Database Applications (15-415)

DBMS Internals- Part III
Lecture 11, February 19, 2014

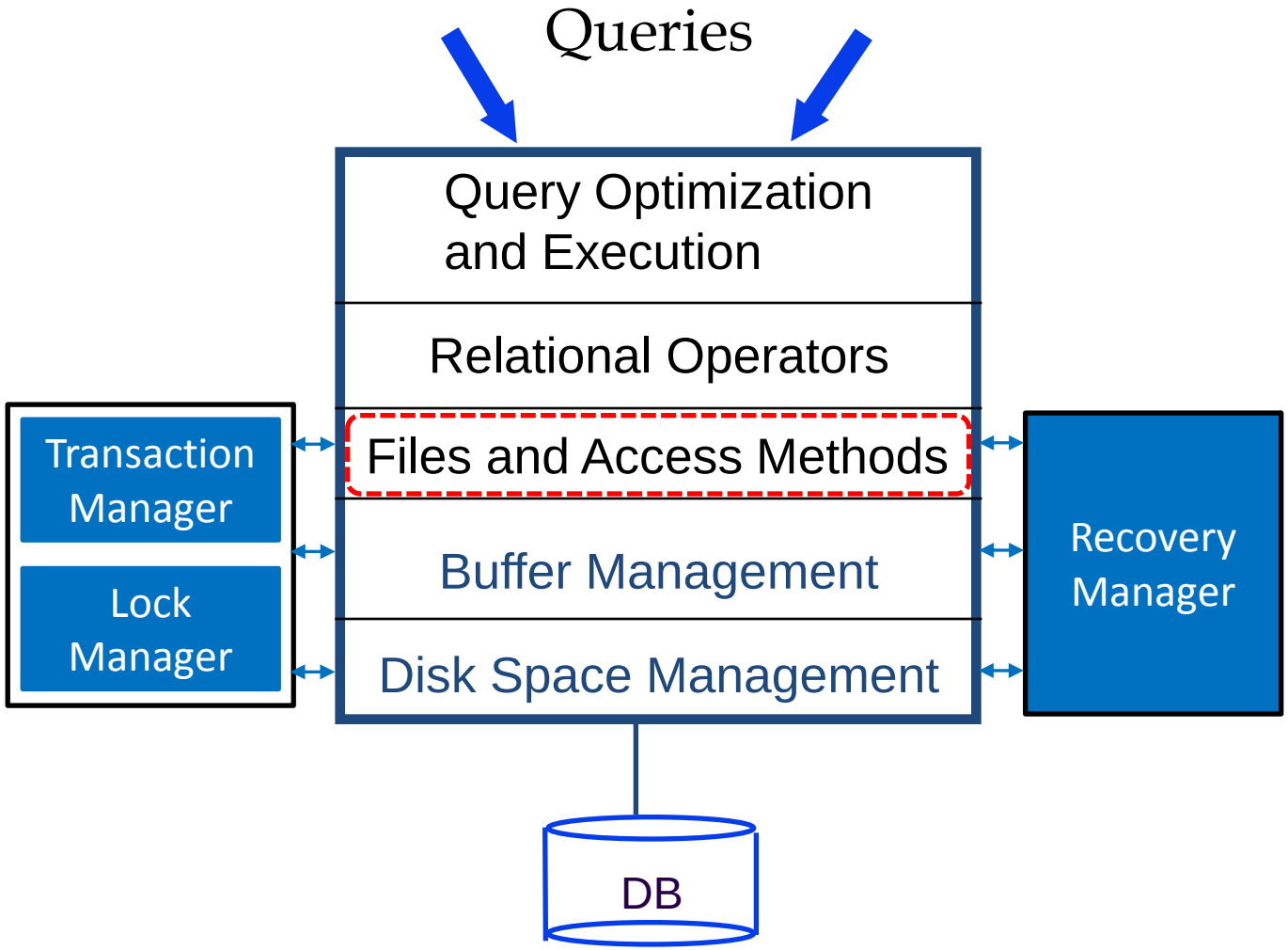
Mohammad Hammoud

جامعة كارنيجي ميلون في قطر
Carnegie Mellon University Qatar

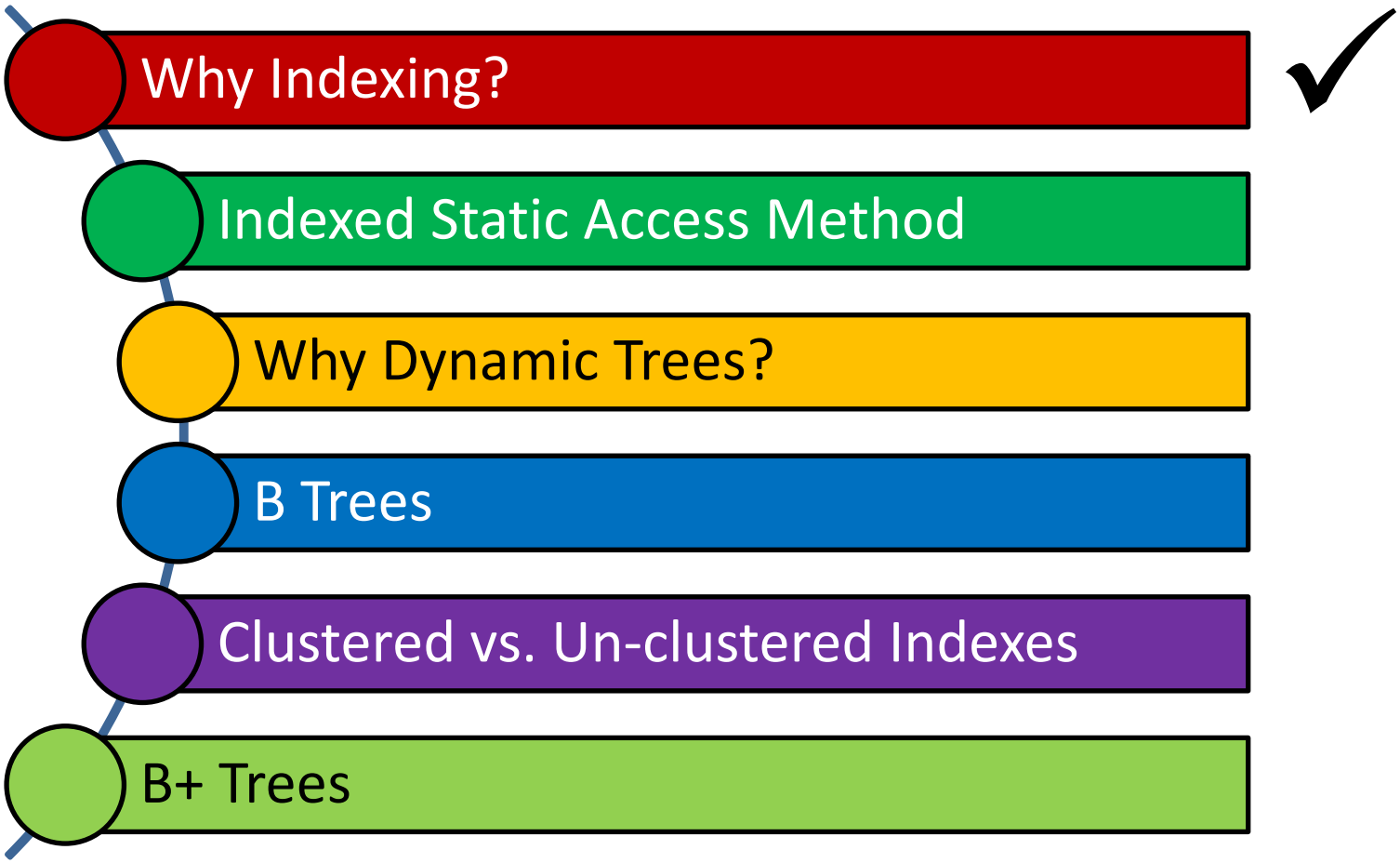
Today…

- Last Session:
  - DBMS Internals- Part II
    - Brief summaries of disks, disk space management, and buffer management
    - Files and Access Methods (file organizations and motivation for indexing)
- Today's Session:
  - DBMS Internals- Part III
    - Tree-based indexes: ISAM, B and B+ (*if time allows*) trees
- Announcements:
  - PS3 is now posted. It is due on March 02, 2014
  - The midterm exam is on Wednesday Feb 26 (*all* material are included)

DBMS Layers

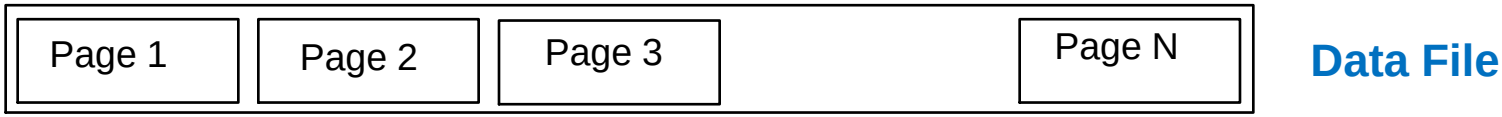

Queries
Query Optimization and Execution
Relational Operators
Files and Access Methods
Buffer Management
Disk Space Management
Transaction Manager
Lock Manager
Recovery Manager
DB

Outline



Motivation

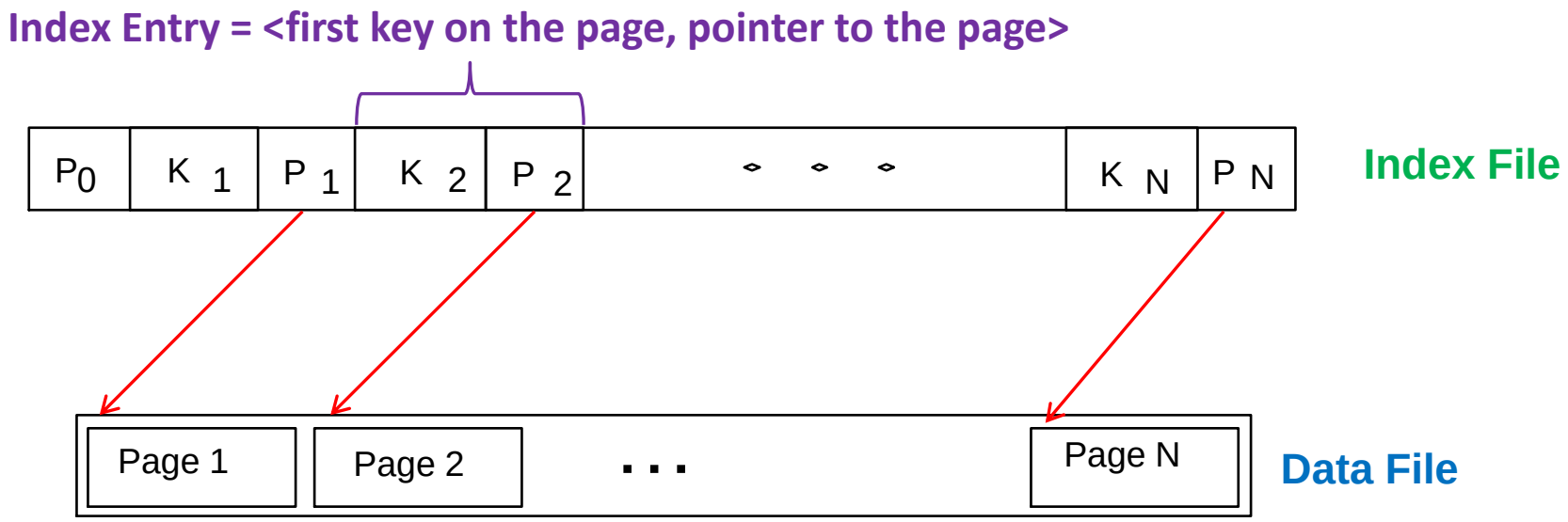
- Consider a file of student records *sorted* by GPA



- 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

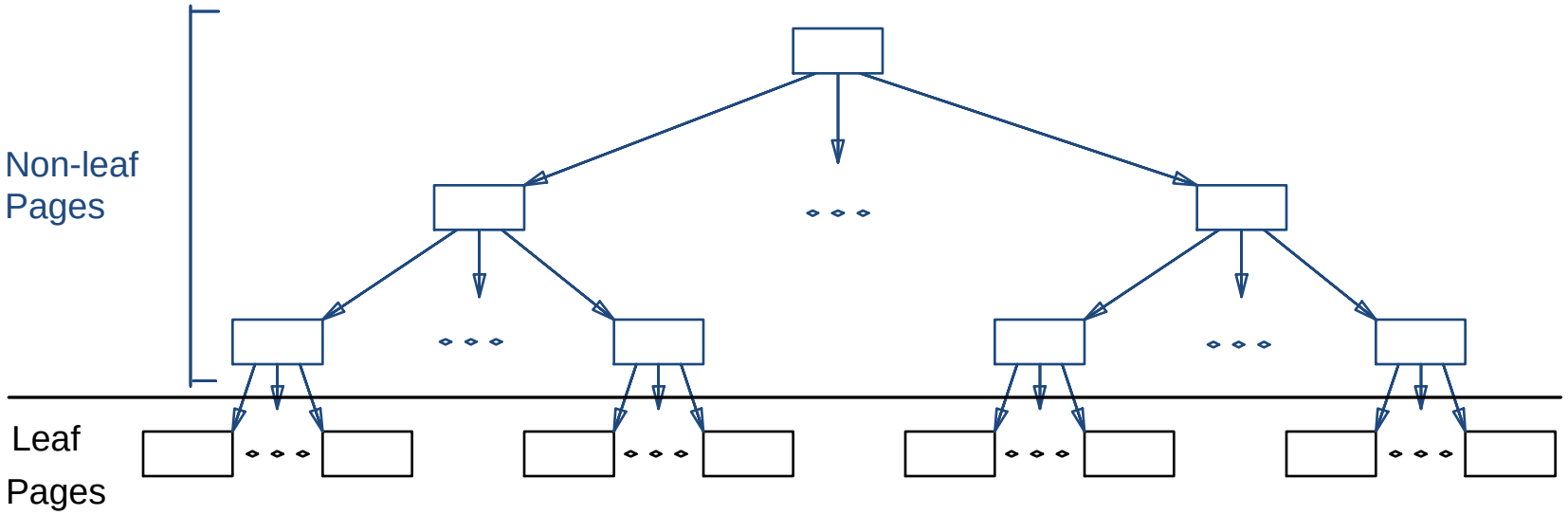
- What about creating an *index file* (with one entry per page) and do binary search there?



- 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?

Where to Store Data Records?

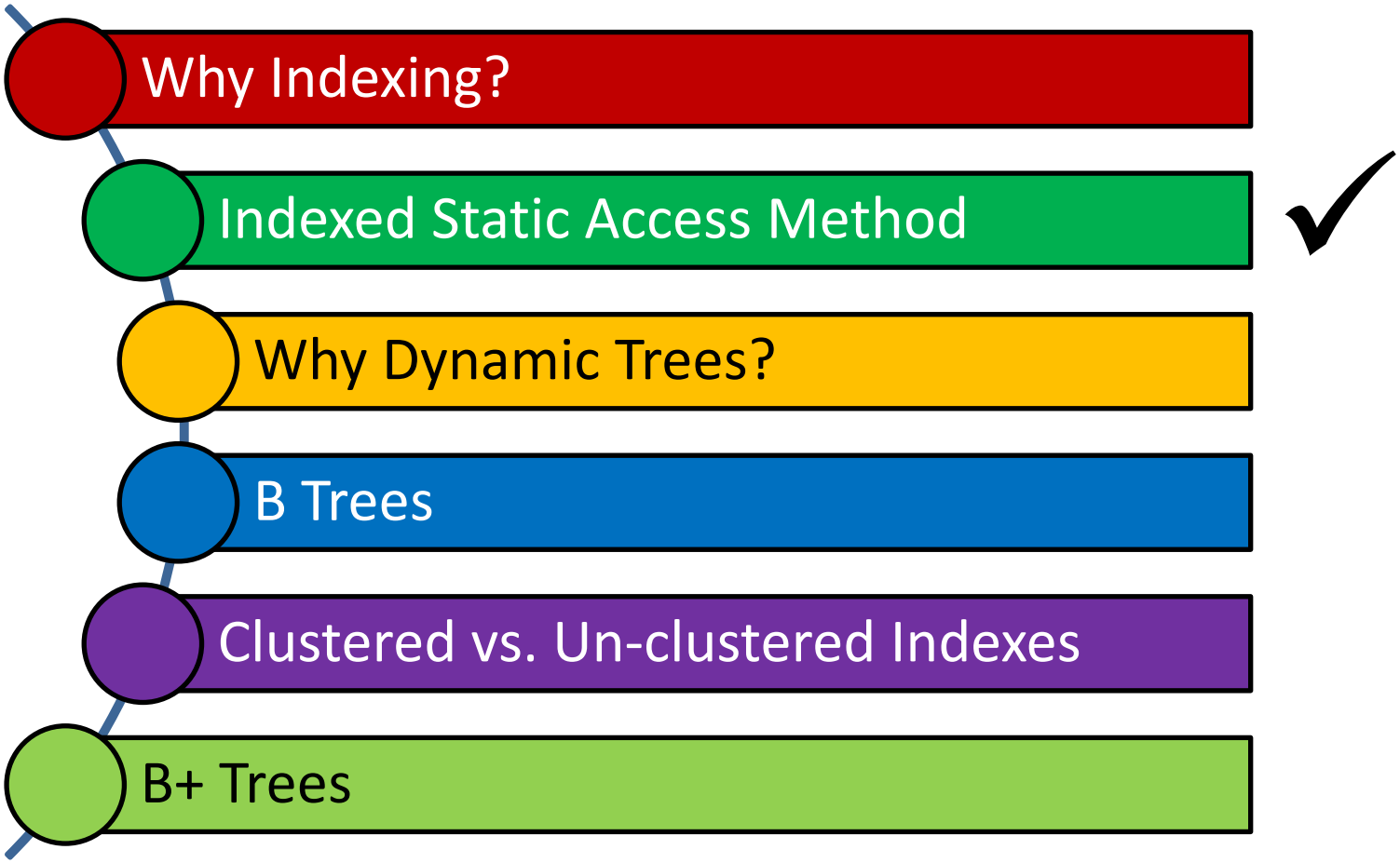
- In general, *3 alternatives* for "data records" (each referred to as ***k*)*** can be pursued:
  - Alternative (1): *K** is an actual data record with key ***k***
  - Alternative (2): *K** is a <***k***, **rid>** pair, where rid is the record id of a data record with search key ***k***
  - Alternative (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:
  - **A (1)**: Leaf pages contain the actual data (i.e., the data records)
  - **A (2)**: Leaf pages contain the <key, rid> pairs and actual data records are stored in a separate file
  - **A (3)**: Leaf pages contain the <key, rid-list> pairs and actual data records are stored in a separate file

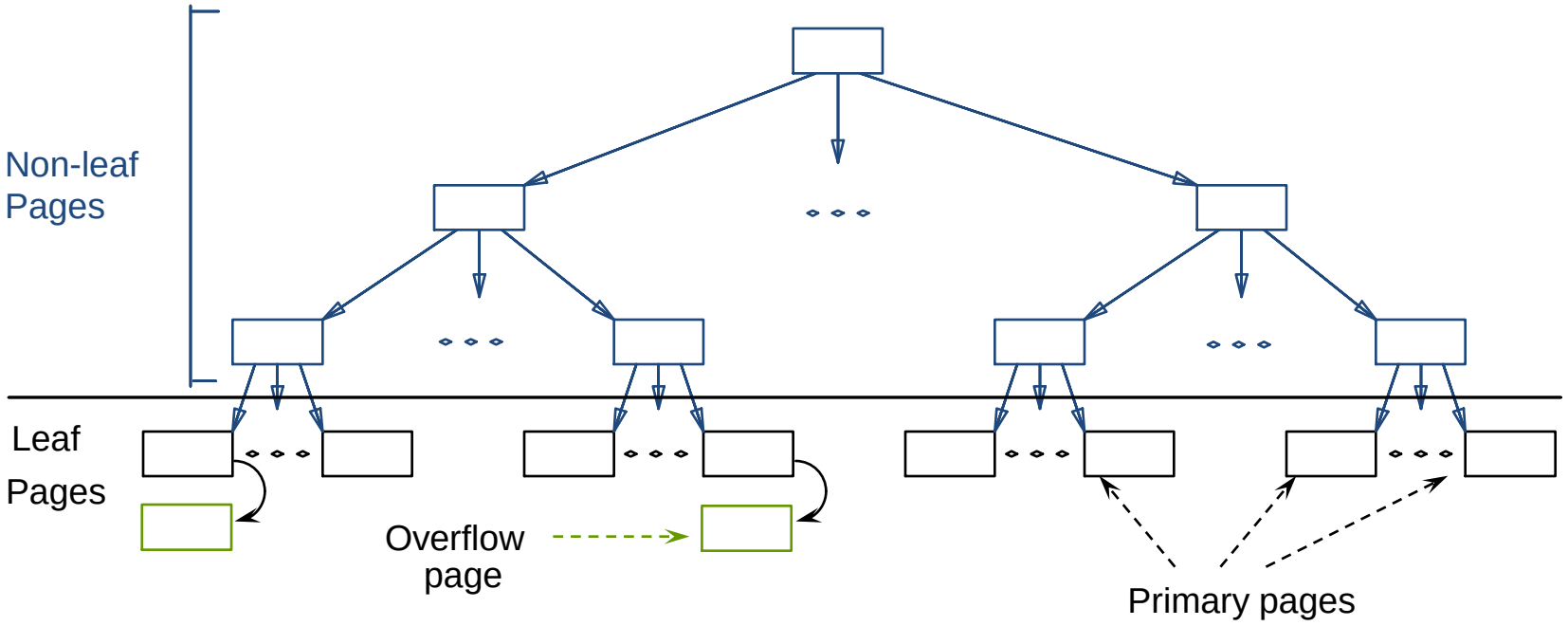
The choice among these alternatives is orthogonal to the *indexing technique.*

Outline



ISAM Trees: Page Overflows

- Now, what if there are a lot of insertions?

