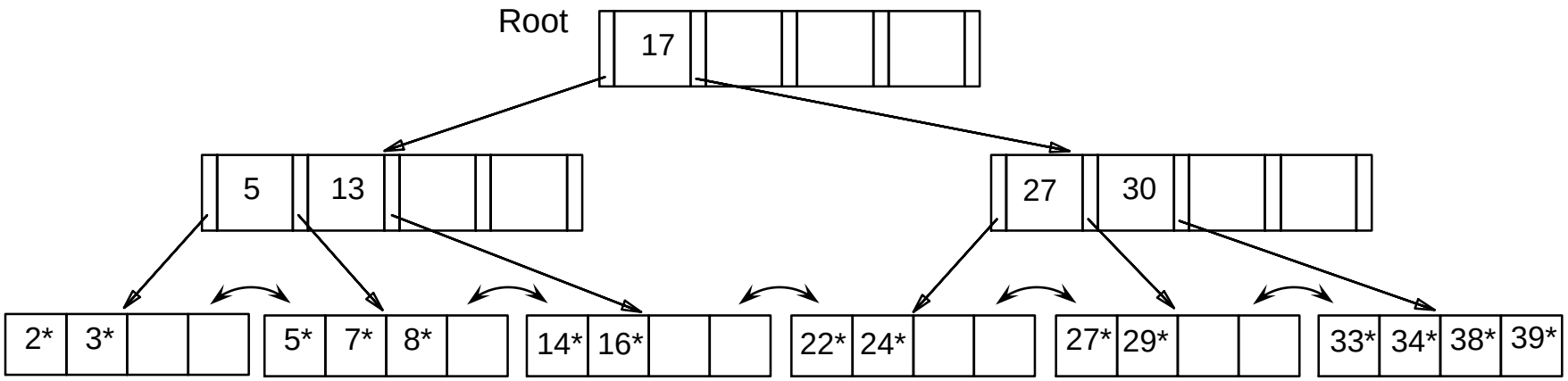
- Delete 20*



"Copy up" 27*, the lowest value in the leaf from which we borrowed 24*

B+ Tree: Examples of Deletions

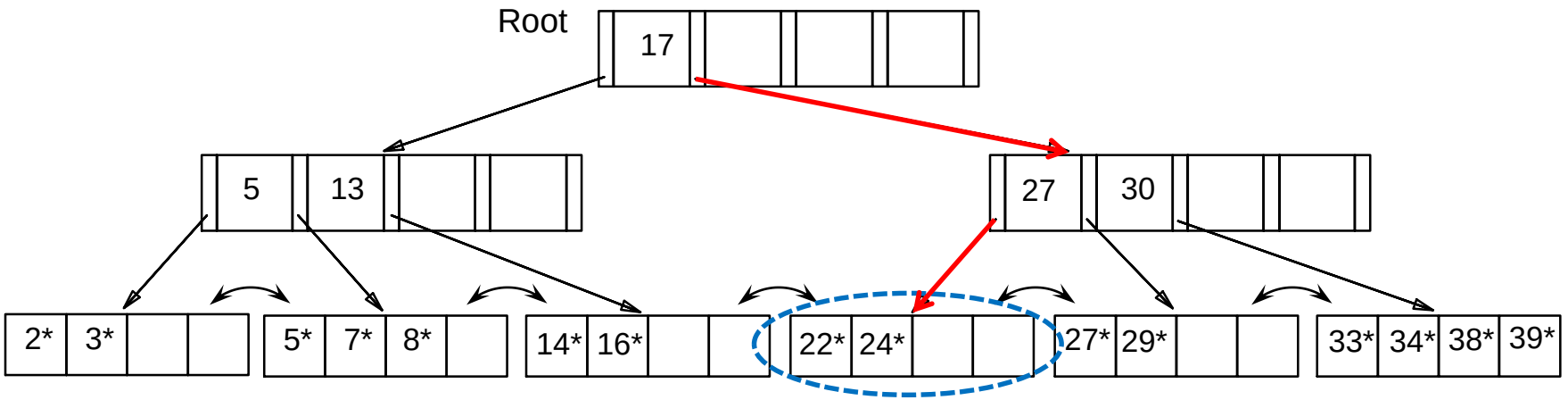
- Delete 20*



FINAL TREE!