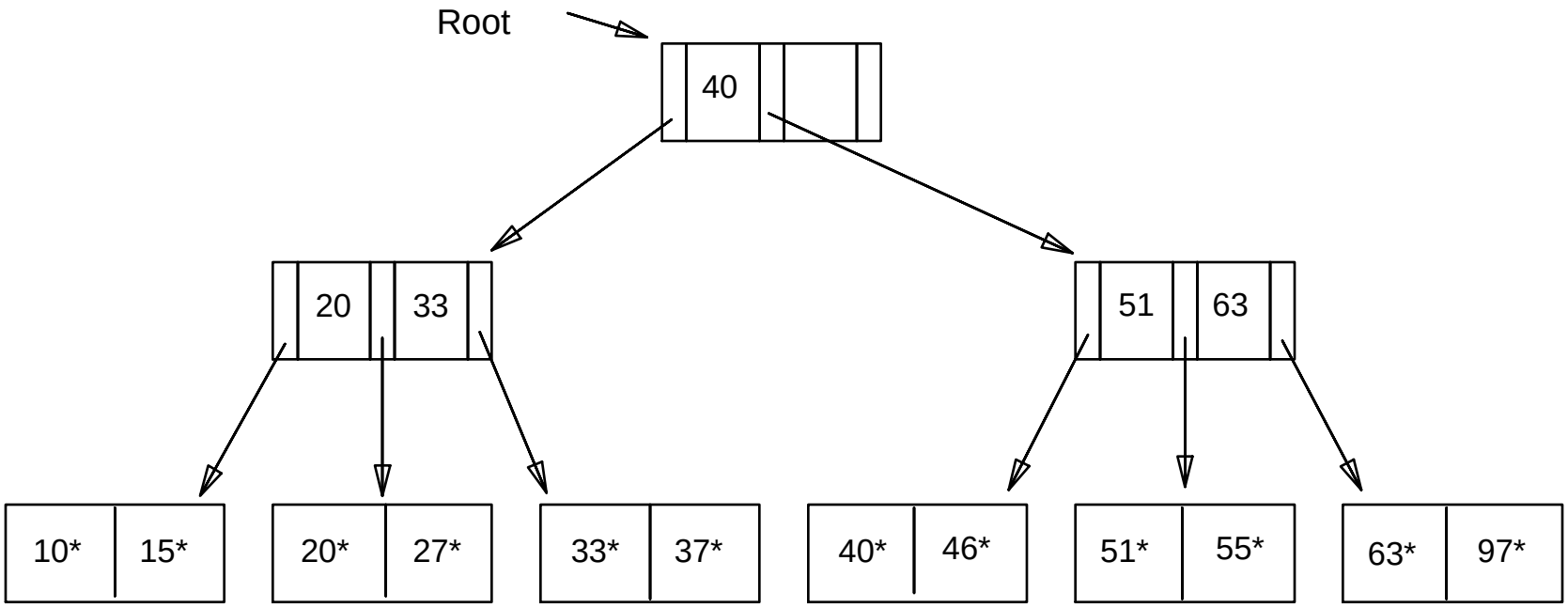


This structure is referred to as *Indexed Sequential Access Method* (ISAM)

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

An Example of ISAM Trees

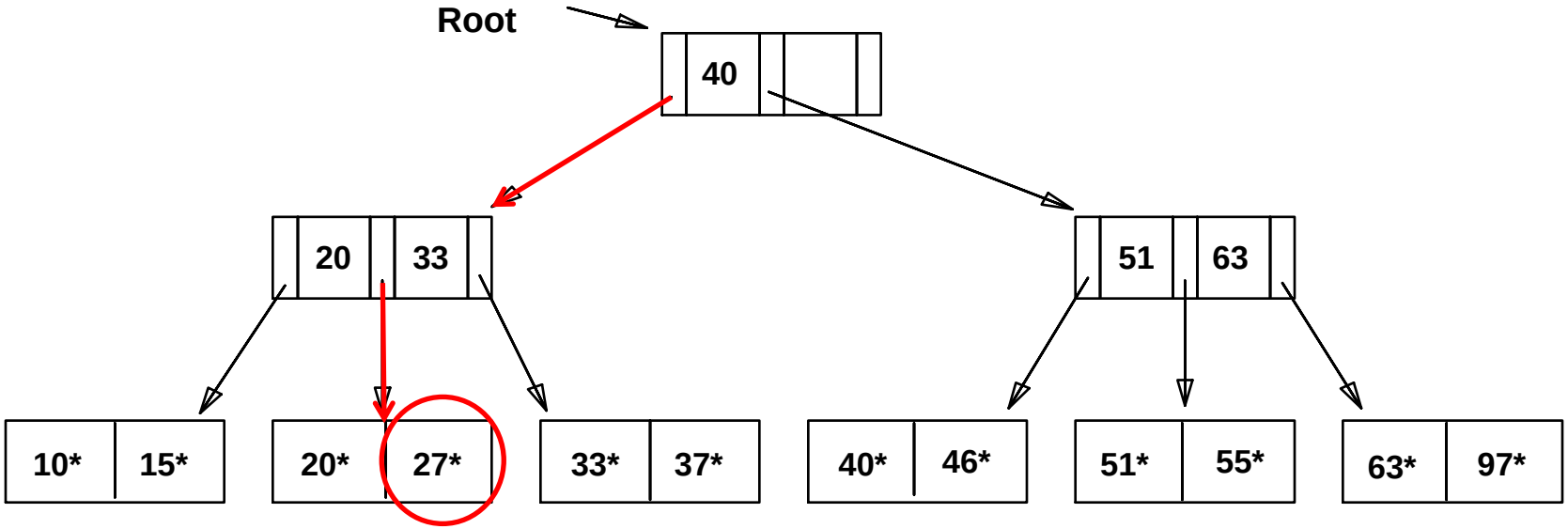


2 Entries Per Page.

ISAM: Searching for Entries

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

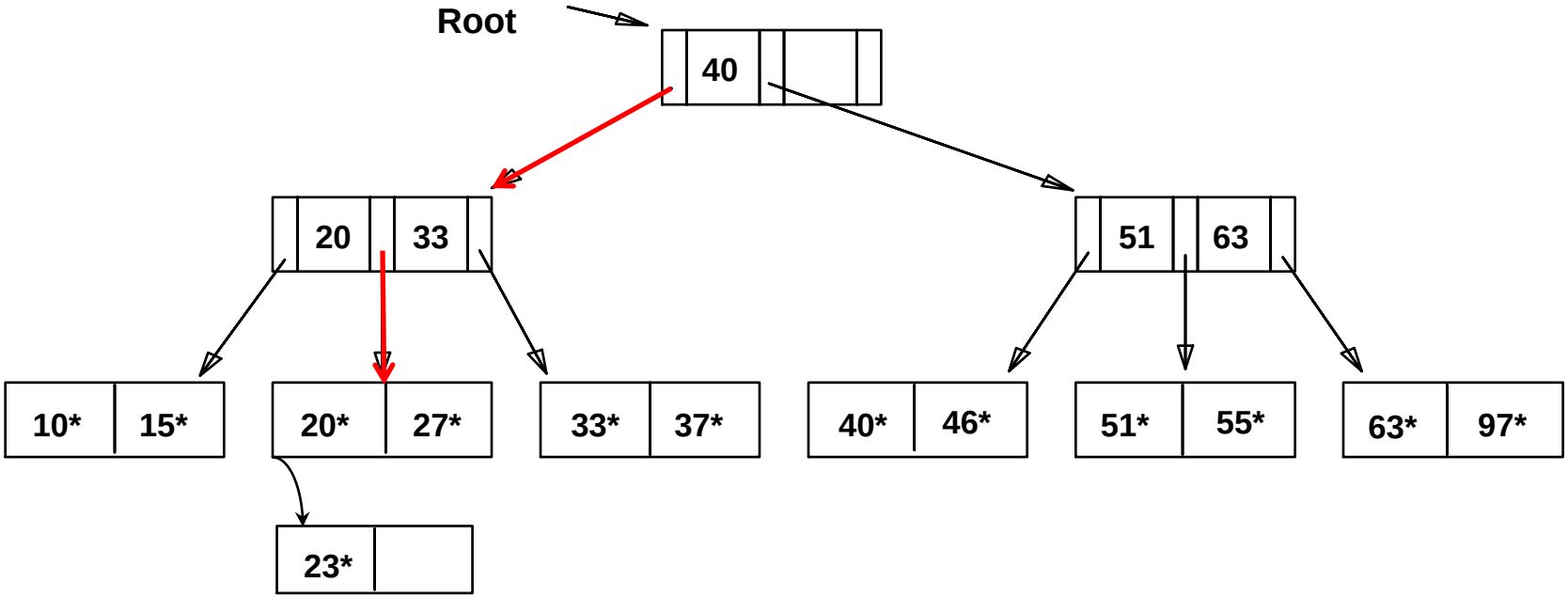
- Search for 27*



ISAM: Inserting Entries

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

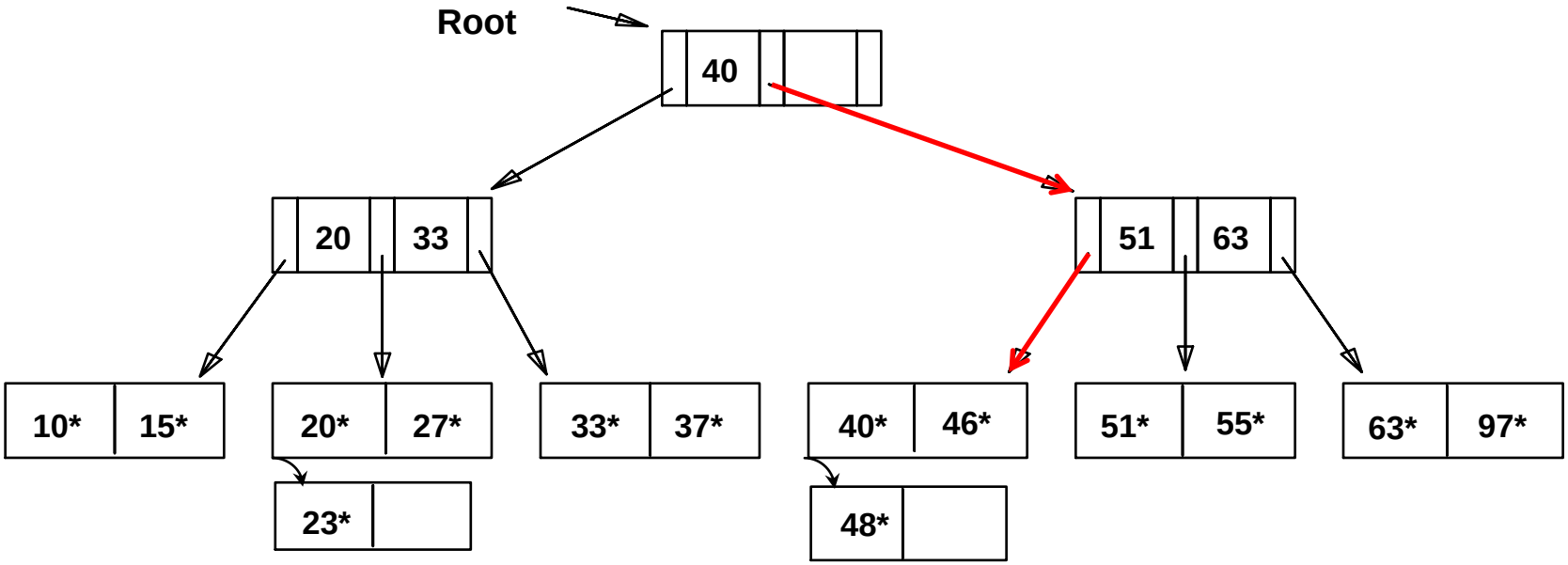
- Insert 23*



ISAM: Inserting Entries

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

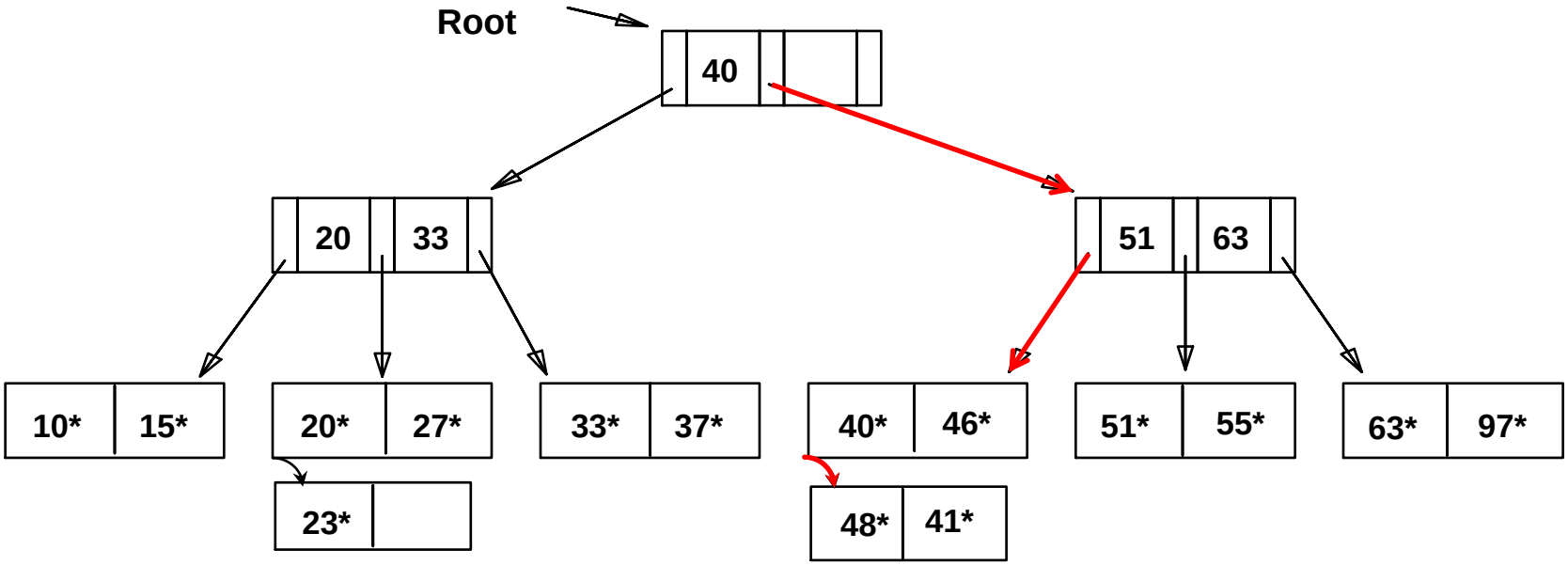
- Insert 48*



ISAM: Inserting Entries

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

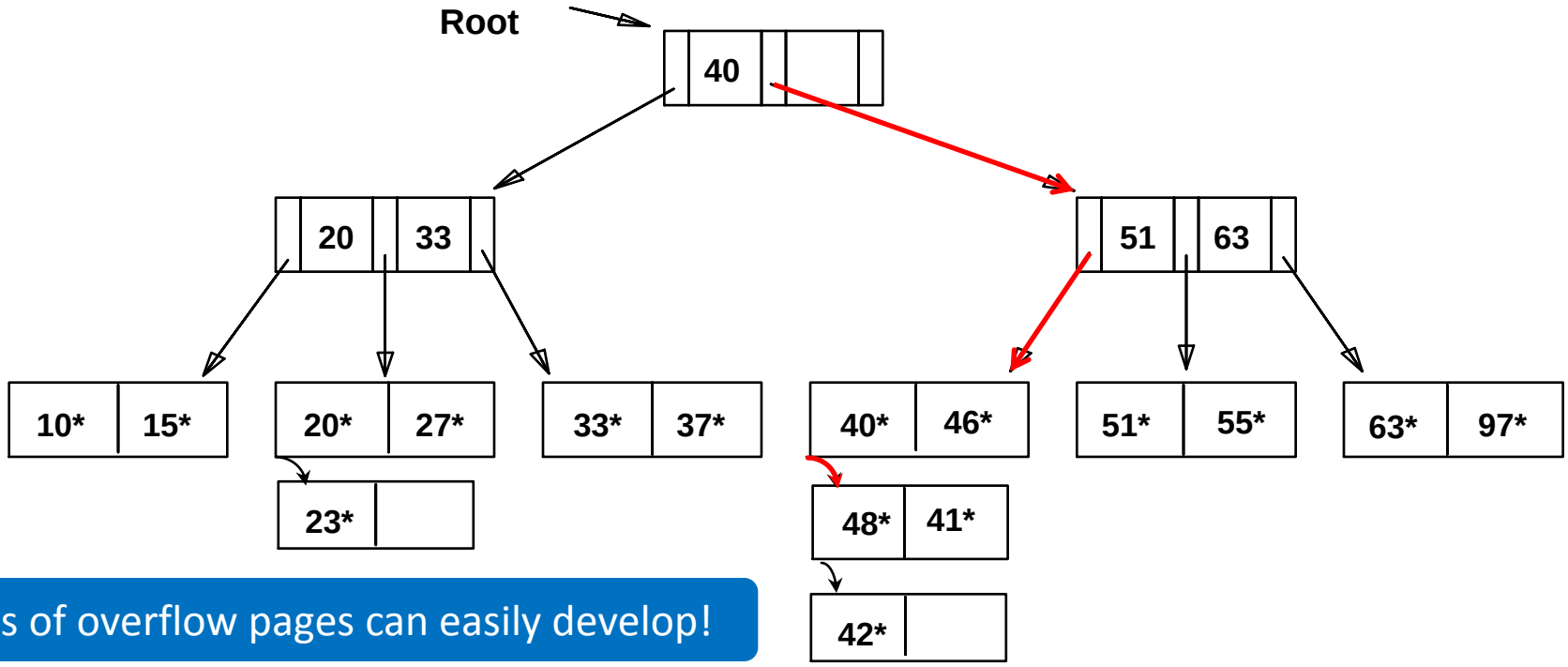
- Insert 41*



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*

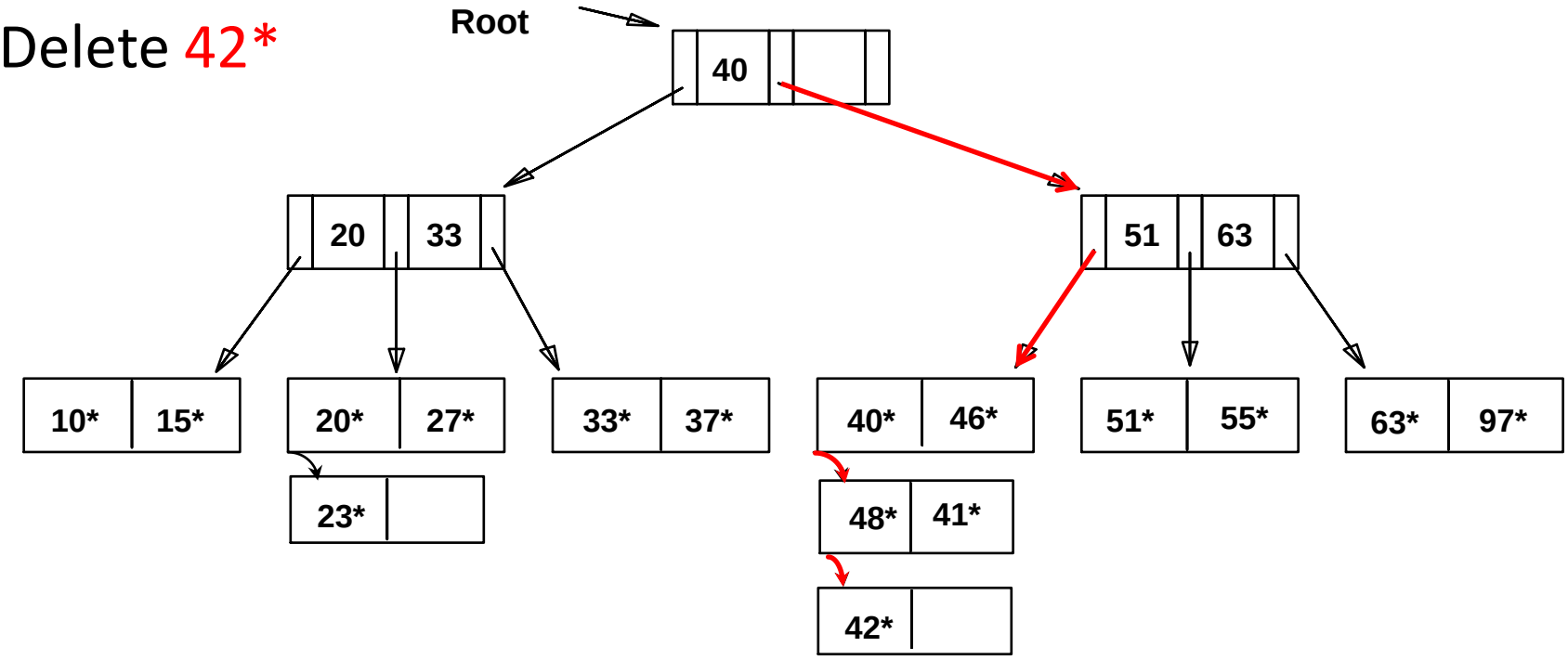


Chains of overflow pages can easily develop!

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*)

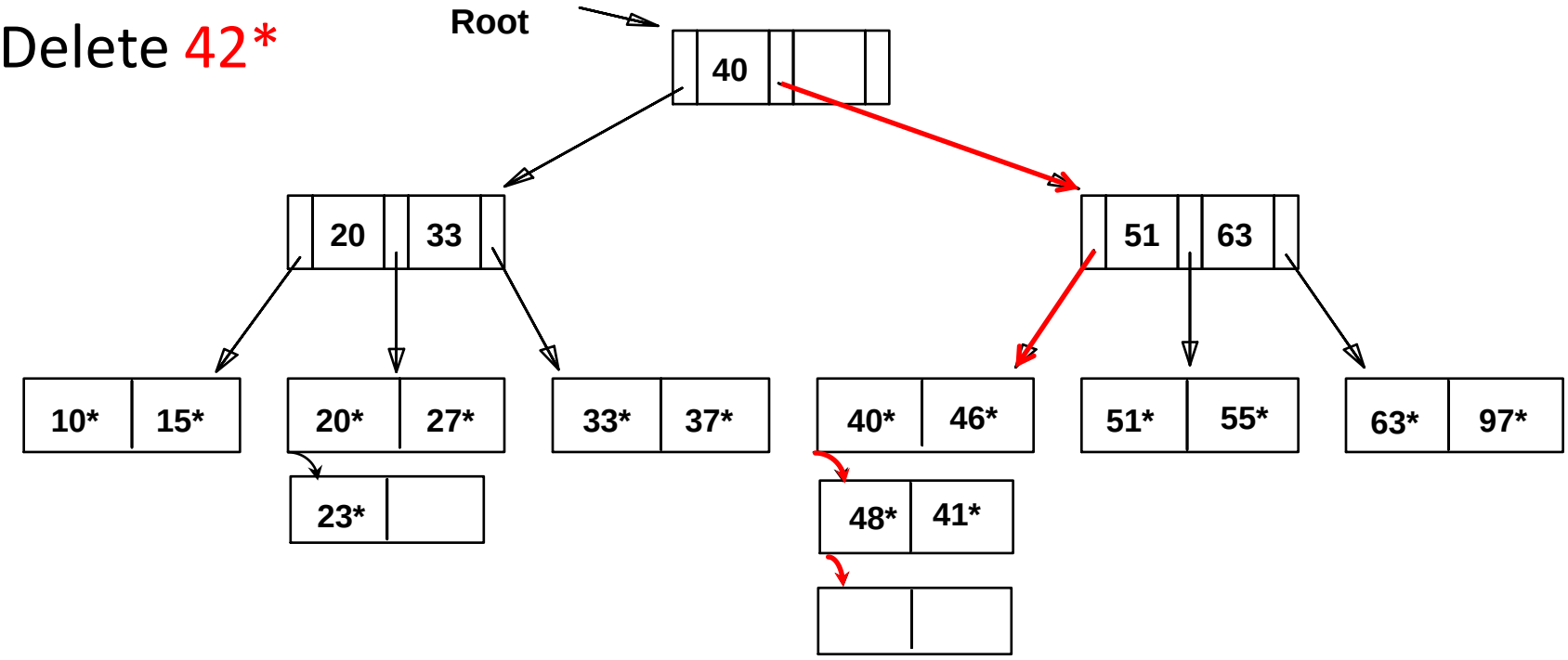
- Delete 42*



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*)

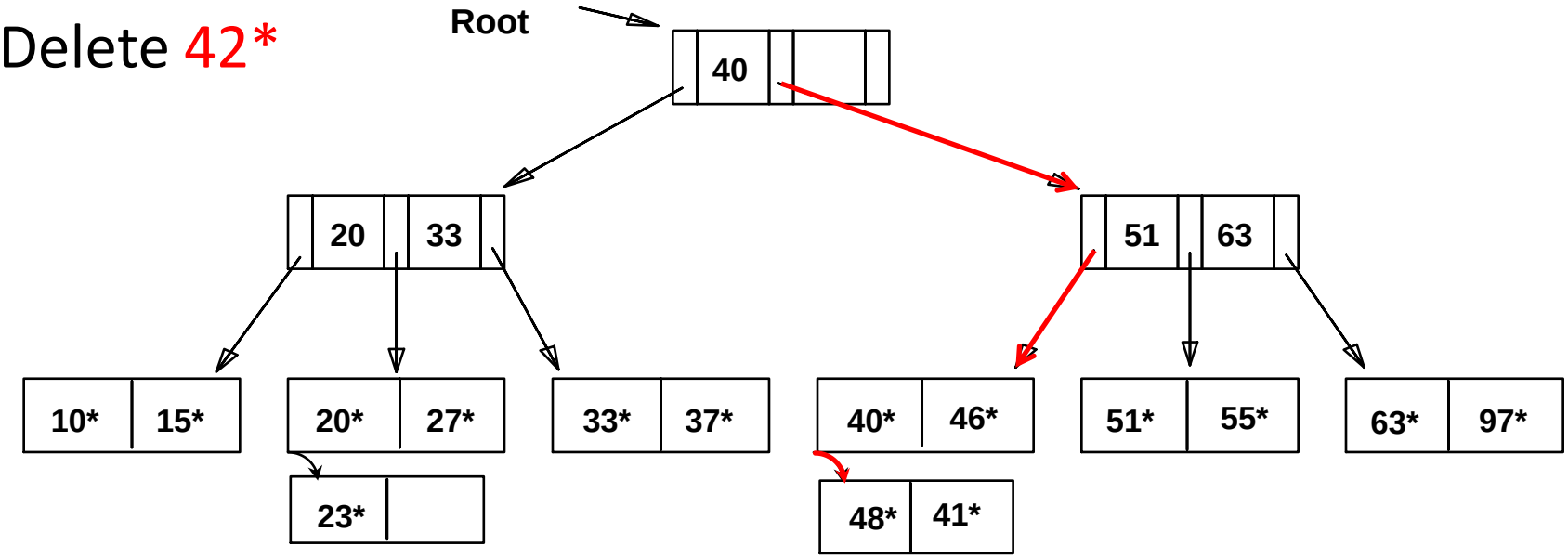
- Delete 42*



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*)

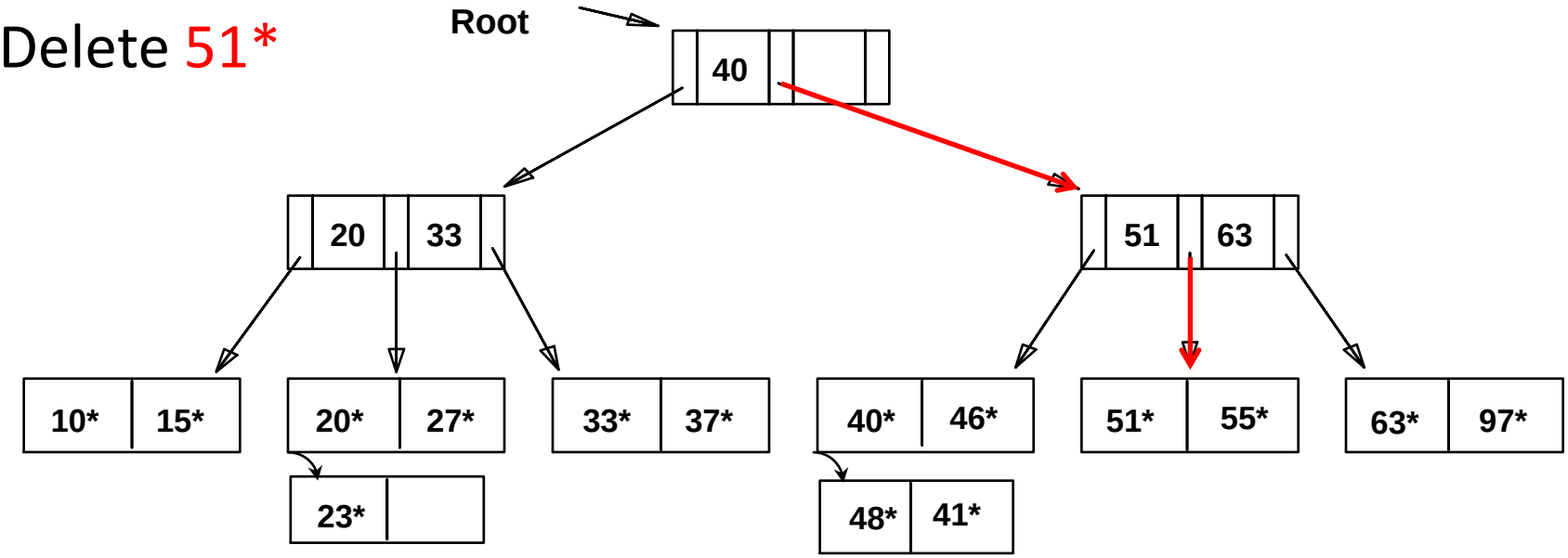
- Delete 42*



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*

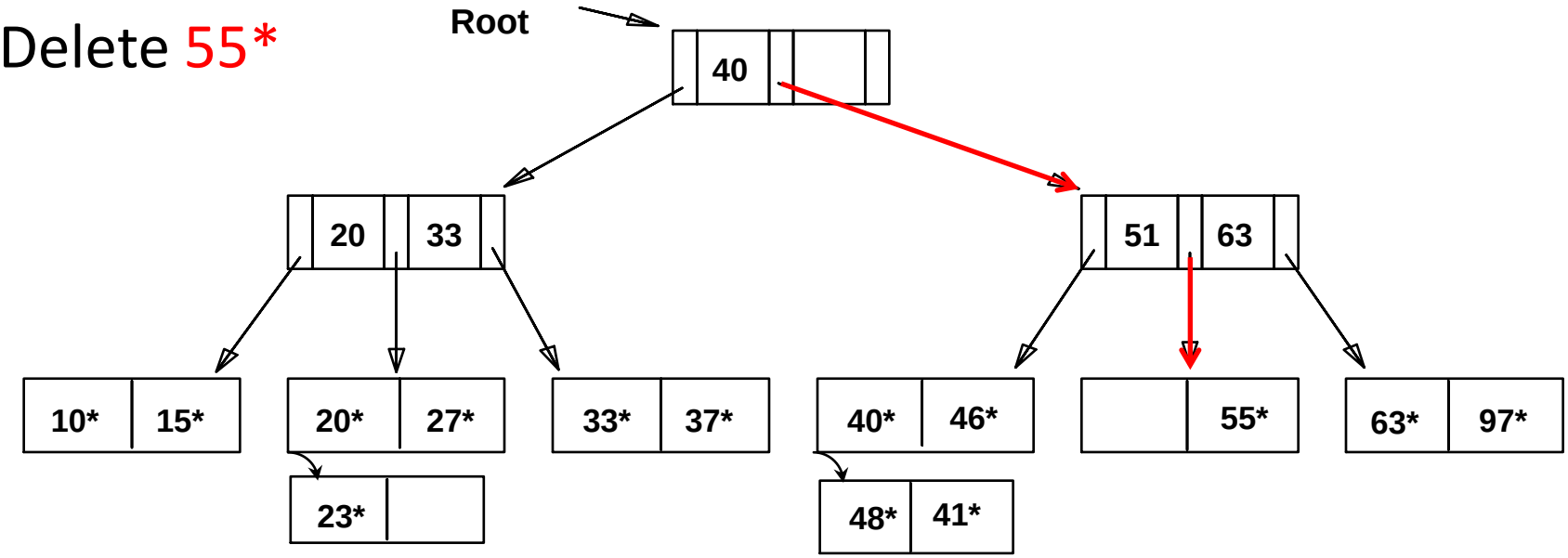


Note that 51 still appears in an index entry, but not in the leaf!

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

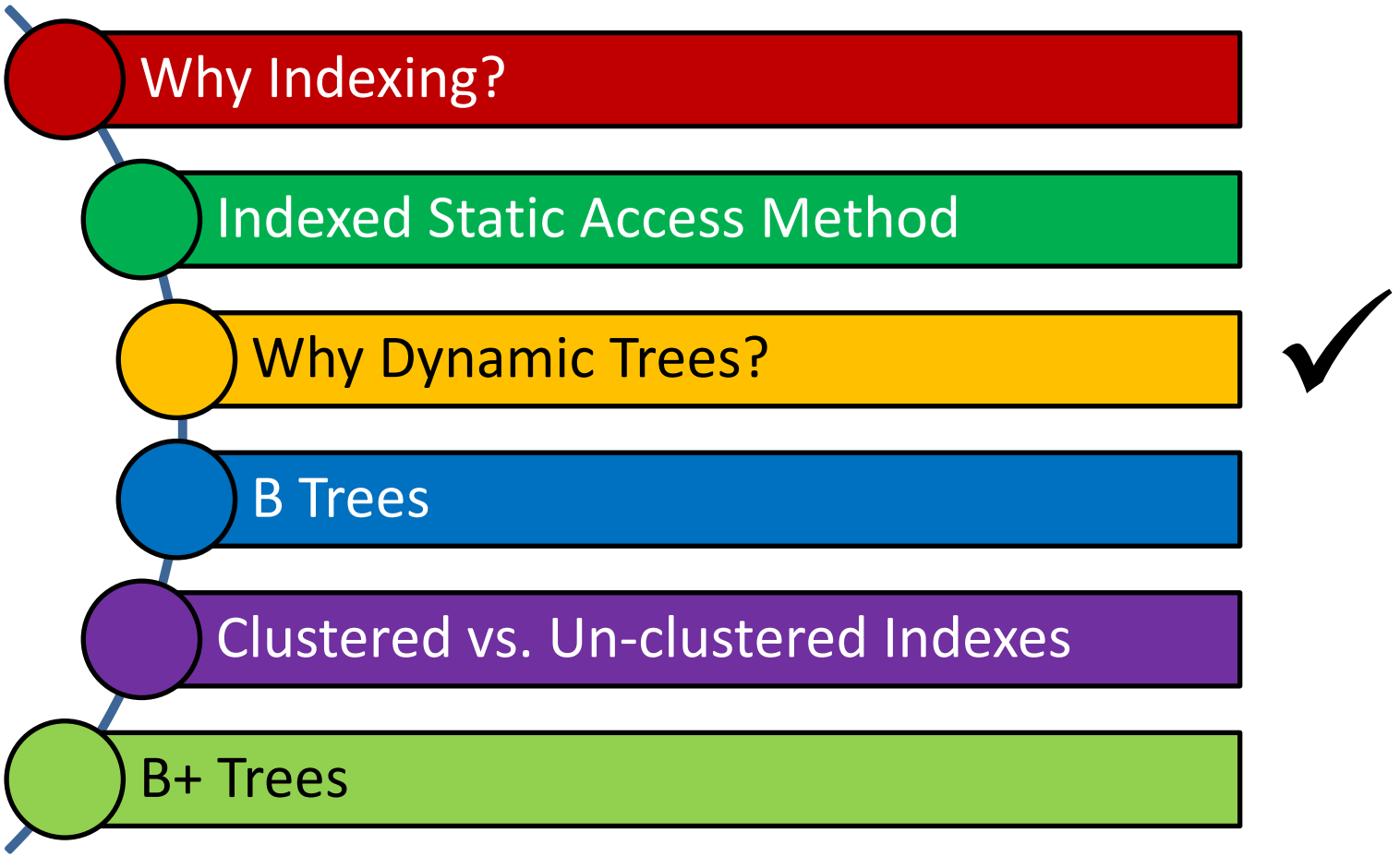
- Once an ISAM file is created, insertions and deletions affect only the contents of leaf pages (i.e., *ISAM is a static structure*!)

- 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



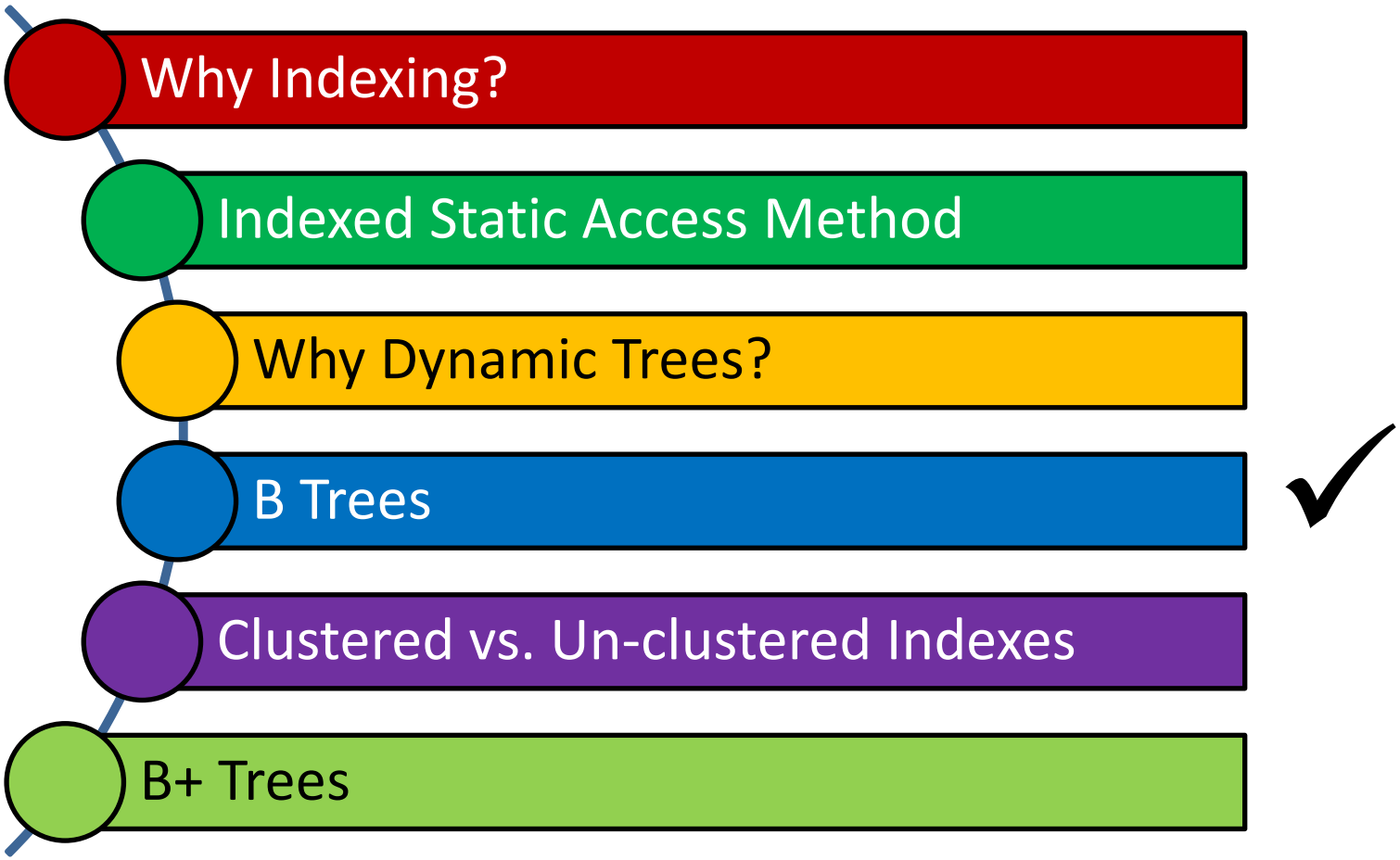
Dynamic Trees

- 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 are B and B+ trees

B and B+ Trees

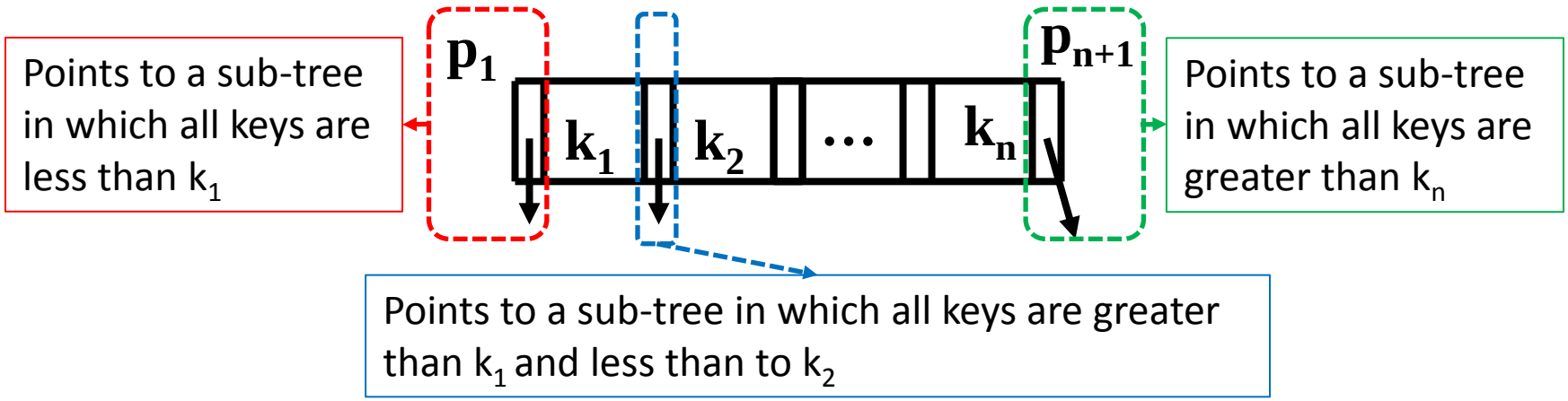
- B and B+ trees are designed to work on disks
  - A B/B+ tree node is usually as large as a whole disk page
  - B/B+ trees copy selected pages from disk into main memory as needed
  - Only a constant number of pages exit in memory at any time; hence, the size of the memory does not limit the size of a B/B+ tree that can be handled
  - O(log (***N***)) for any operation, assuming ***N***-key B/B+ tree!

Outline



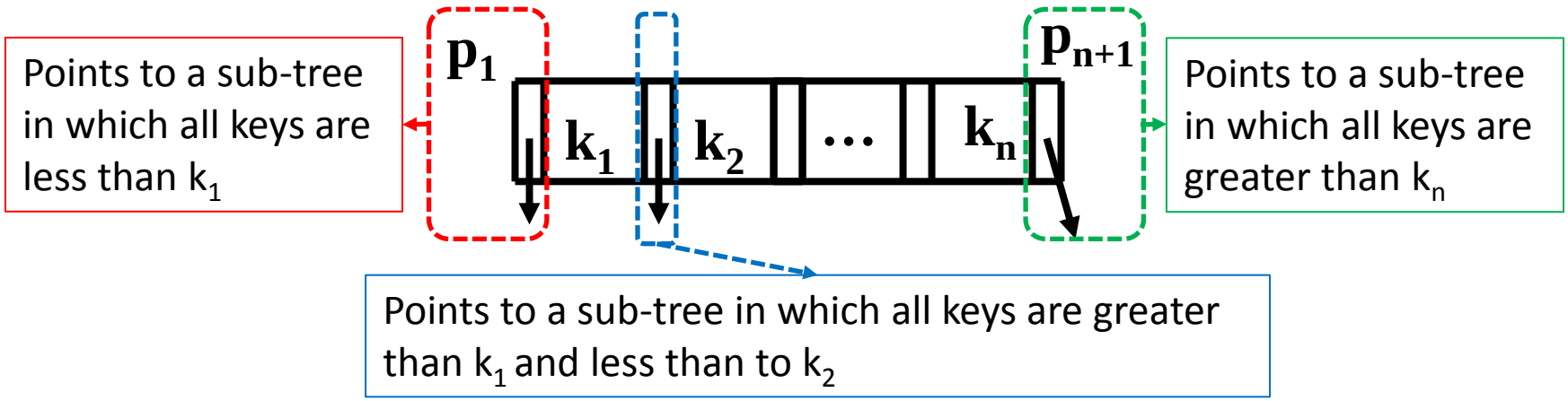
B Tree Properties

- 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***



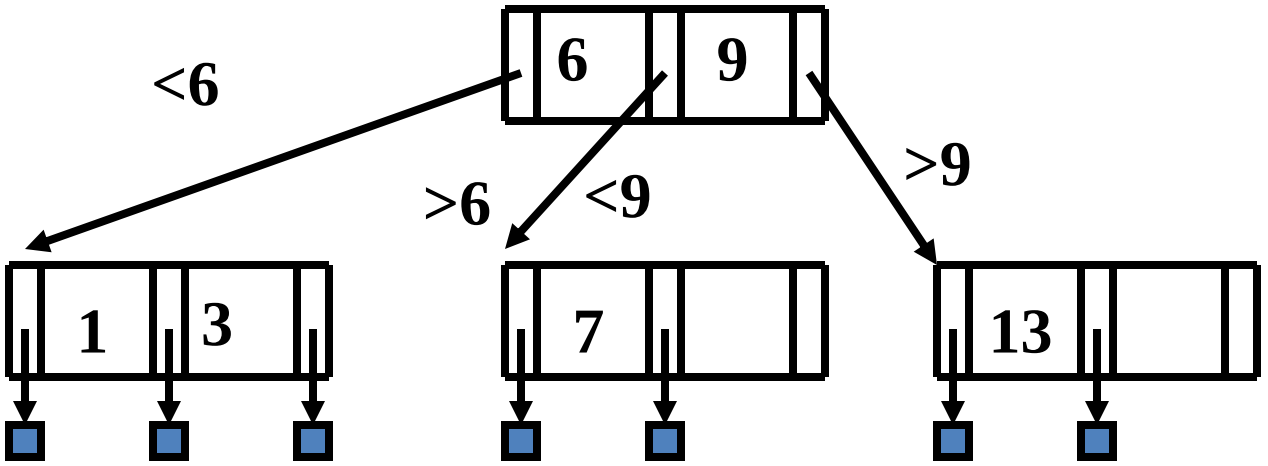
B Tree Properties

- Each node in a B tree of order ***d*** (this is a measure of the capacity of a tree):
  - Has at most ***2d*** keys
  - A variant of a B tree, known as ***B* tree***, requires each internal node to be at least 2/3 full, rather than half full, as a B tree requires.
  - All leaves are on the same level
  - Has exactly ***n-1*** keys if the number of pointers is ***n***