B+ Tree: Examples of Deletions

- Delete 24*

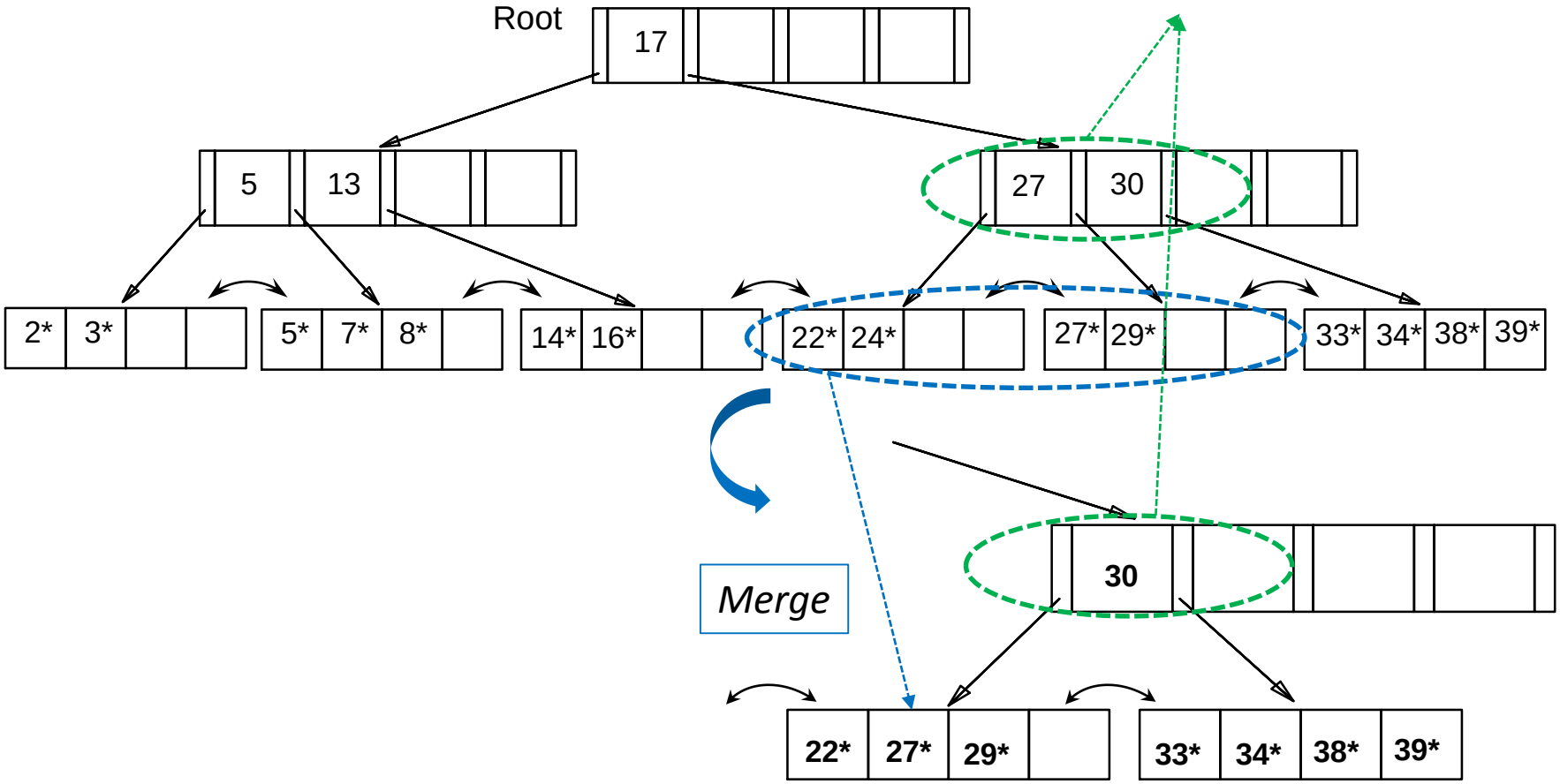


The affected leaf will contain only 1 entry and the sibling cannot lend any entry (i.e., redistribution is not applicable); hence, *merge*!

B+ Tree: Examples of Deletions

- Delete 24*

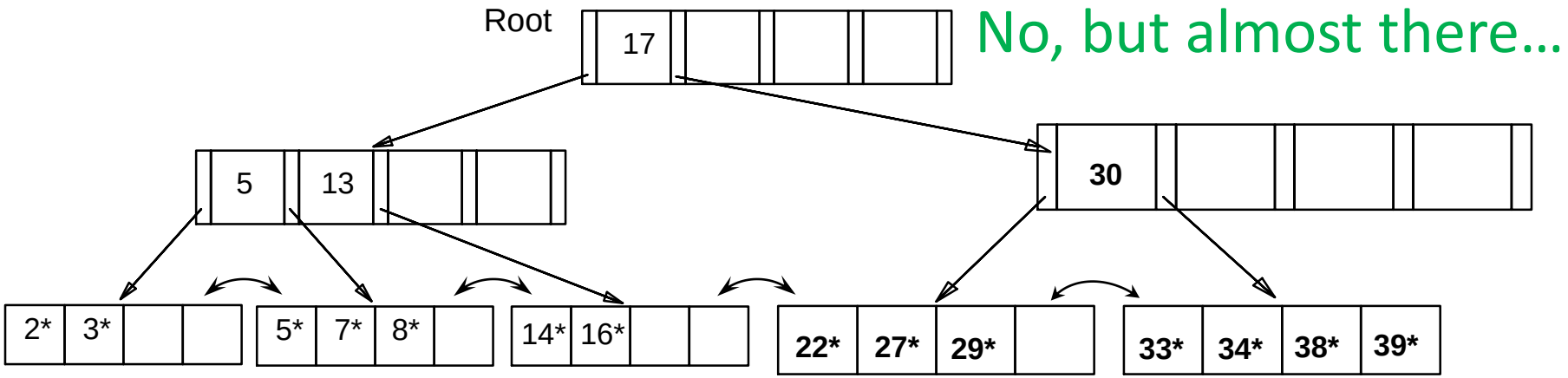
"Toss" 27 because the page that it was pointing to does not exist anymore!