A B Tree Example

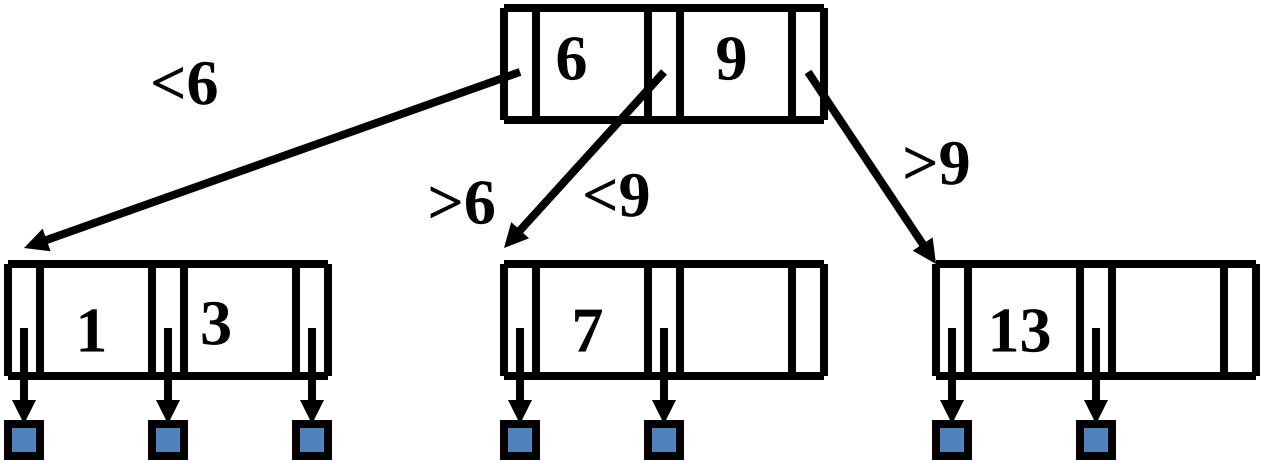
- Below is a B Tree example with order ***d = 1***



B trees are *balanced* search trees (they generalize binary trees)

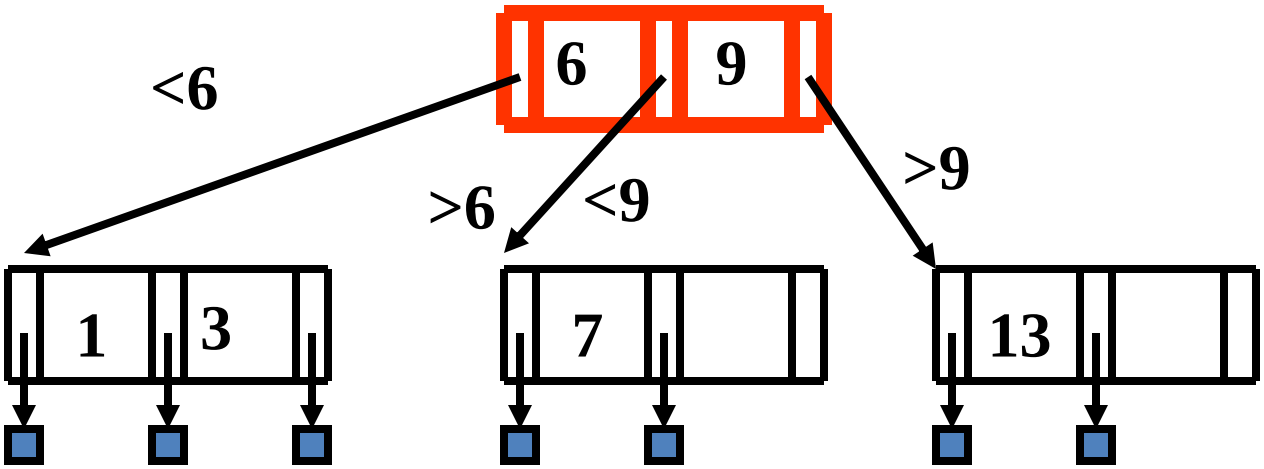
B Tree: Queries

- Algorithm for *exact match* query? (E.g., ssn=8?)



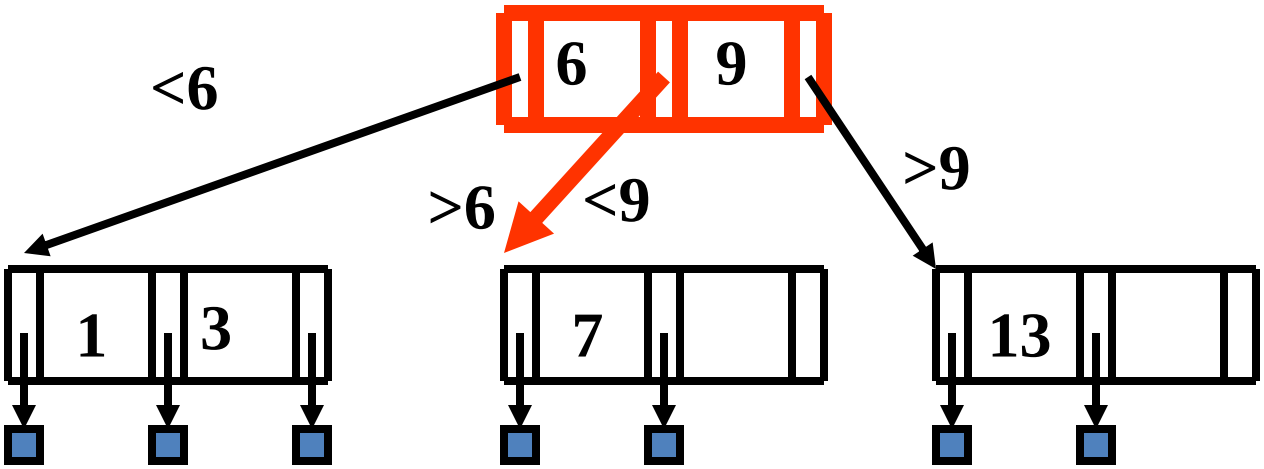
B Tree: Queries

- Algorithm for *exact match* query? (E.g., ssn=8?)



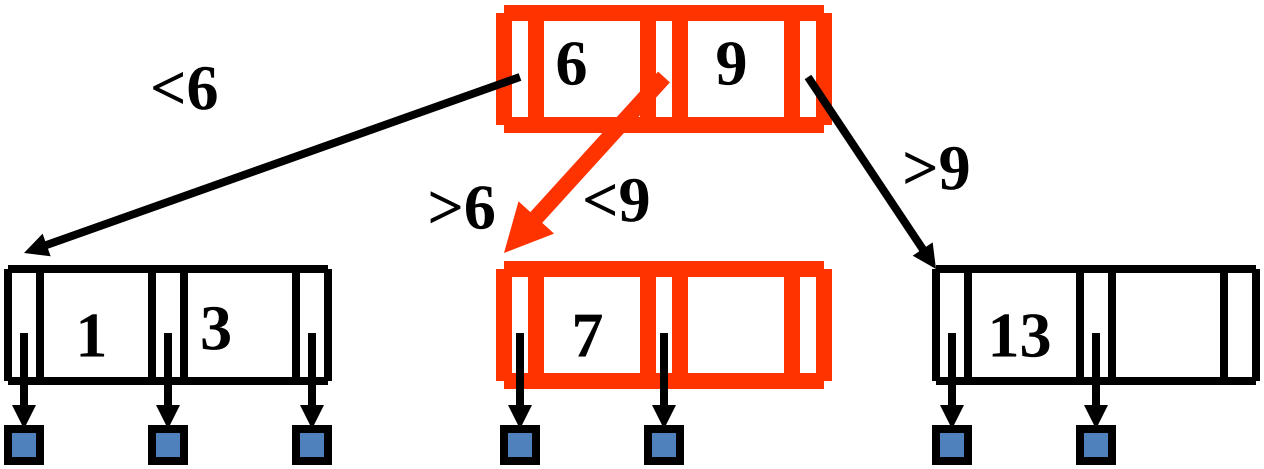
B Tree: Queries

- Algorithm for *exact match* query? (E.g., ssn=8?)



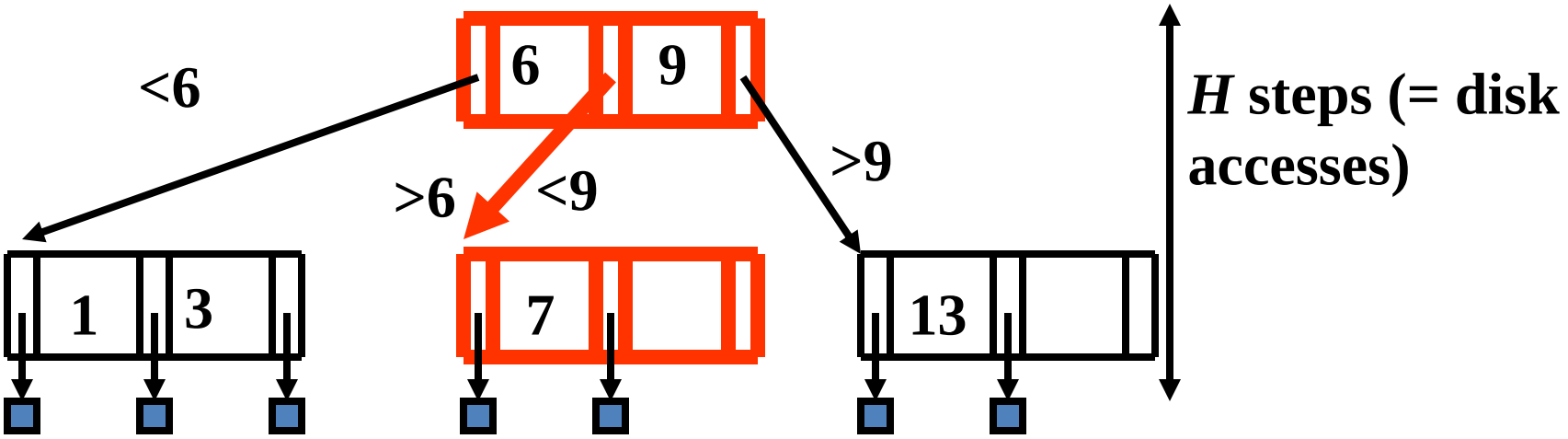
B Tree: Queries

- Algorithm for *exact match* query? (E.g., ssn=8?)



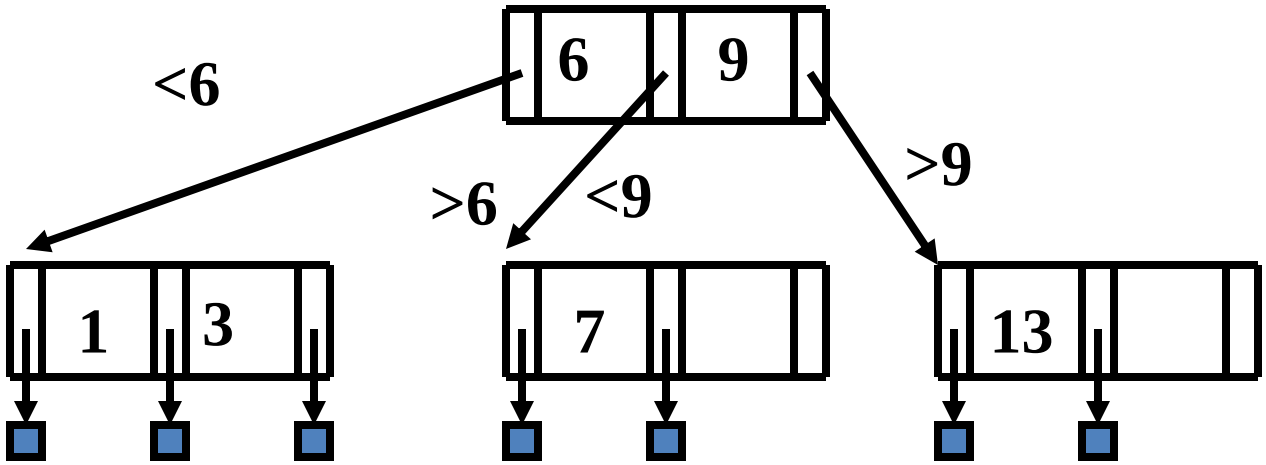
B Tree: Queries

- Algorithm for *exact match* query? (E.g., ssn=8?)



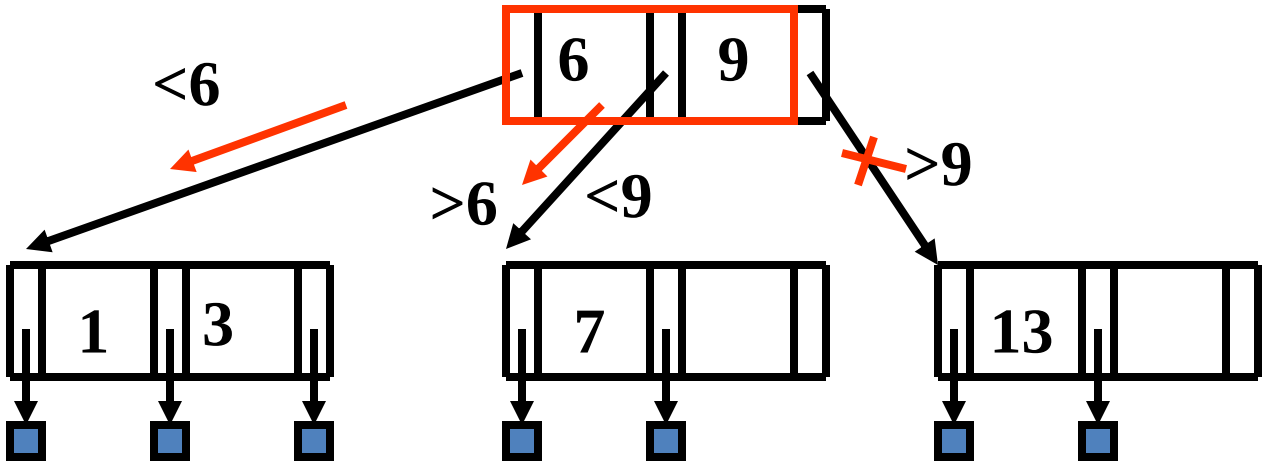
B Tree: Queries

- What about *range* queries (E.g., ***5<salary<8***)



B Tree: Queries

- What about *range* queries (E.g., ***5<salary<8***)



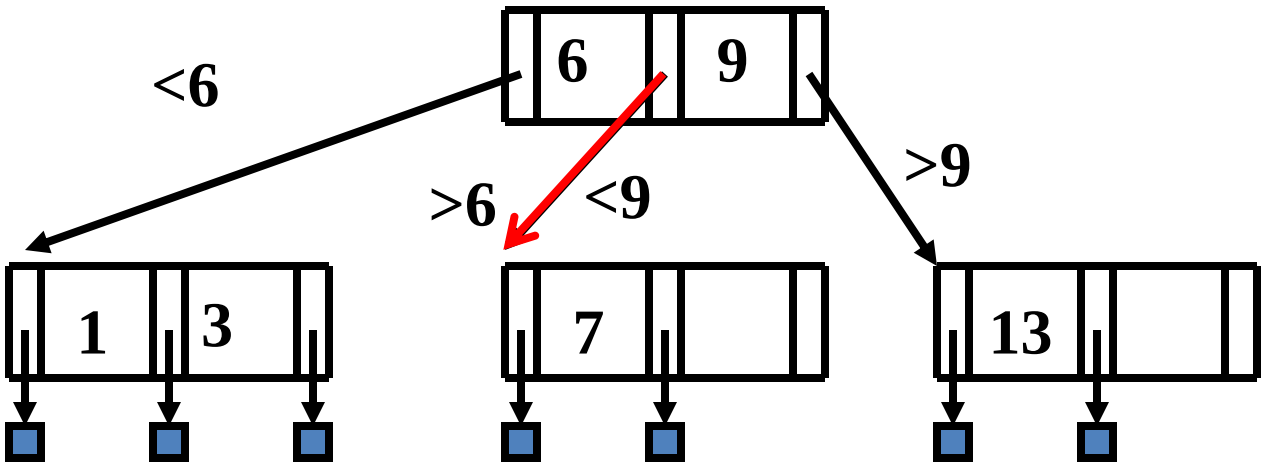
B Tree: Insertions

- Search begins at root, and key comparisons direct it to a leaf (as in ISAM)
- Insert the given entry in the located leaf
- If an overflow occurs, *split* the node and *push up* the *middle key* (recursively)

How do we define an overflow in a B tree?

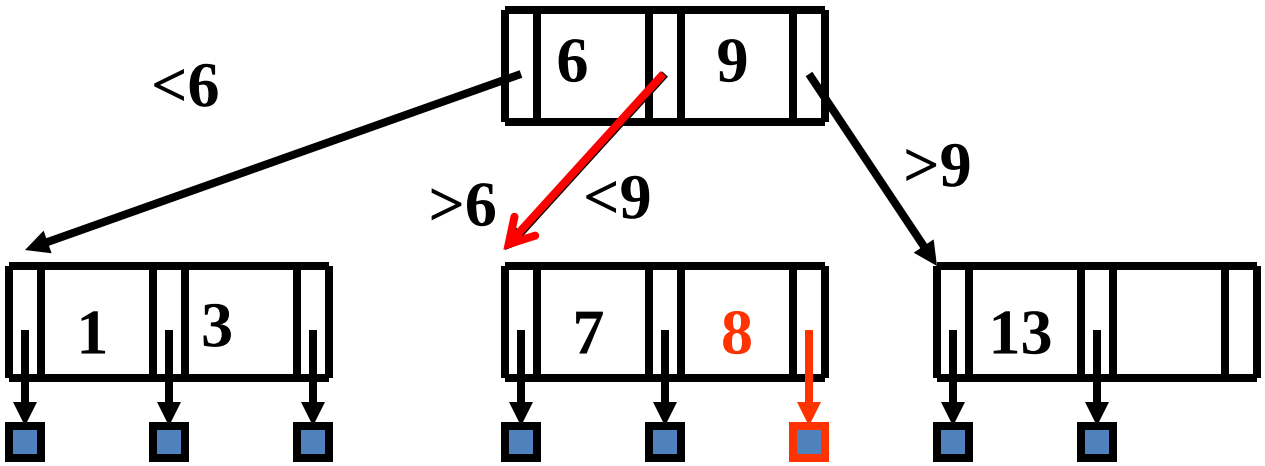
B Tree: Examples of Insertions

- Easy case: insert ‘8’



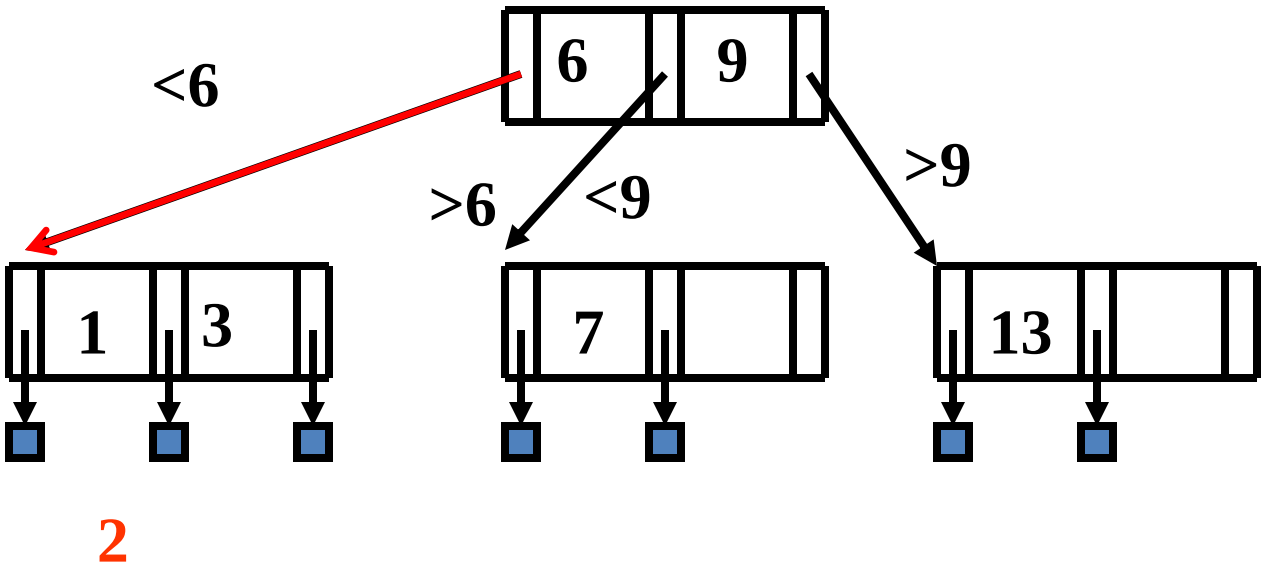
B Tree: Examples of Insertions

- Easy case: insert ‘8’



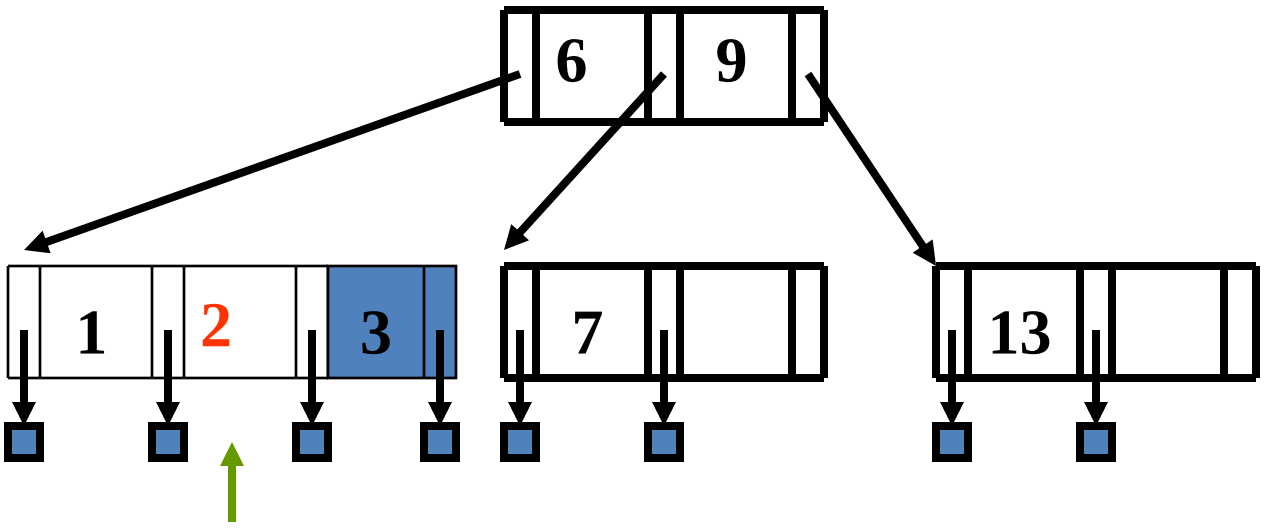
B Tree: Examples of Insertions

- Hard case: insert ‘2’