B+ Tree: Examples of Deletions

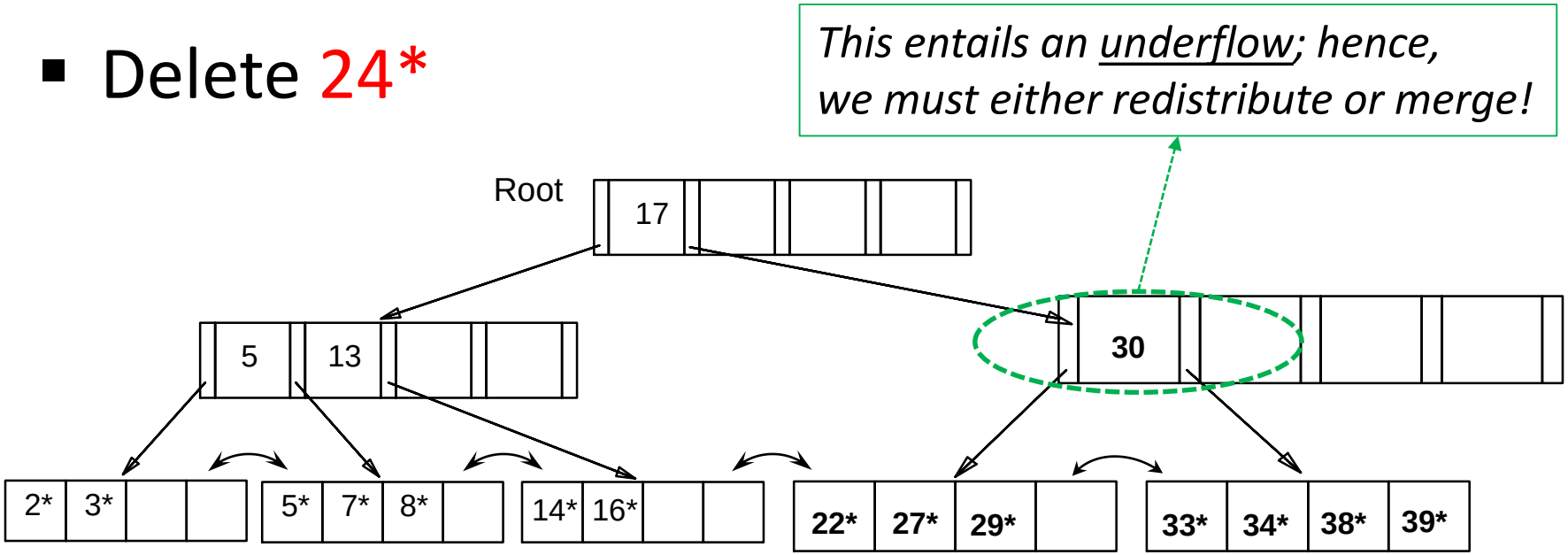
- Delete 24*

Is it done?