B Tree: Examples of Insertions

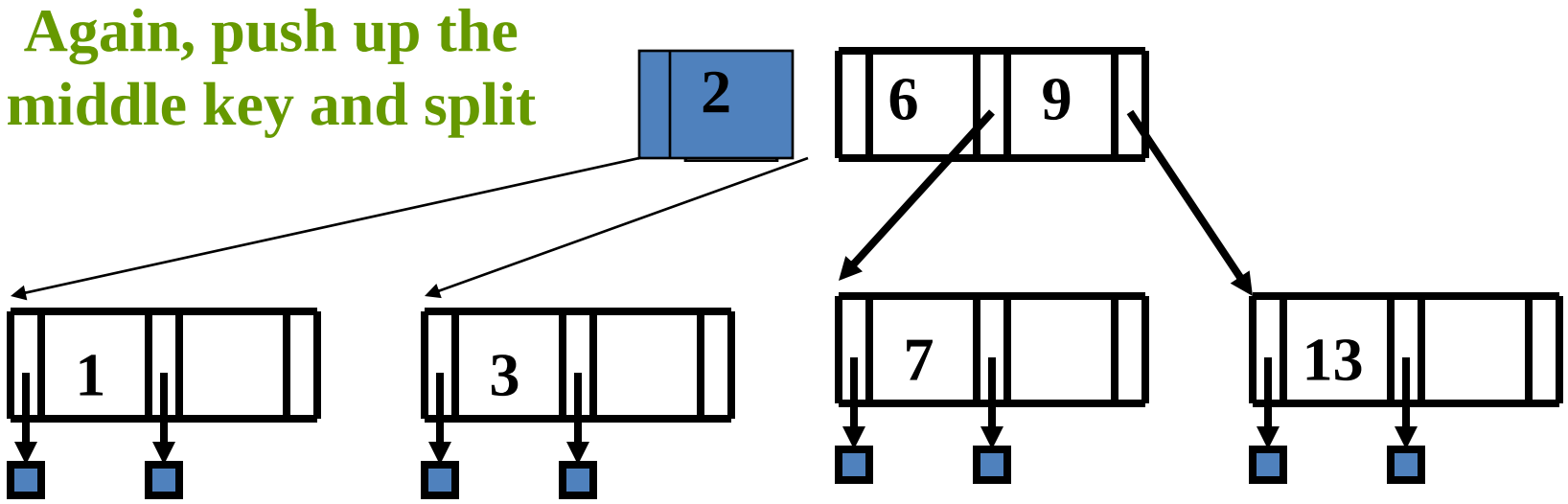
- Hard case: insert ‘2’



Push up the middle key and split

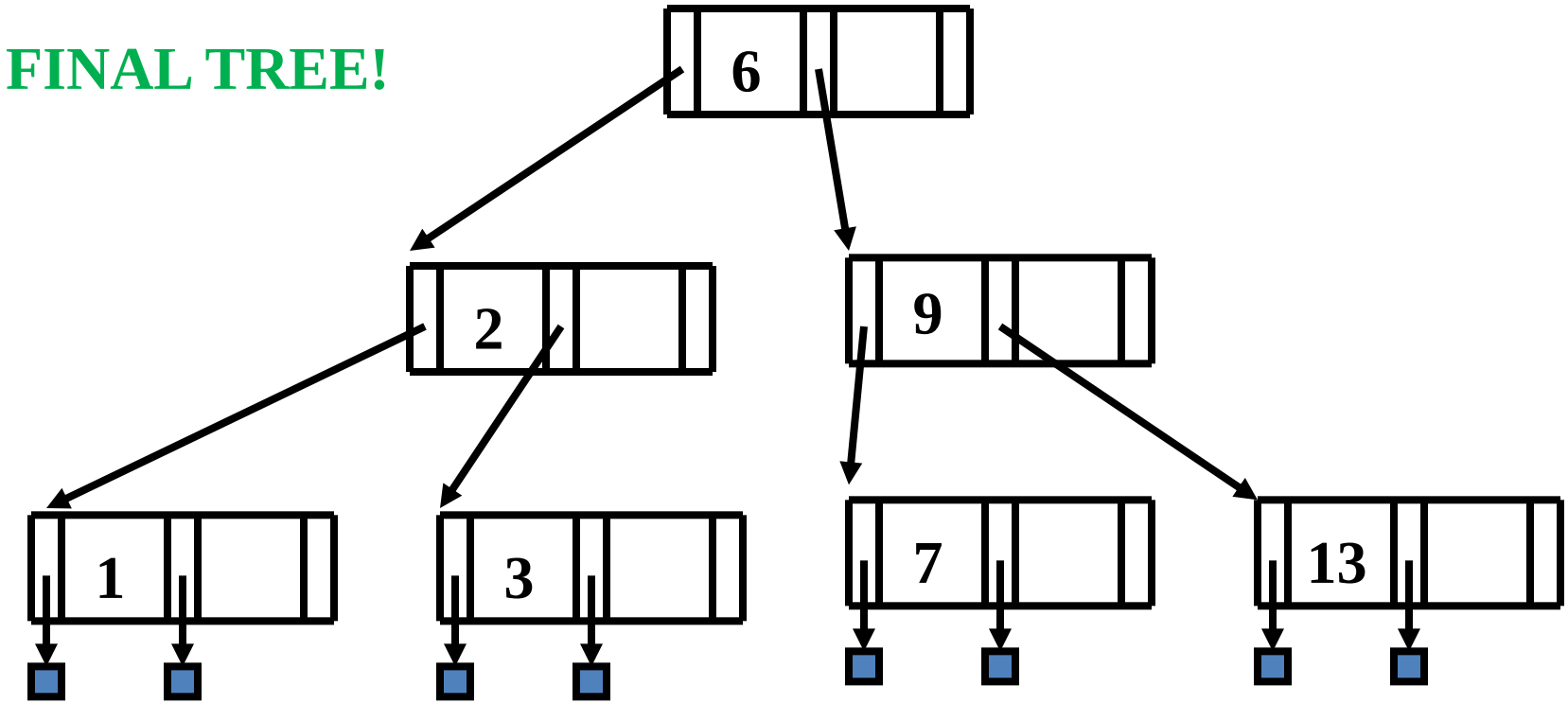
B-Tree: Examples of Insertions

- Hard case: insert ‘2’



B Tree: Examples of Insertions

- Hard case: insert ‘2’



Automatic, incremental re-organization (contrast with ISAM!)

Pseudo-code: B Tree Insertions

```
INSERTION OF KEY 'K'
    find the correct leaf node 'L';
    if ( 'L' overflows ){
        split 'L', and push middle key to parent node 'P';
        if ('P' overflows){
            repeat the split recursively; }
    else{
        add the key 'K' in node 'L';
        /* maintaining the key order in 'L' */  }
```

B Tree: Deletions

- Delete begins at root, and key comparisons direct it to a leaf (as in ISAM)
- Delete entry (*if found*) in the located leaf
- If an underflow occurs, *merge* nodes

How do we define an underflow in a B tree?

B Tree: Deletions

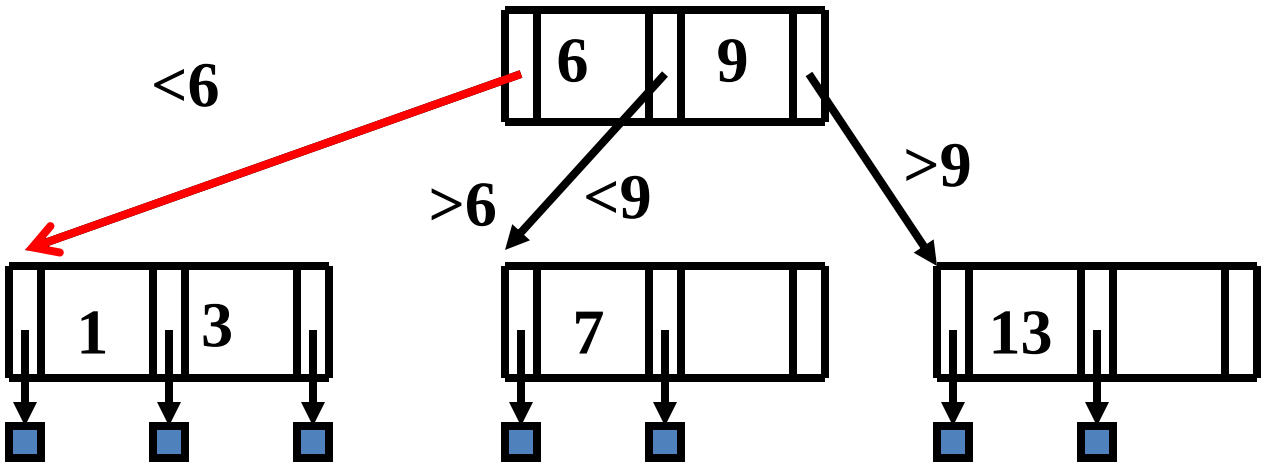
- Four cases:
  - Case1: delete a key at a leaf – no underflow
  - Case2: delete non-leaf key – no underflow
  - Case3: delete leaf-key; underflow, and ‘rich sibling’
  - Case4: delete leaf-key; underflow, and ‘poor sibling’

B Tree: Deletions

- Four cases:
  - Case1: delete a key at a leaf – no underflow
  - Case2: delete non-leaf key – no underflow
  - Case3: delete leaf-key; underflow, and ‘rich sibling’
  - Case4: delete leaf-key; underflow, and ‘poor sibling’

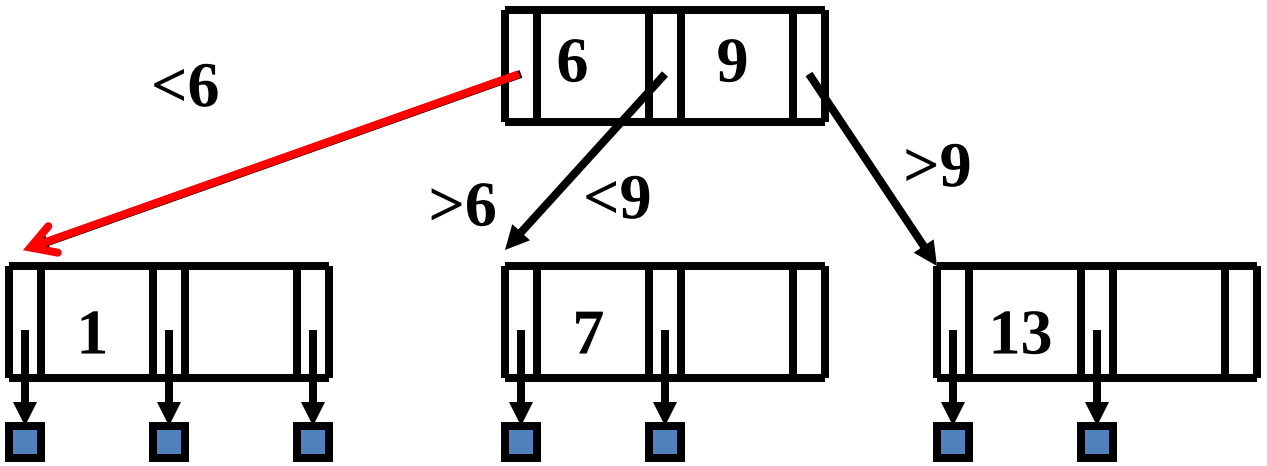
Examples of Deletions: *Case 1*

- Easy case: delete ‘3’



Examples of Deletions: *Case 1*

- Easy case: delete ‘3’

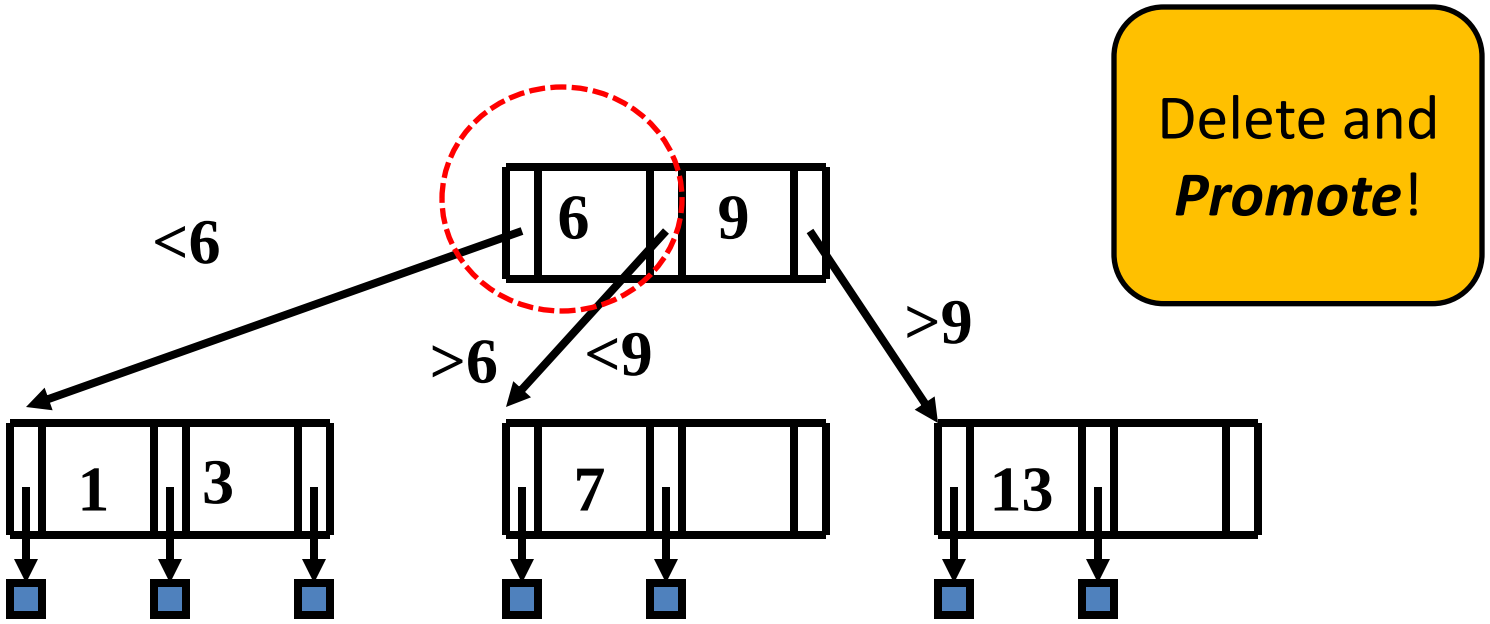


B-Tree: Deletions

- Four cases:
  - Case1: delete a key at a leaf – no underflow
  - Case2: delete non-leaf key – no underflow
  - Case3: delete leaf-key; underflow, and ‘rich sibling’
  - Case4: delete leaf-key; underflow, and ‘poor sibling’

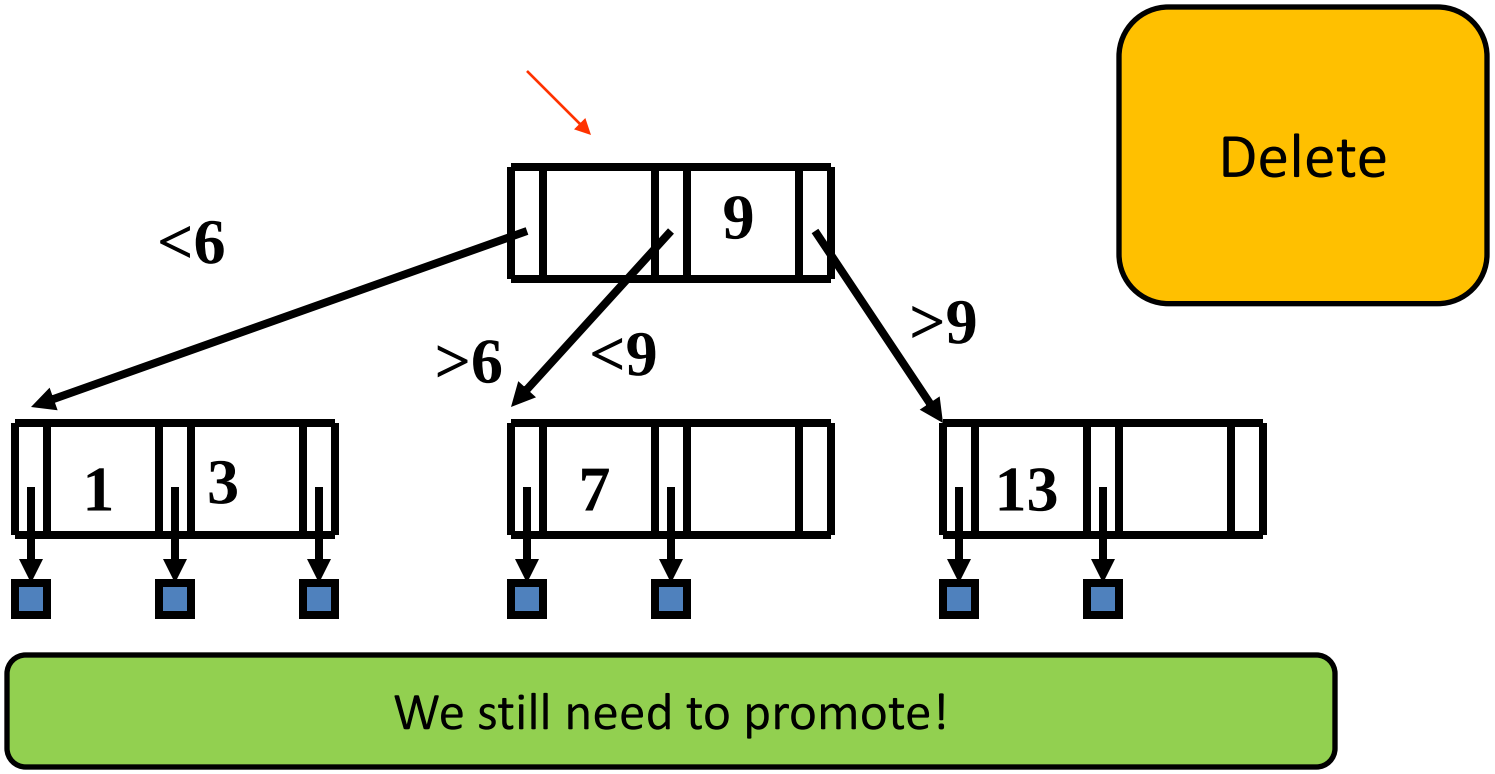
Examples of Deletions: *Case 2*

- Delete ‘6’



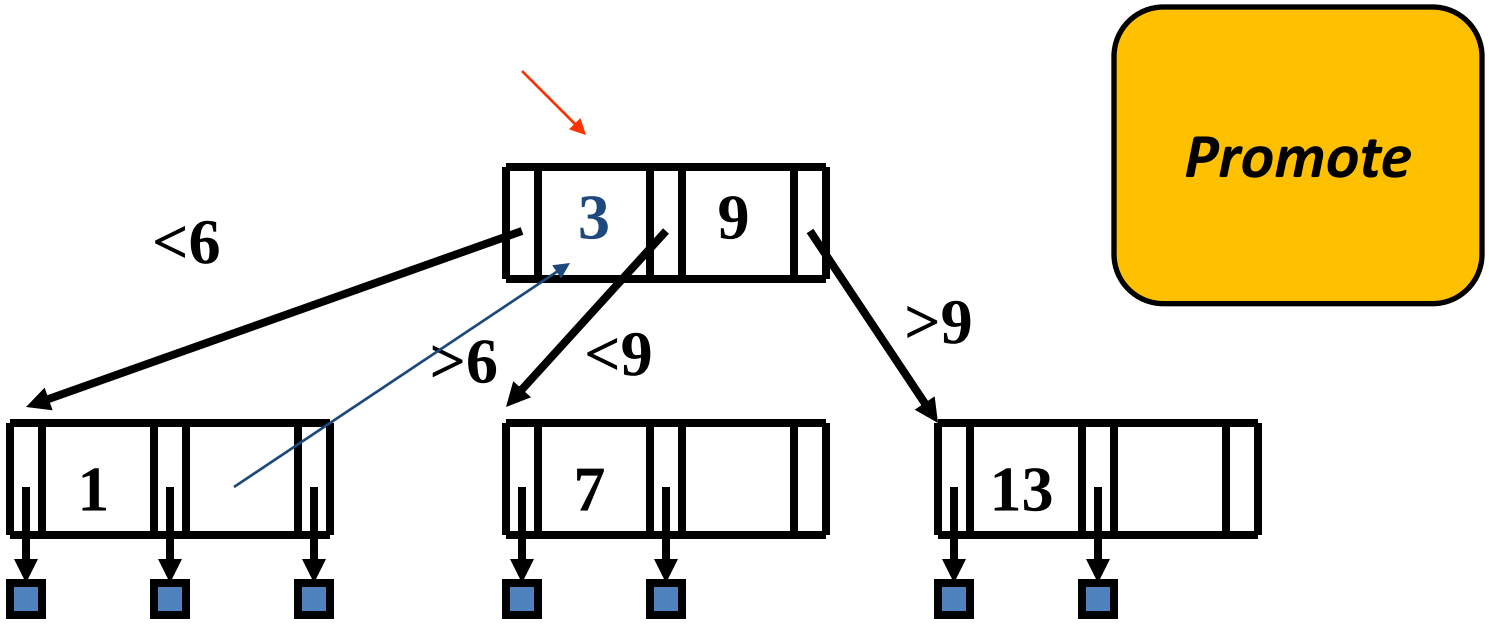
Examples of Deletions: *Case 2*

- Delete ‘6’



Examples of Deletions: *Case 2*

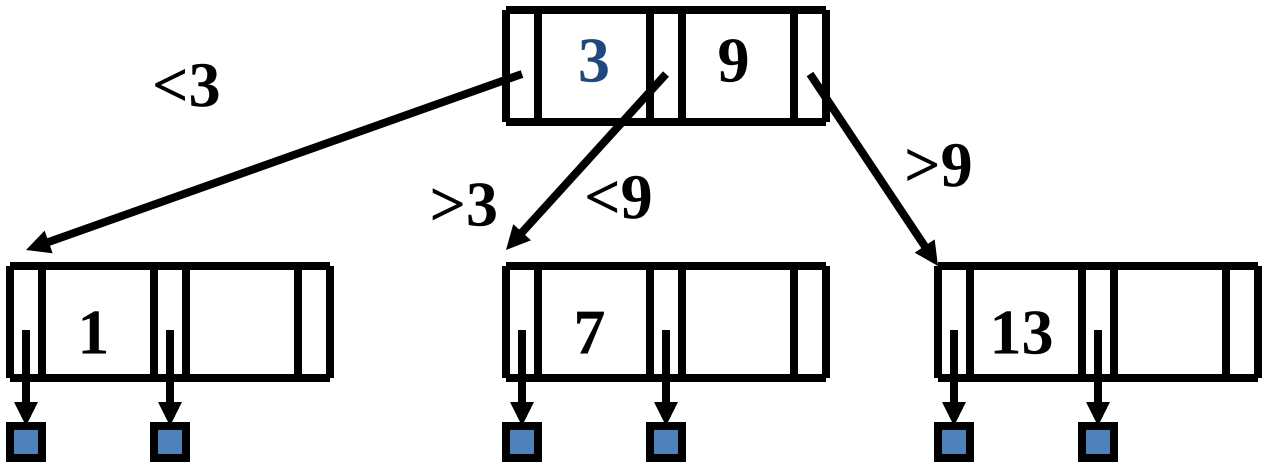
- Delete ‘6’



Examples of Deletions: Case 2

- Delete ‘6’

FINAL TREE!