B+ Tree: Examples of Deletions

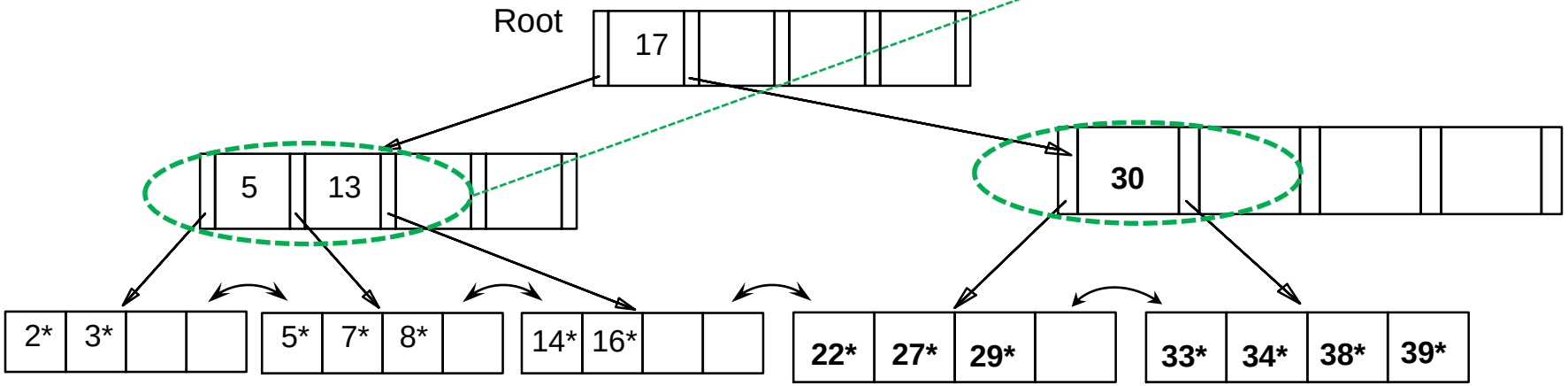
- Delete 24*



B+ Tree: Examples of Deletions

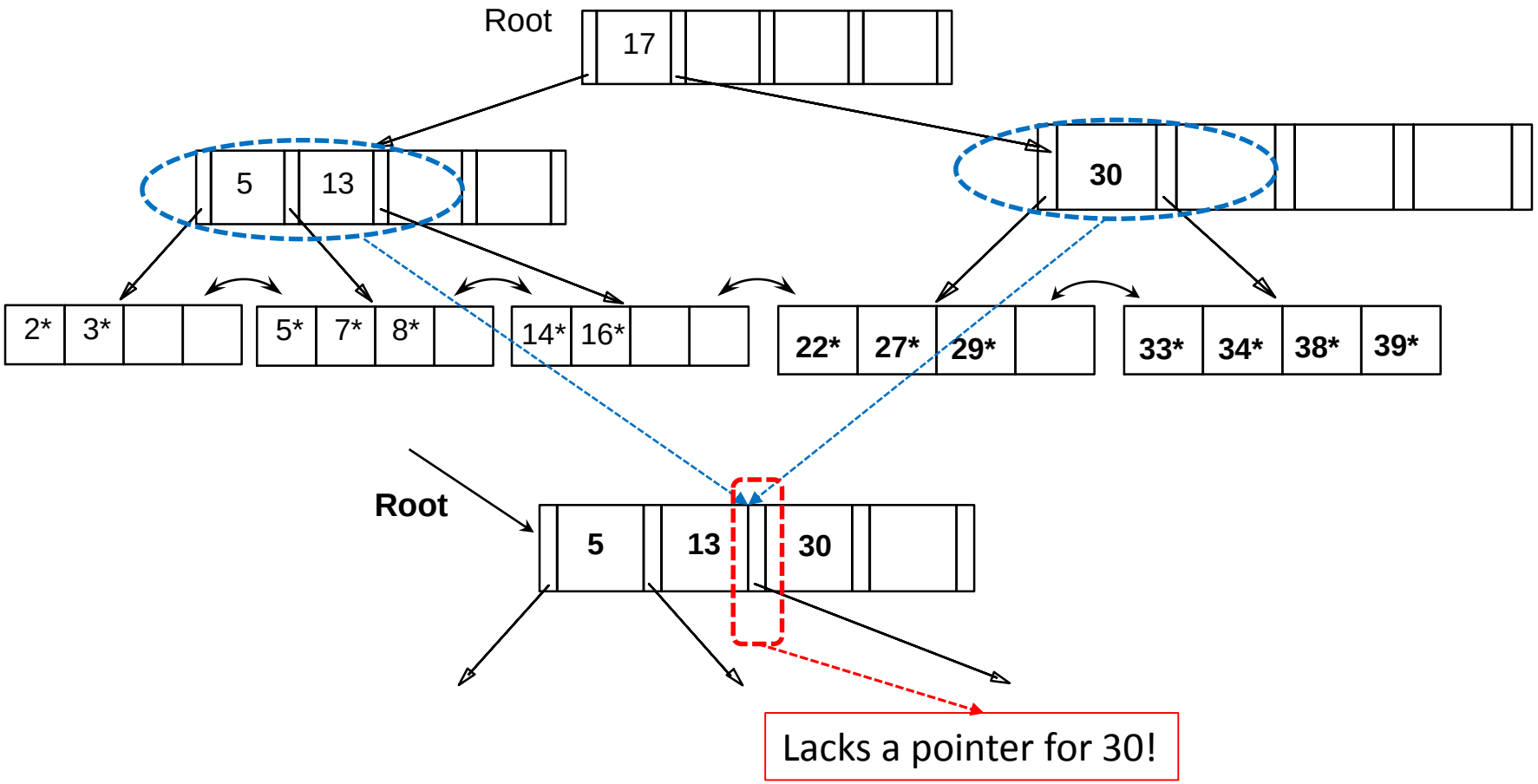
- Delete 24*

The sibling is "poor" (i.e., redistribution is not applicable)*; hence, merge!*



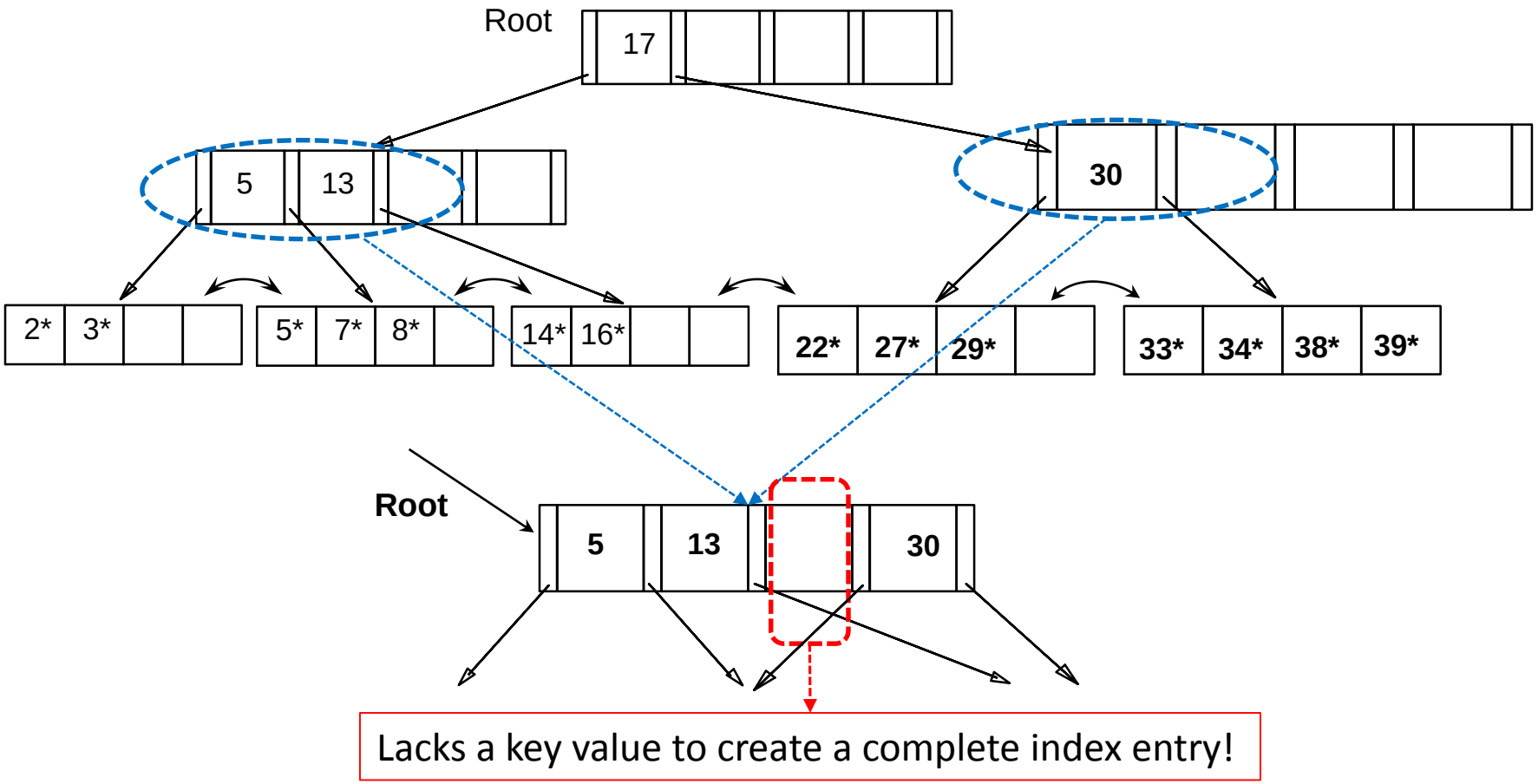
B+ Tree: Examples of Deletions

- Delete 24*



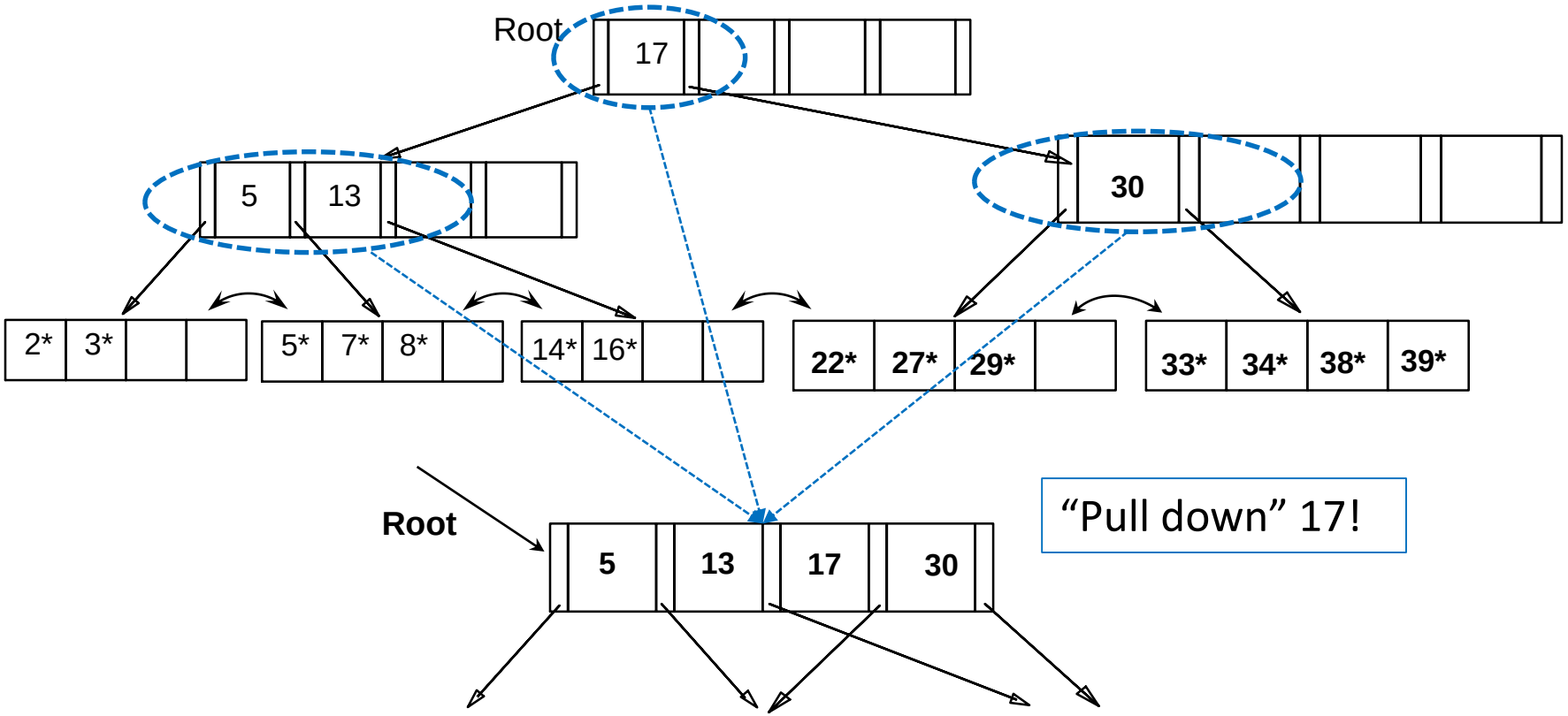
B+ Tree: Examples of Deletions