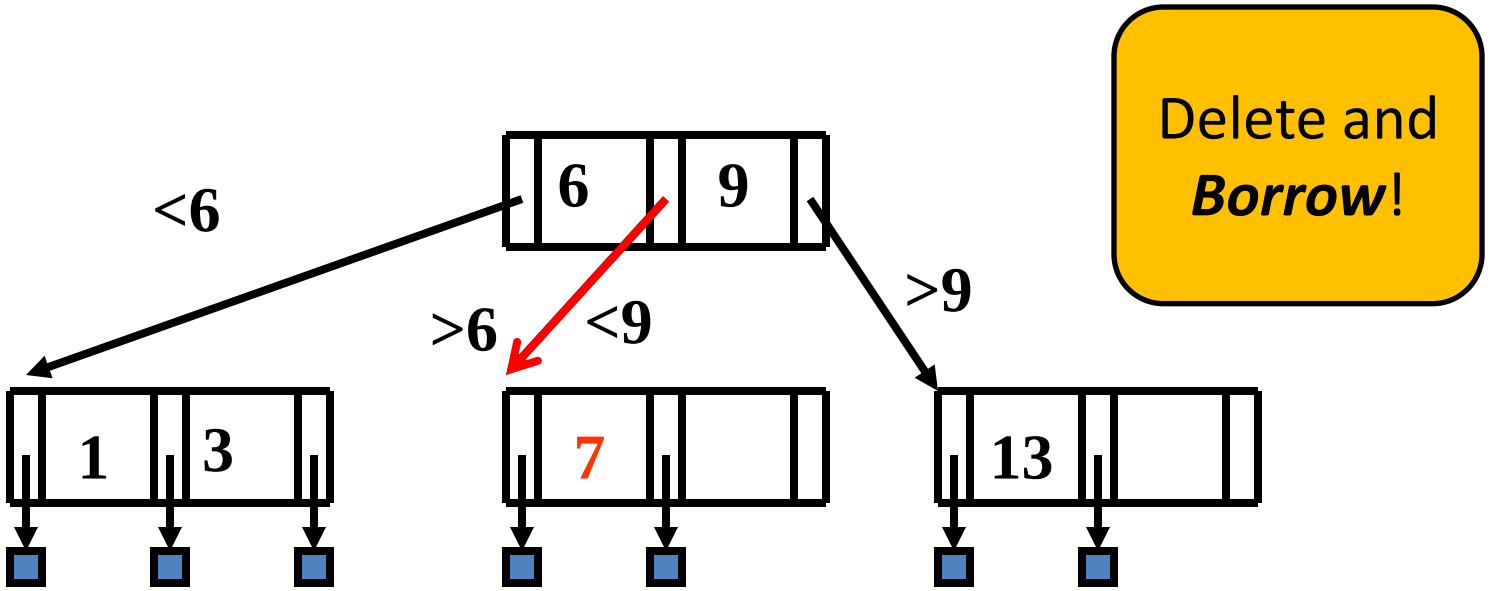


B Tree: Deletions

- Four cases:
  - Case1: delete a key at a leaf – no underflow
  - Case2: delete non-leaf key – no underflow
  - Case3: delete leaf-key; underflow, and ‘rich sibling’
  - Case4: delete leaf-key; underflow, and ‘poor sibling’

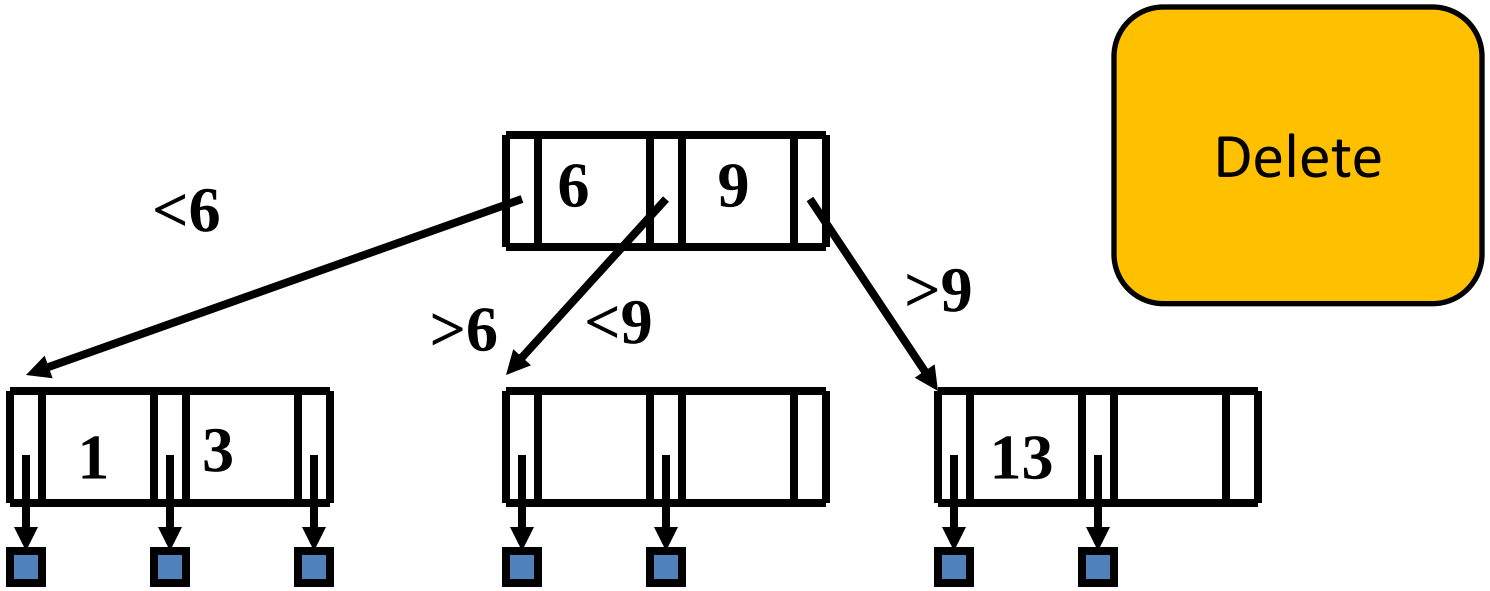
Examples of Deletions: *Case 3*

- Delete ‘7’



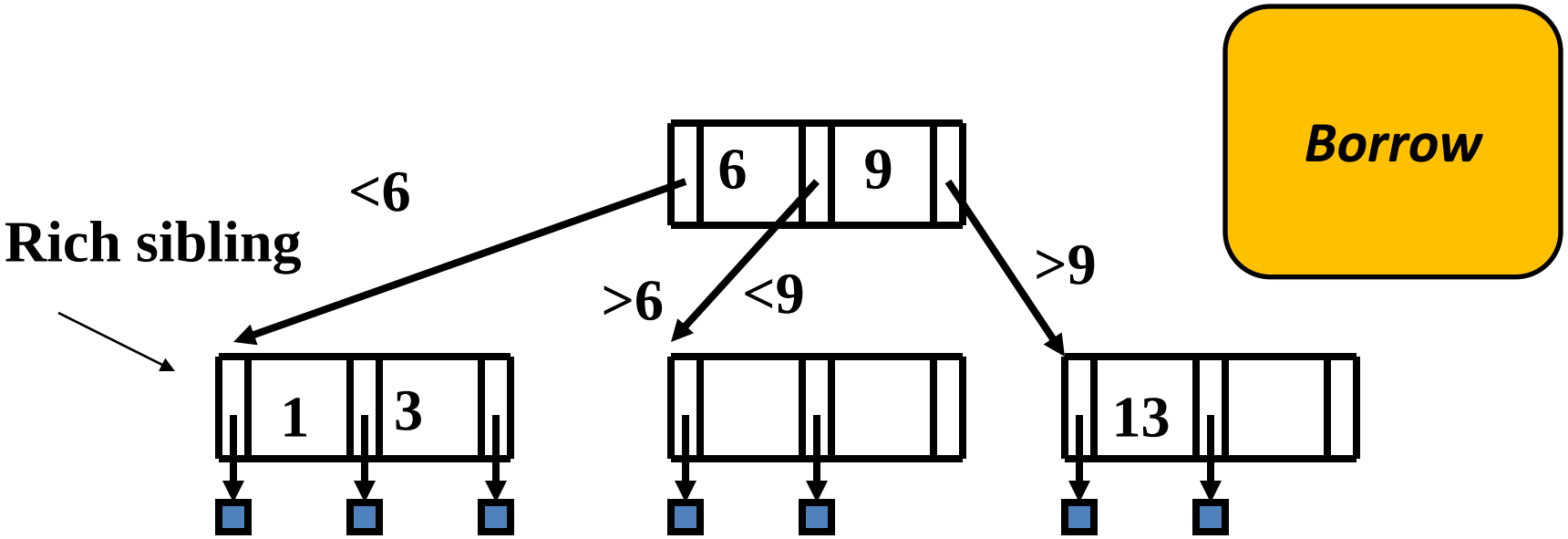
Examples of Deletions: *Case 3*

- Delete ‘7’



Examples of Deletions: *Case 3*

- Delete ‘7’

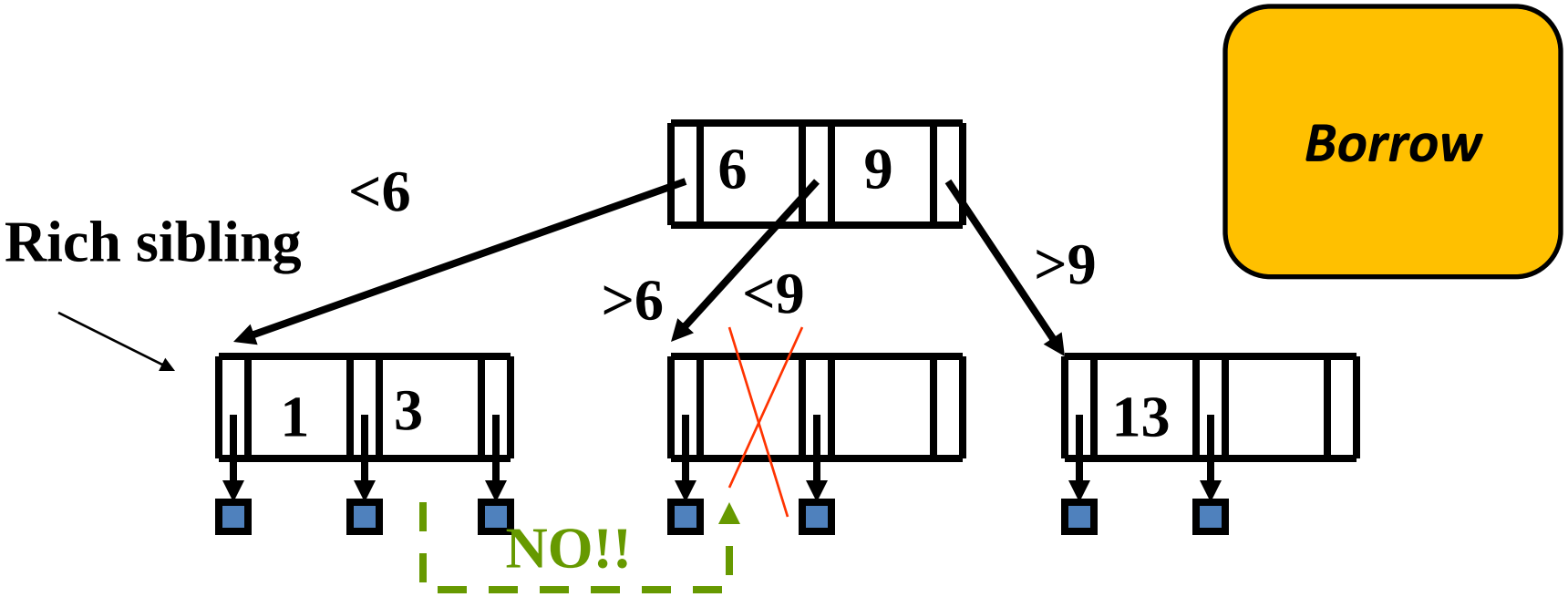


‘Rich’ = can lend a key, without under-flowing

‘Borrow’ = can happen ONLY through the parent

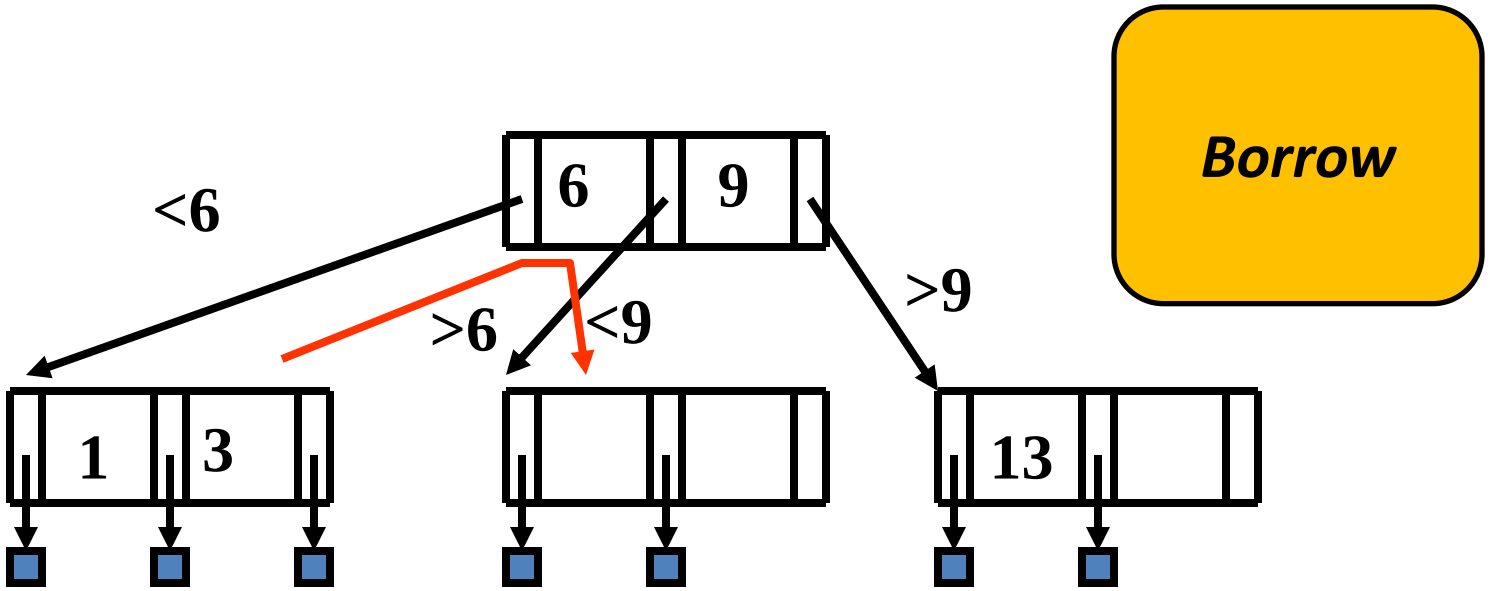
Examples of Deletions: *Case 3*

- Delete ‘7’



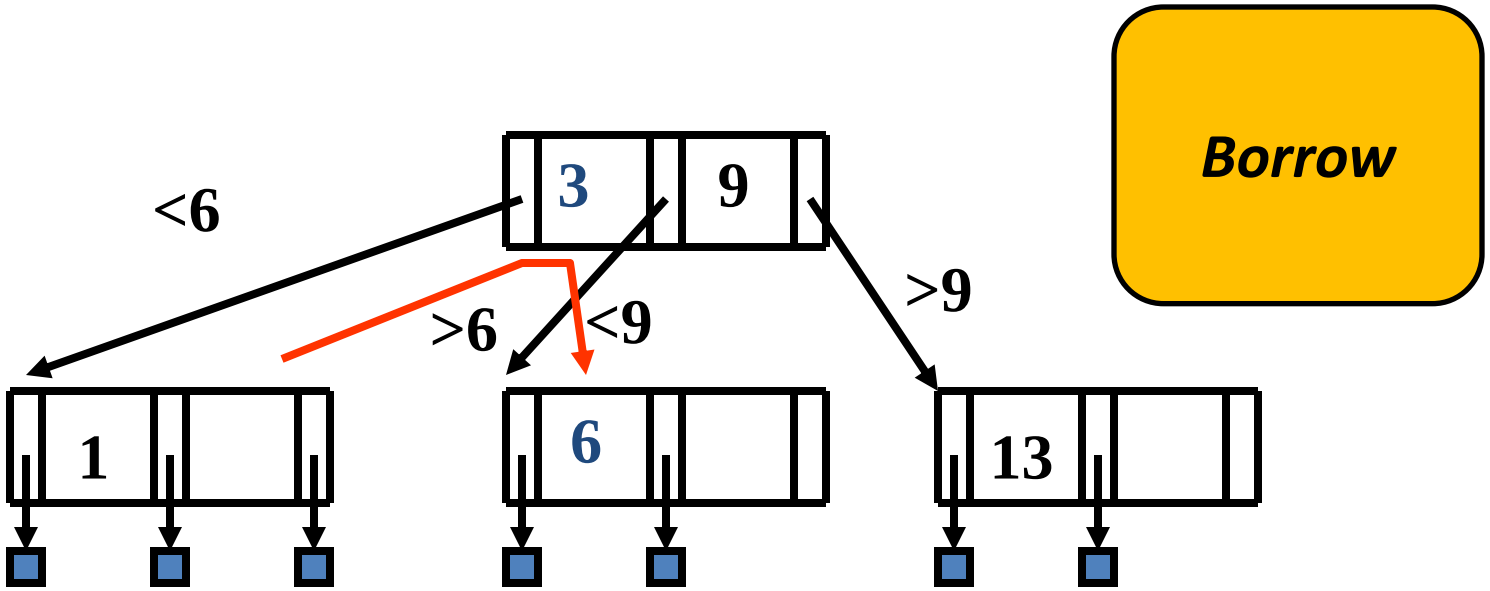
Examples of Deletions: *Case 3*

- Delete ‘7’



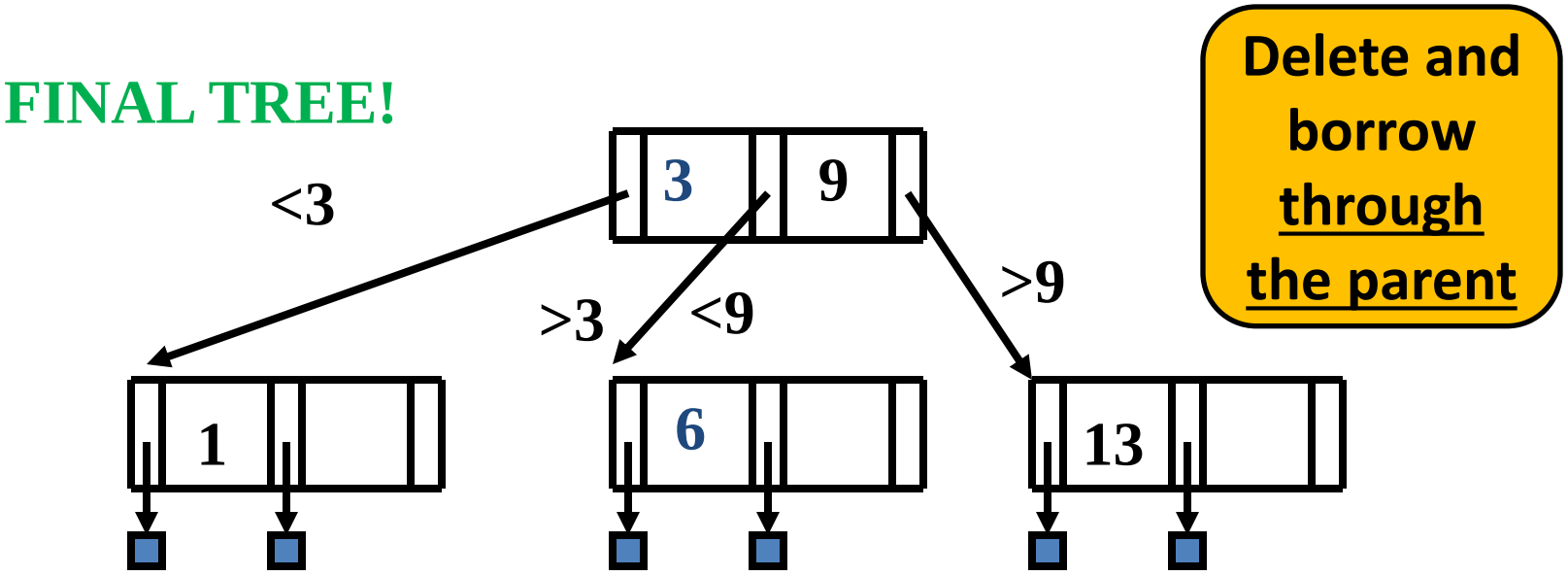
Examples of Deletions: *Case 3*

- Delete ‘7’



Examples of Deletions: *Case 3*

- Delete ‘7’