- Delete 24*



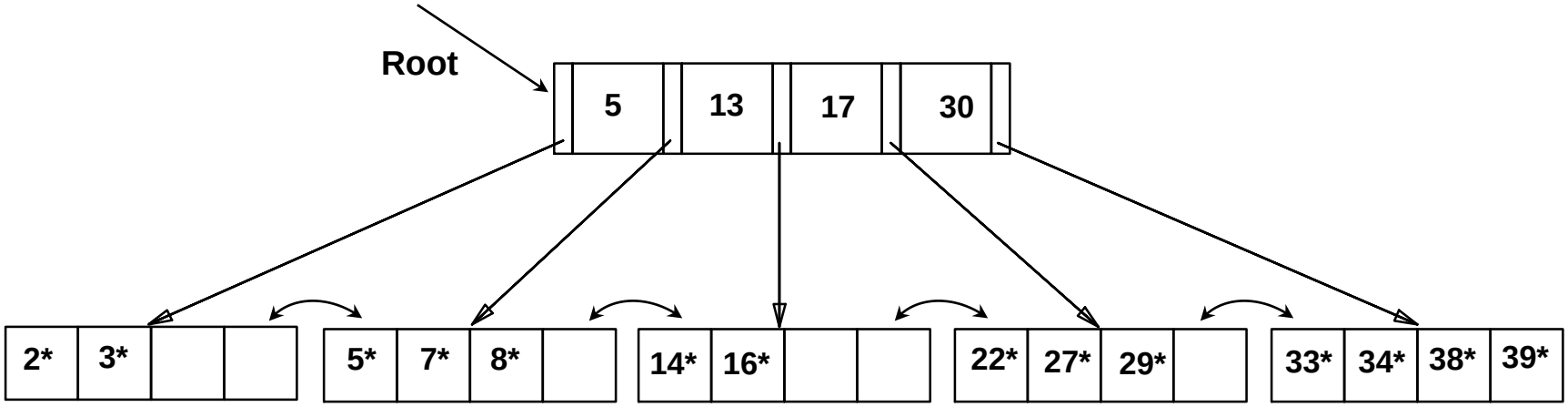
B+ Tree: Examples of Deletions

- Delete 24*



B+ Tree: Examples of Deletions

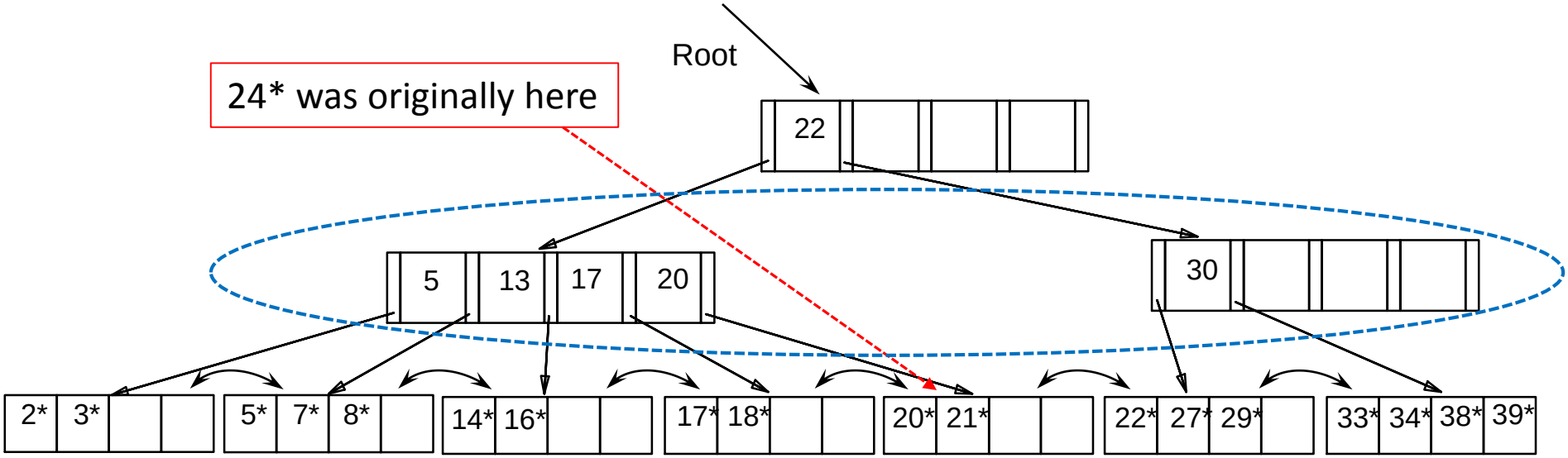
- Delete 24*



FINAL TREE!

B+ Tree: Examples of Deletions

- Delete 24*

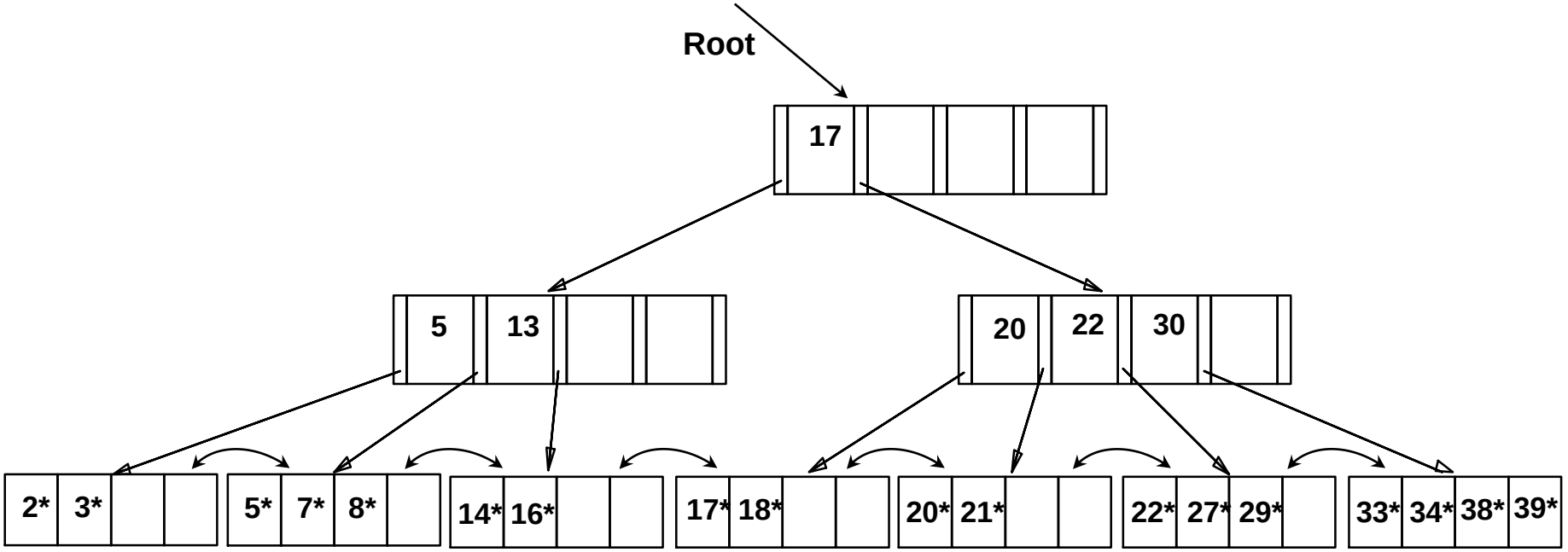


Assume (instead) the above tree *during* deleting 24*

Now we can re-distribute (*instead of merging*) keys!