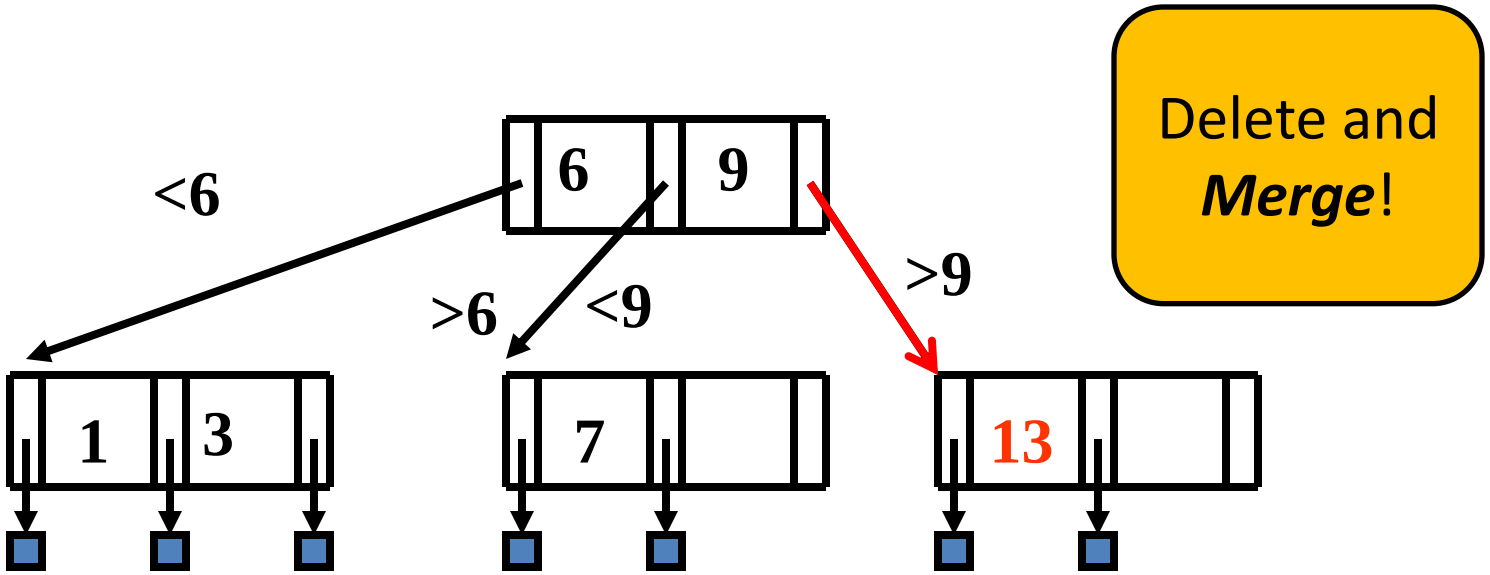


B Tree: Deletions

- Four cases:
  - Case1: delete a key at a leaf – no underflow
  - Case2: delete non-leaf key – no underflow
  - Case3: delete leaf-key; underflow, and ‘rich sibling’
  - Case4: delete leaf-key; underflow, and ‘poor sibling’

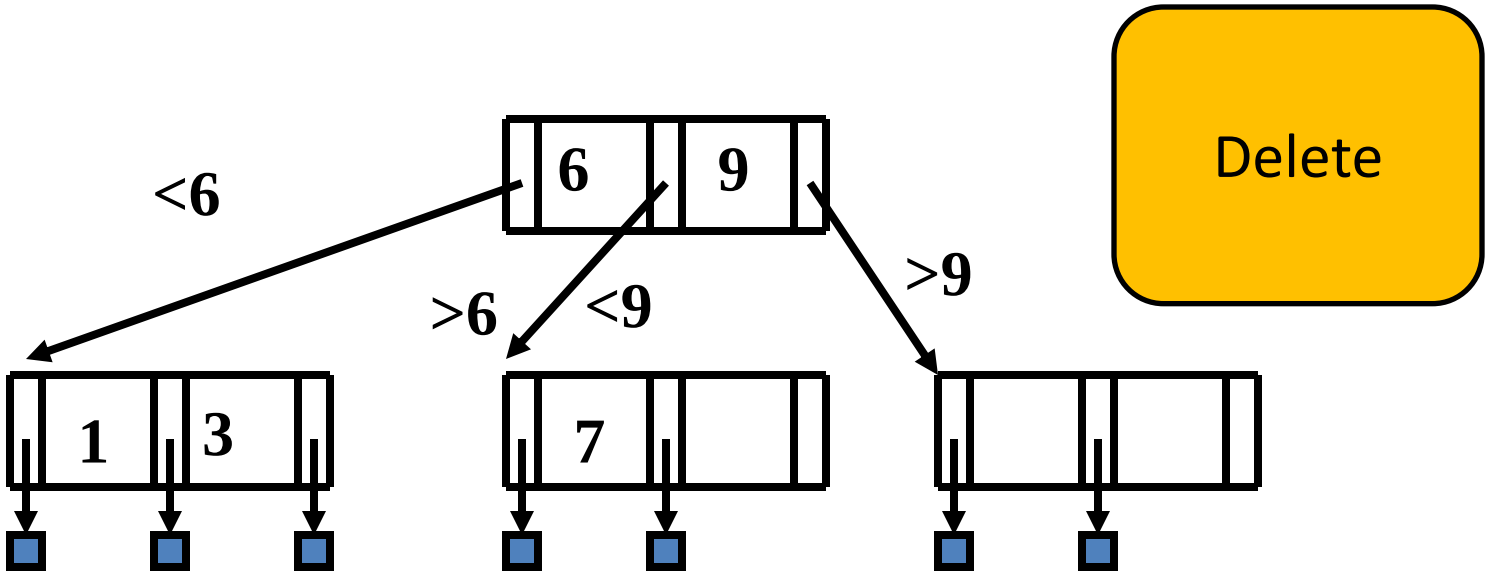
Examples of Deletions: *Case 4*

- Delete '13'



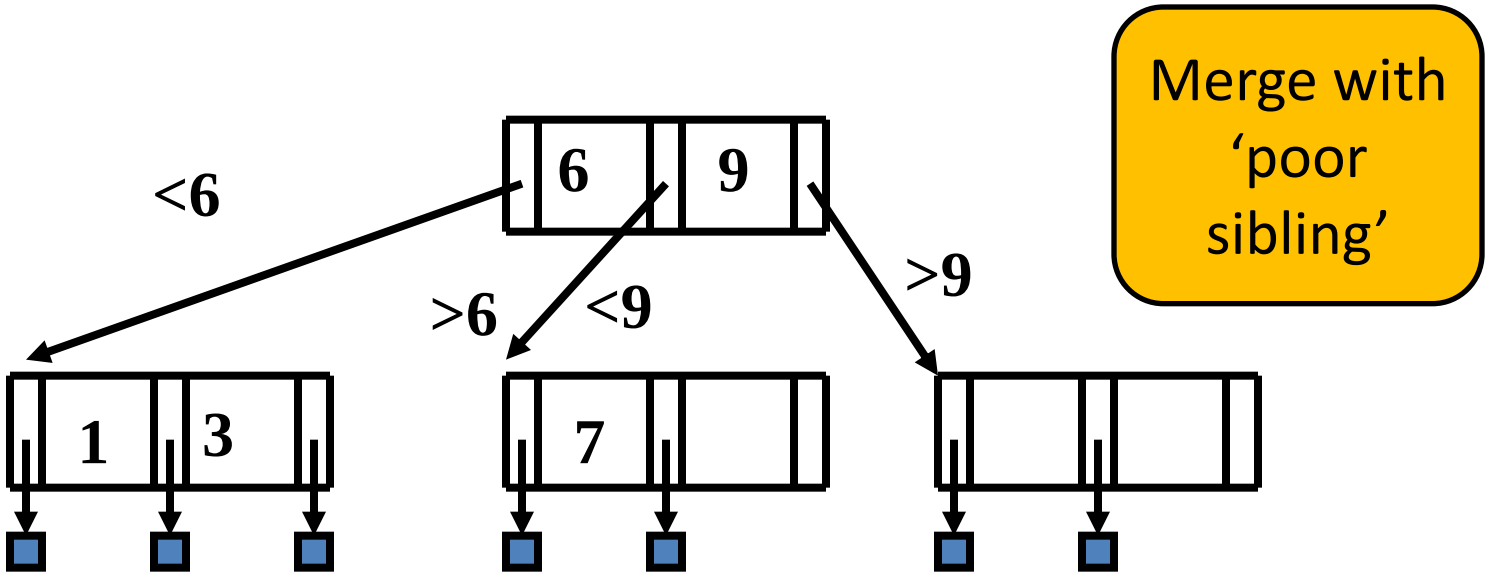
Examples of Deletions: *Case 4*

- Delete '13'



Examples of Deletions: *Case 4*

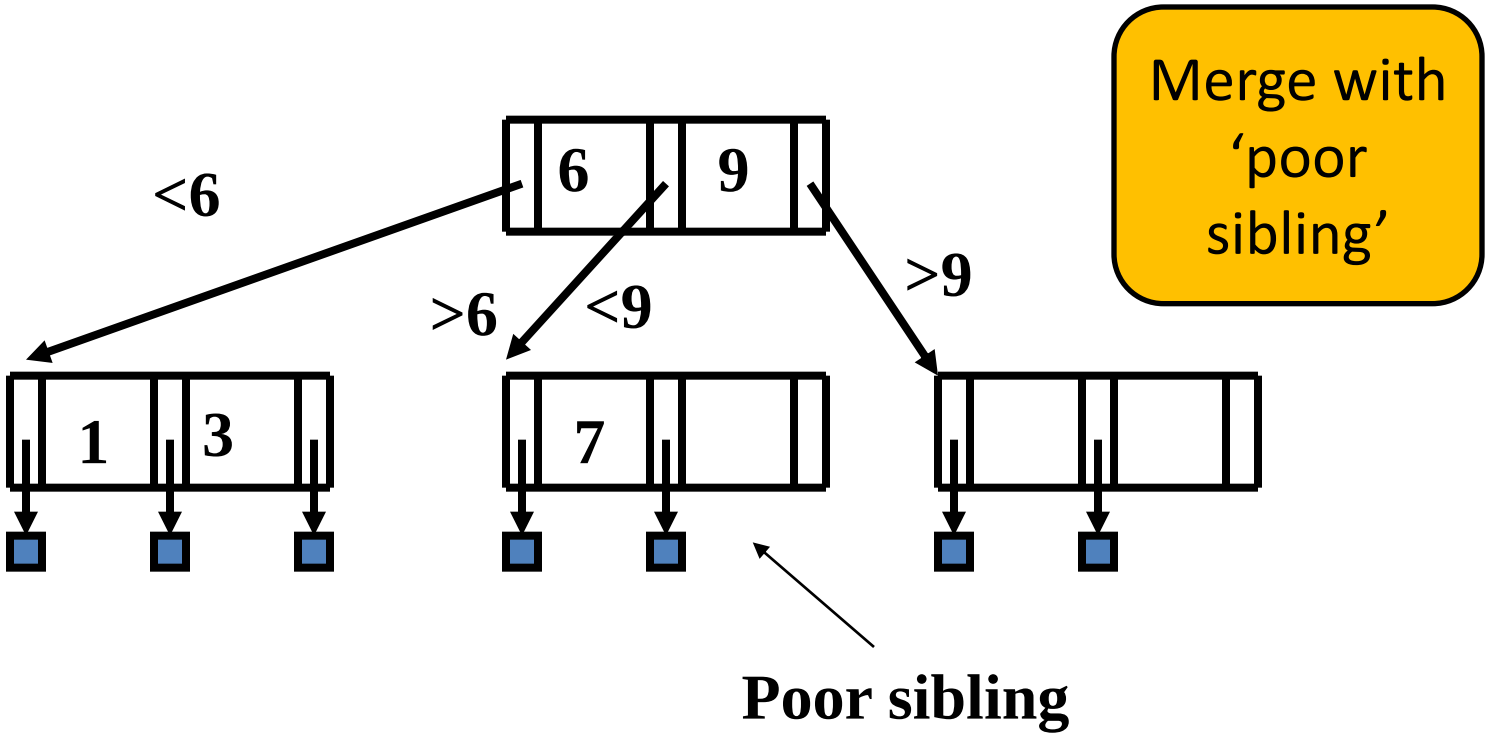
- Delete ‘13’



‘Poor’ = can host a key, without overflowing

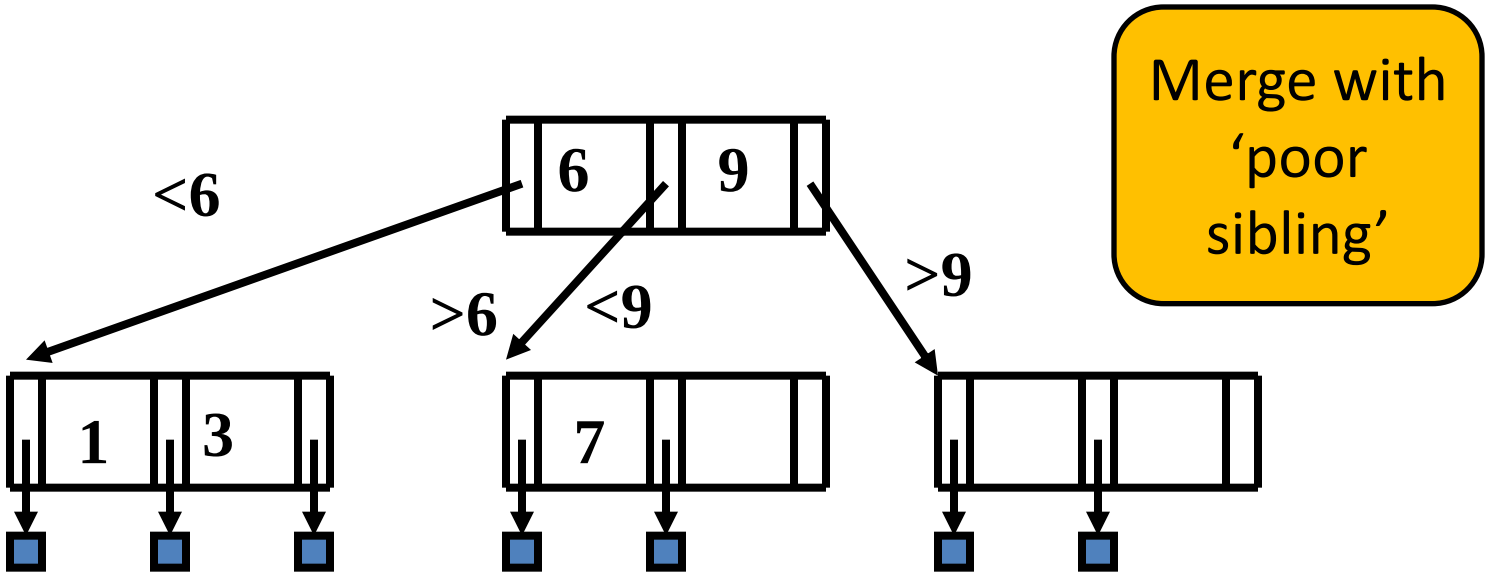
Examples of Deletions: *Case 4*

- Delete '13'



Examples of Deletions: *Case 4*

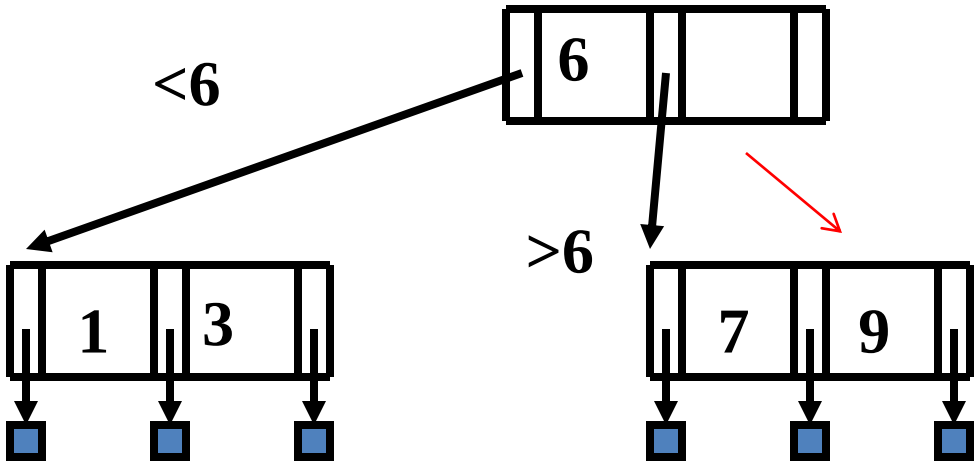
- Delete '13'



Merge, by pulling a key from the **parent** (*the opposite of insertions!*)

Examples of Deletions: *Case 4*

- Delete '13'

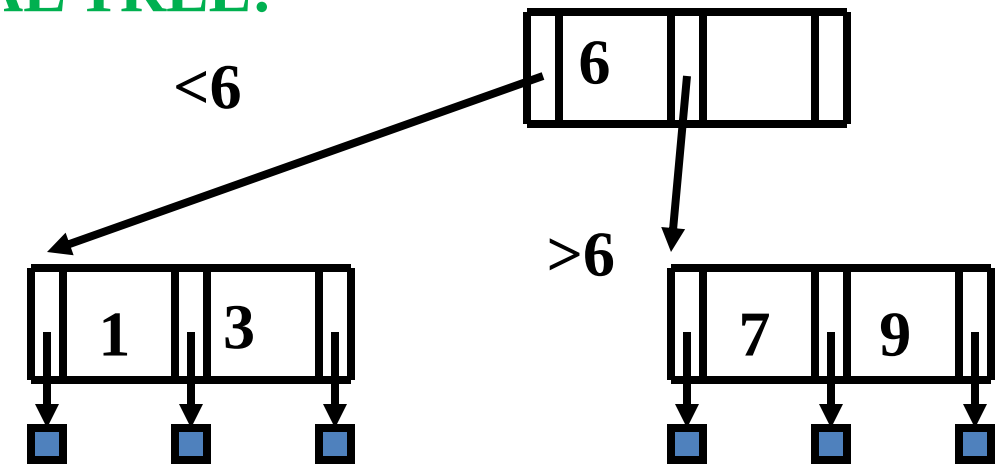


Merge & Pull from Parent

Examples of Deletions: *Case 4*

- Delete '13'

FINAL TREE!



Merge with 'poor sibling'

But, what if the parent underflows?

Repeat recursively!

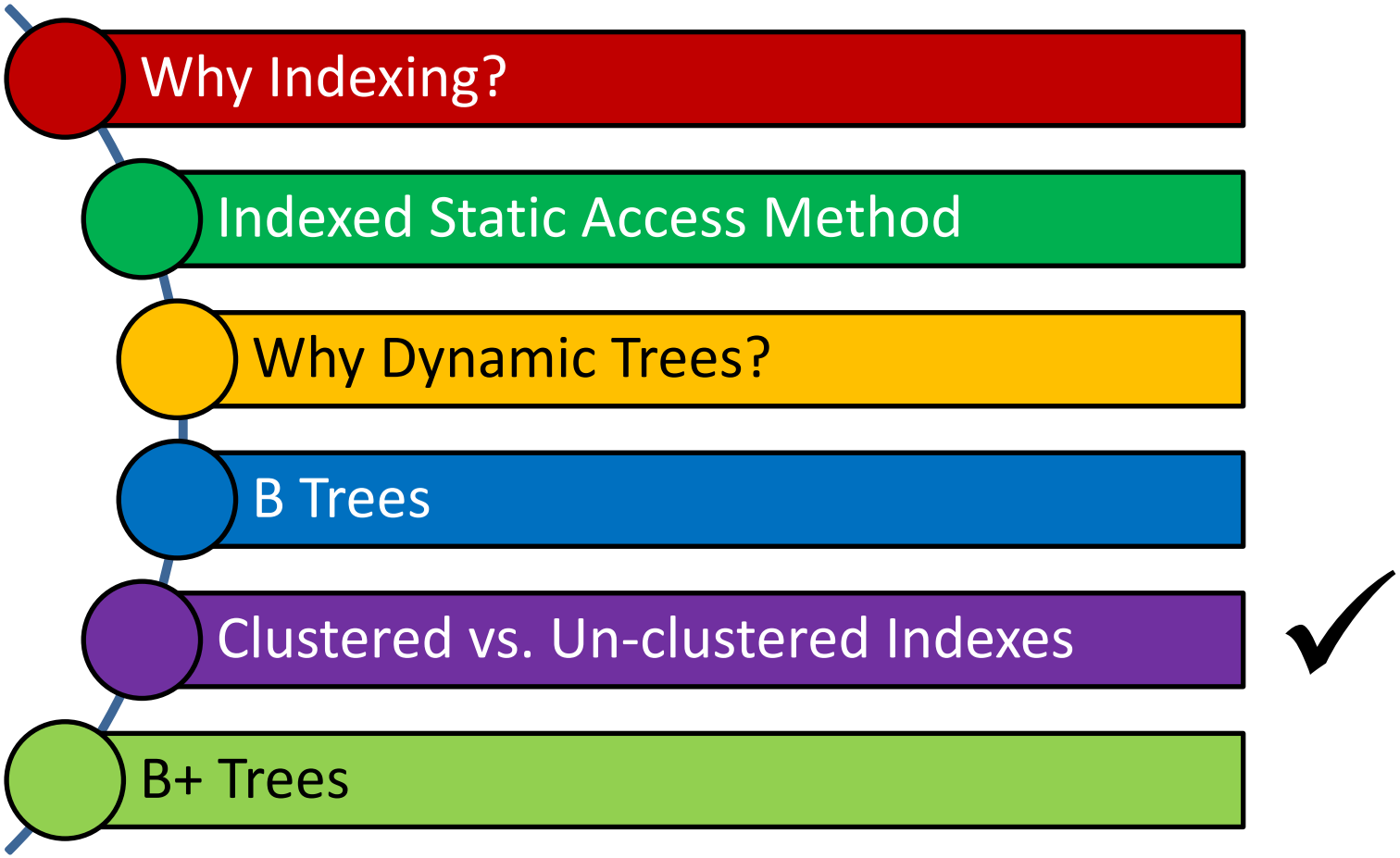
Pseudo-code: B Tree Deletions

```
DELETION OF KEY 'K'
   locate key 'K', in node 'N'
   if( 'N' is a non-leaf node) {
      delete 'K' from 'N';
      find the immediately largest key 'K1';
         /* which is guaranteed   to be on a leaf node 'L' */
      copy 'K1' in the old position of 'K';
      invoke this DELETION routine on 'K1' from the leaf node
'L';
   else {
/* 'N' is a leaf node */
... (next slide..)
```

Pseudo-code: B Tree Deletions

```
/* 'N' is a leaf node */
      if( 'N' underflows ){
         let 'N1' be the sibling of 'N';
         if( 'N1' is "rich"){   /* ie., N1 can lend us a key */
            borrow a key from 'N1'  THROUGH the parent node;
         }else{    /* N1 is 1 key away from underflowing */
            MERGE: pull the key from the parent 'P',
               and merge it with the keys of 'N' and 'N1'  into a
new node;
            if( 'P' underflows){ repeat recursively }
         }
   }
```