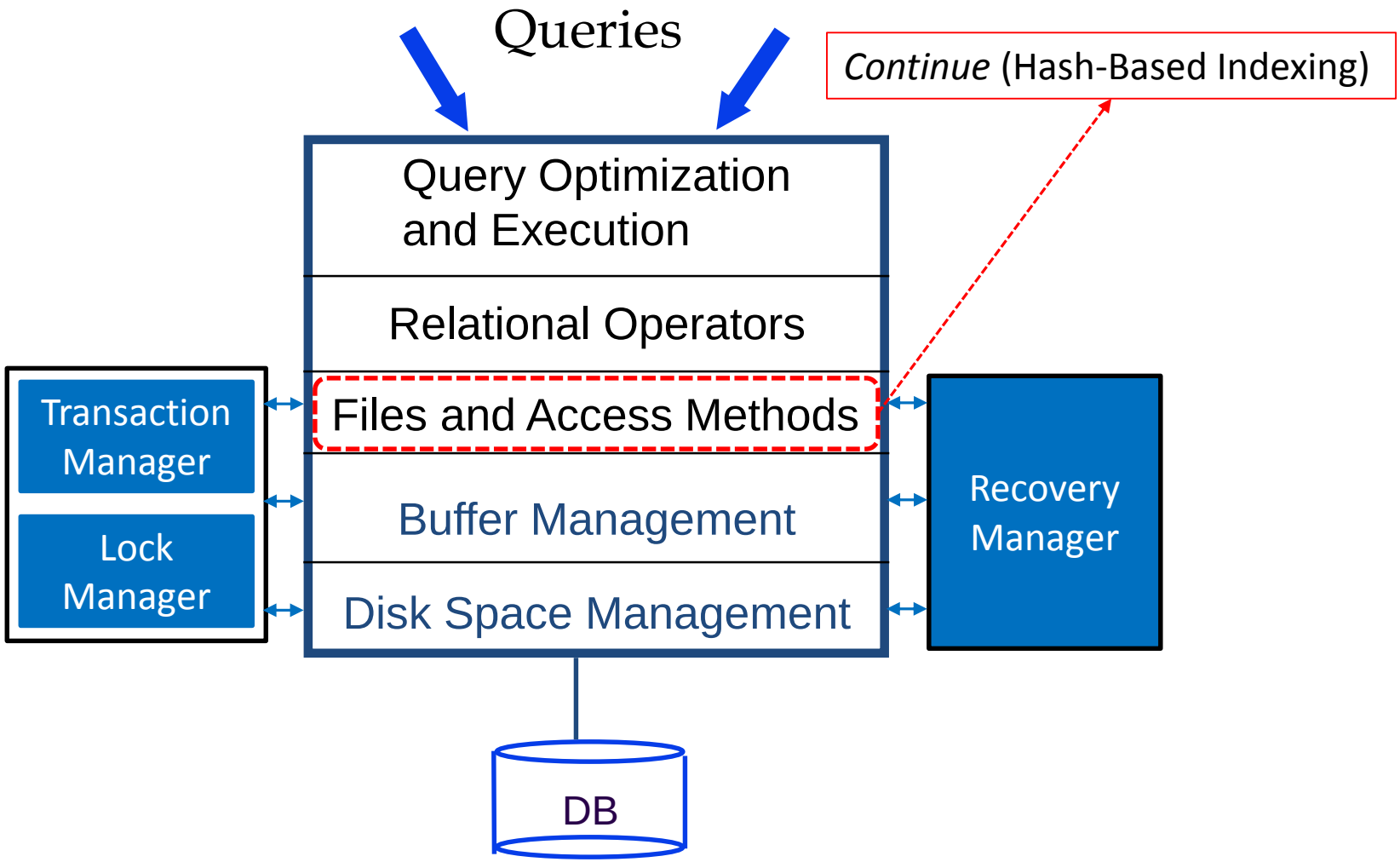
B+ Tree: Examples of Deletions

- Delete 24*



DONE! It suffices to re-distribute only 20; 17 was redistributed for illustration.

Next Class


Queries
Continue (Hash-Based Indexing)
Query Optimization and Execution
Relational Operators
Files and Access Methods
Buffer Management
Disk Space Management
Transaction Manager
Lock Manager
Recovery Manager
DB