Outline



Clustered vs. Un-clustered Indexes

- Indexes can be either clustered or un-clustered

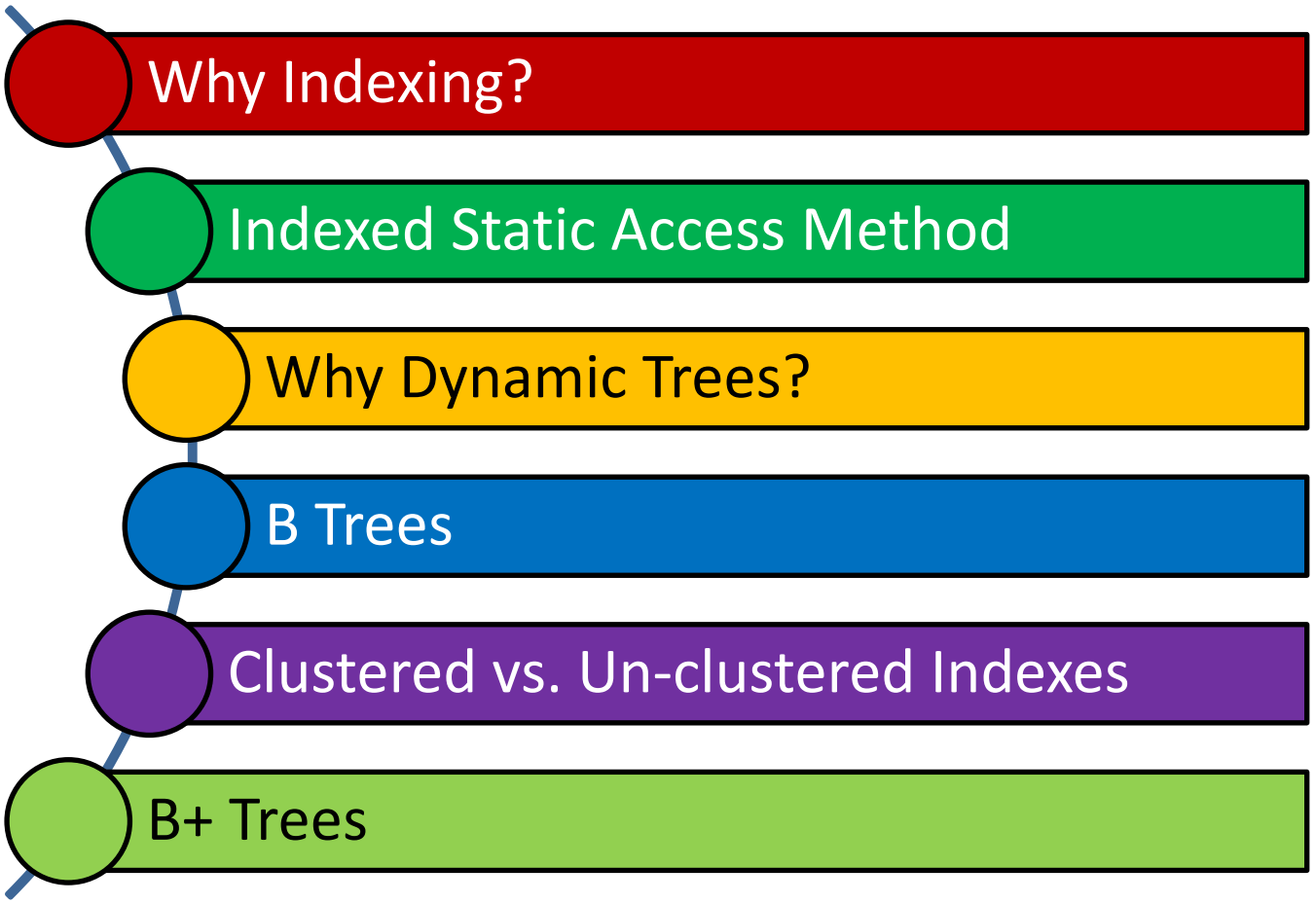
- Clustered Indexes:
  - When the ordering of data records is the same as (or close to) the ordering of entries in some index

- Un-clustered Indexes:
  - When the ordering of data records differs from the ordering of entries in some index

Clustered vs. Un-clustered Indexes

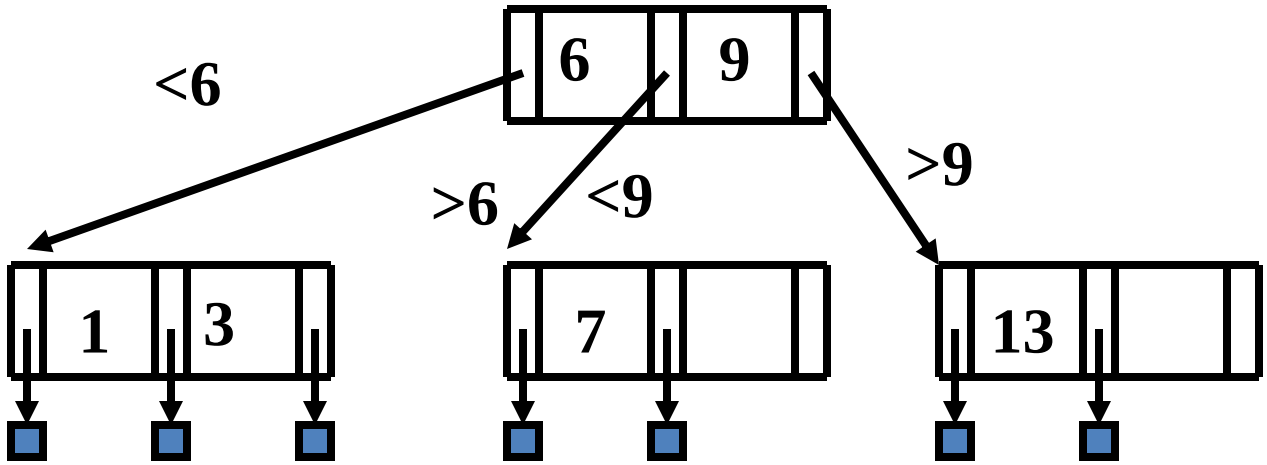
- Is an index that uses Alternative (1) clustered or un-clustered?
  - Clustered
- Is an index that uses Alternatives (2) or (3) clustered or un-clustered?
  - Clustered if data records are sorted on the search key field
- In practice:
  - A clustered index is an index that uses Alternative (1)
  - Indexes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

Outline



B+ Trees: Motivation

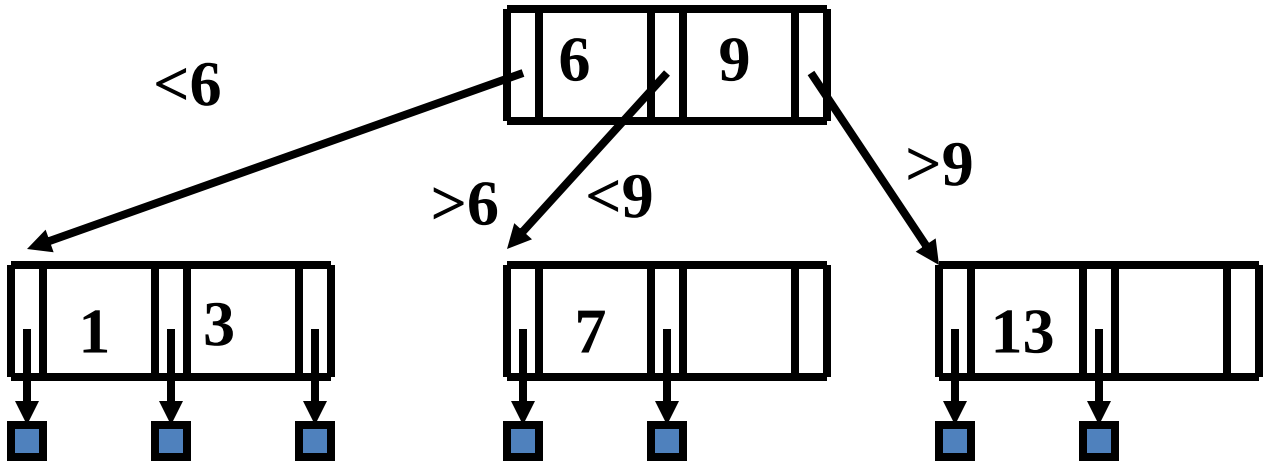
- For clustering indexes, data records are *scattered*



IDEA 1: *replicate* keys from non-leaf nodes, to make sure every key appears at only one leaf node!

B+ Trees: Motivation

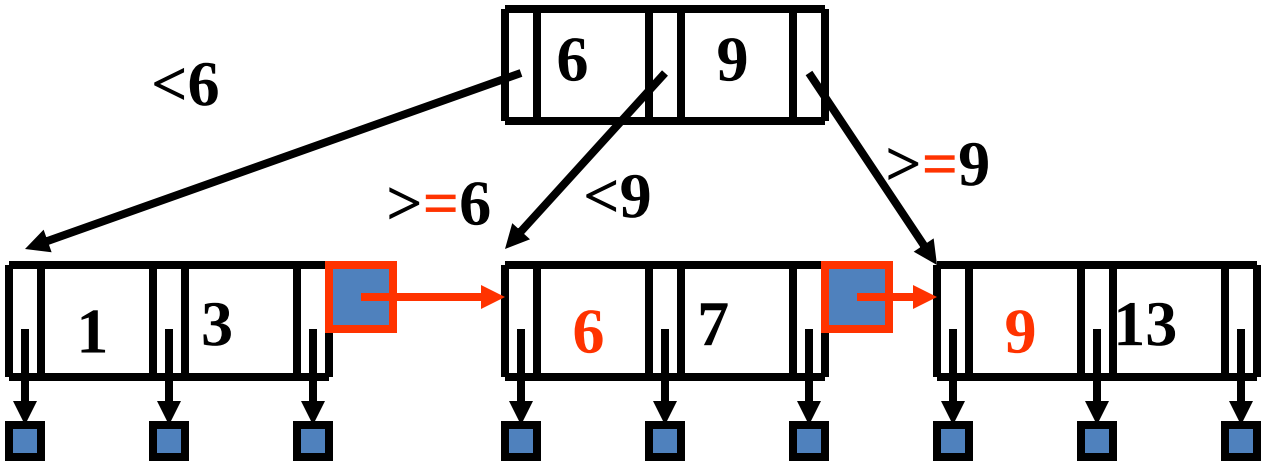
- How can we facilitate sequential operations?



IDEA 2: String all leaf nodes together!

B+ Trees: Motivation

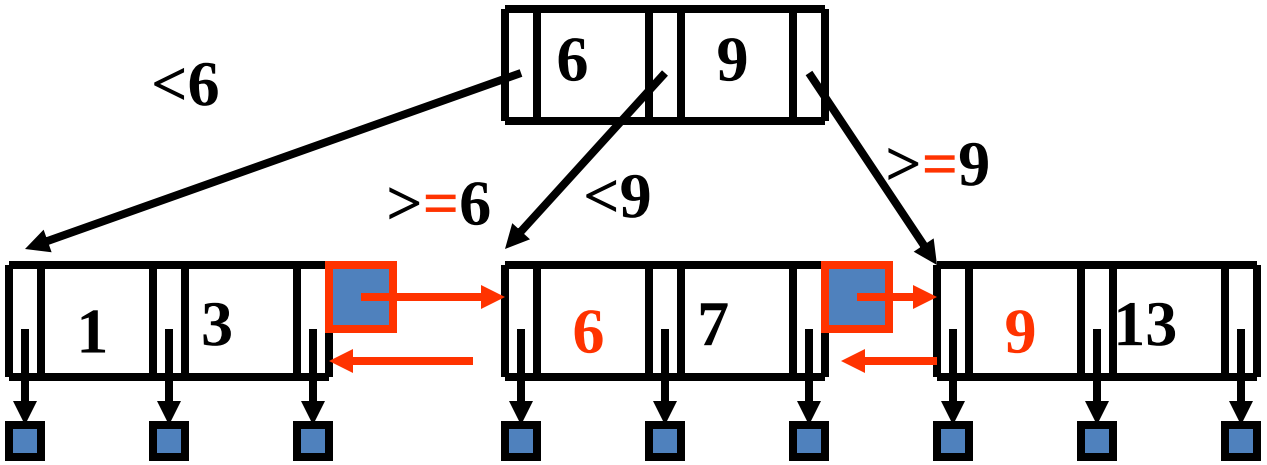
- Towards B+ trees



Is this enough?

B+ Trees: Motivation

- B+ trees

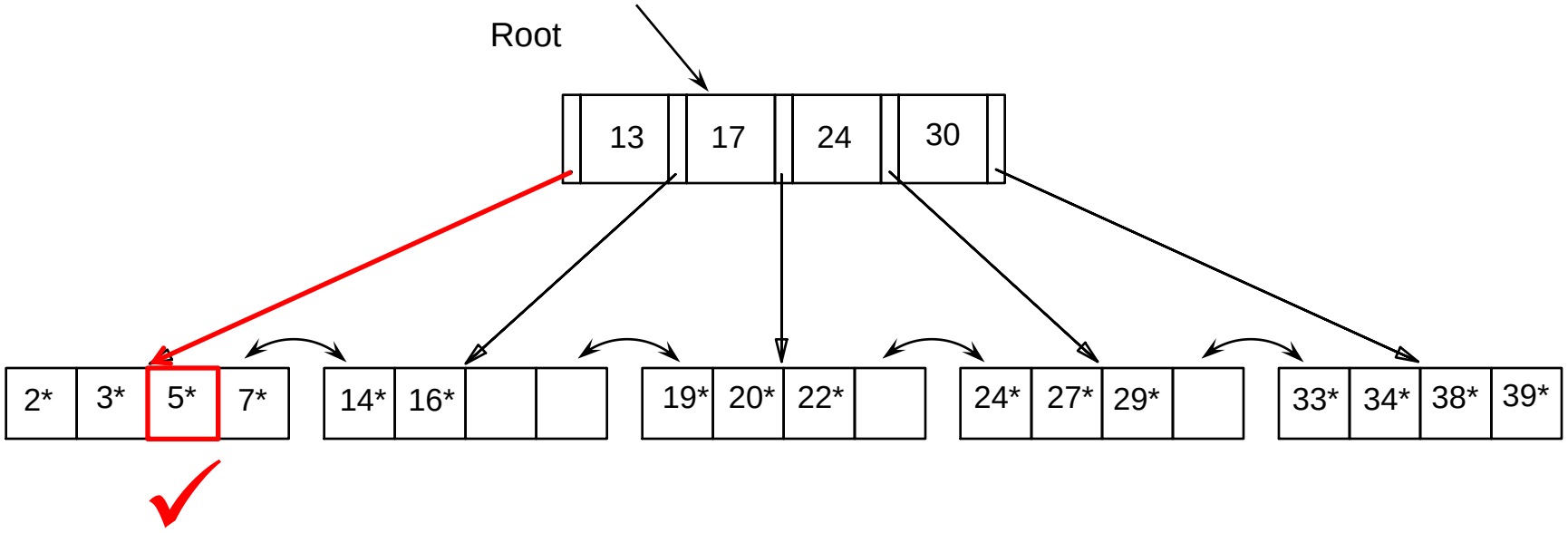


Vital for clustering indexes!

B+ Tree: Searching for Entries

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

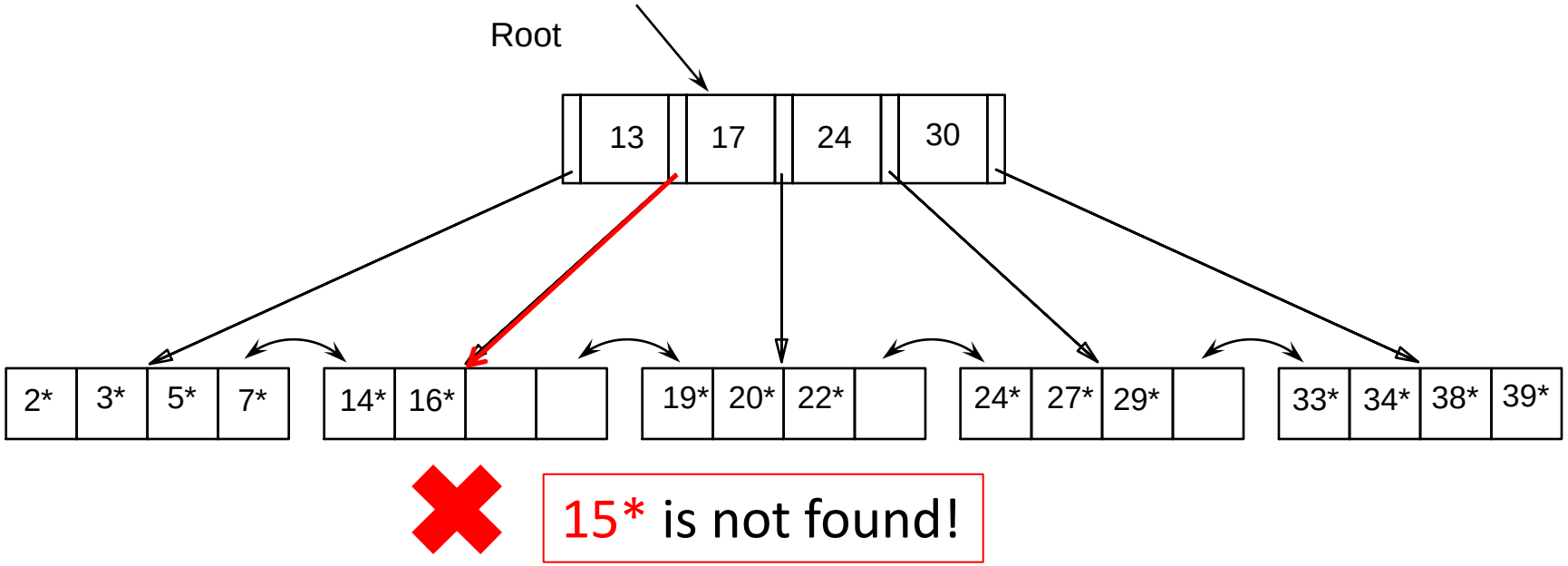
- Example 1: Search for entry 5*



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*

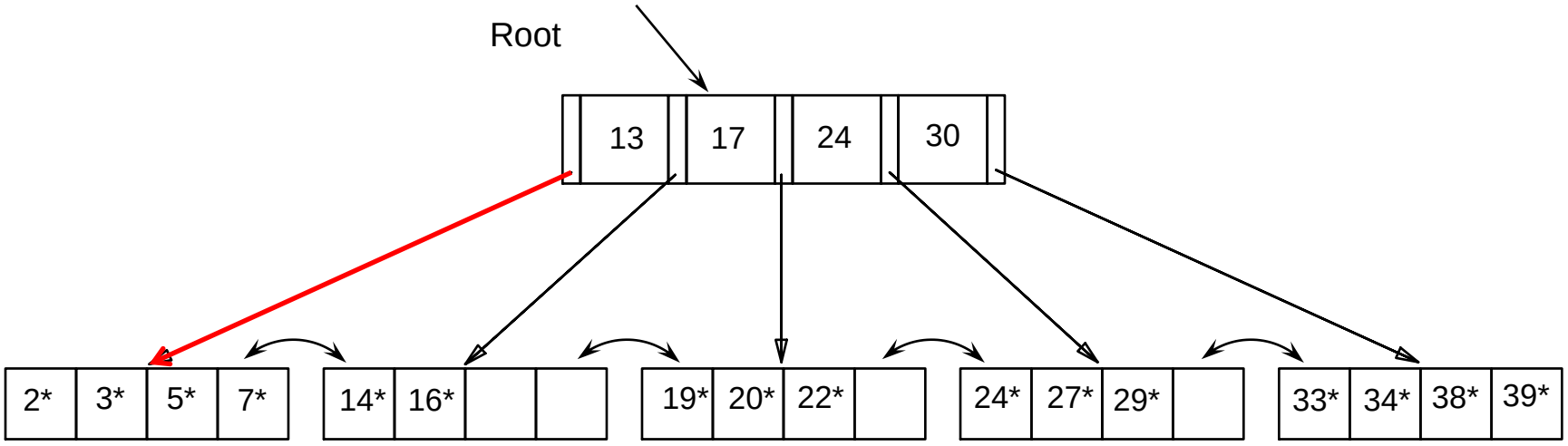


B+ Trees: Inserting Entries

- Find correct leaf ***L***
- Put data entry onto ***L***
  - If ***L*** has enough space, *done*!
  - Else, must *split* ***L*** *into* ***L*** *and a new node* L_2
    - Redistribute 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

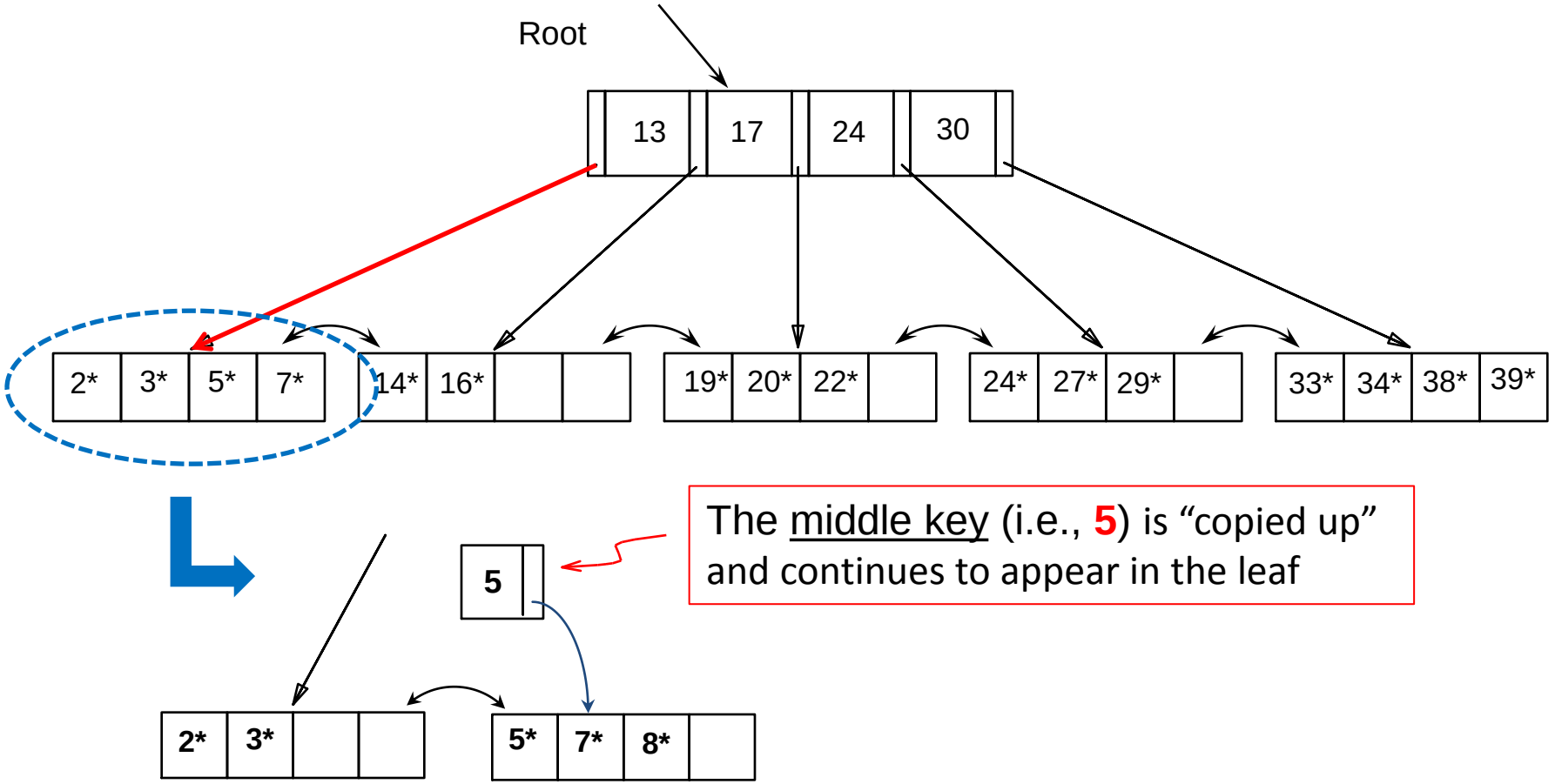
- Insert entry 8*



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

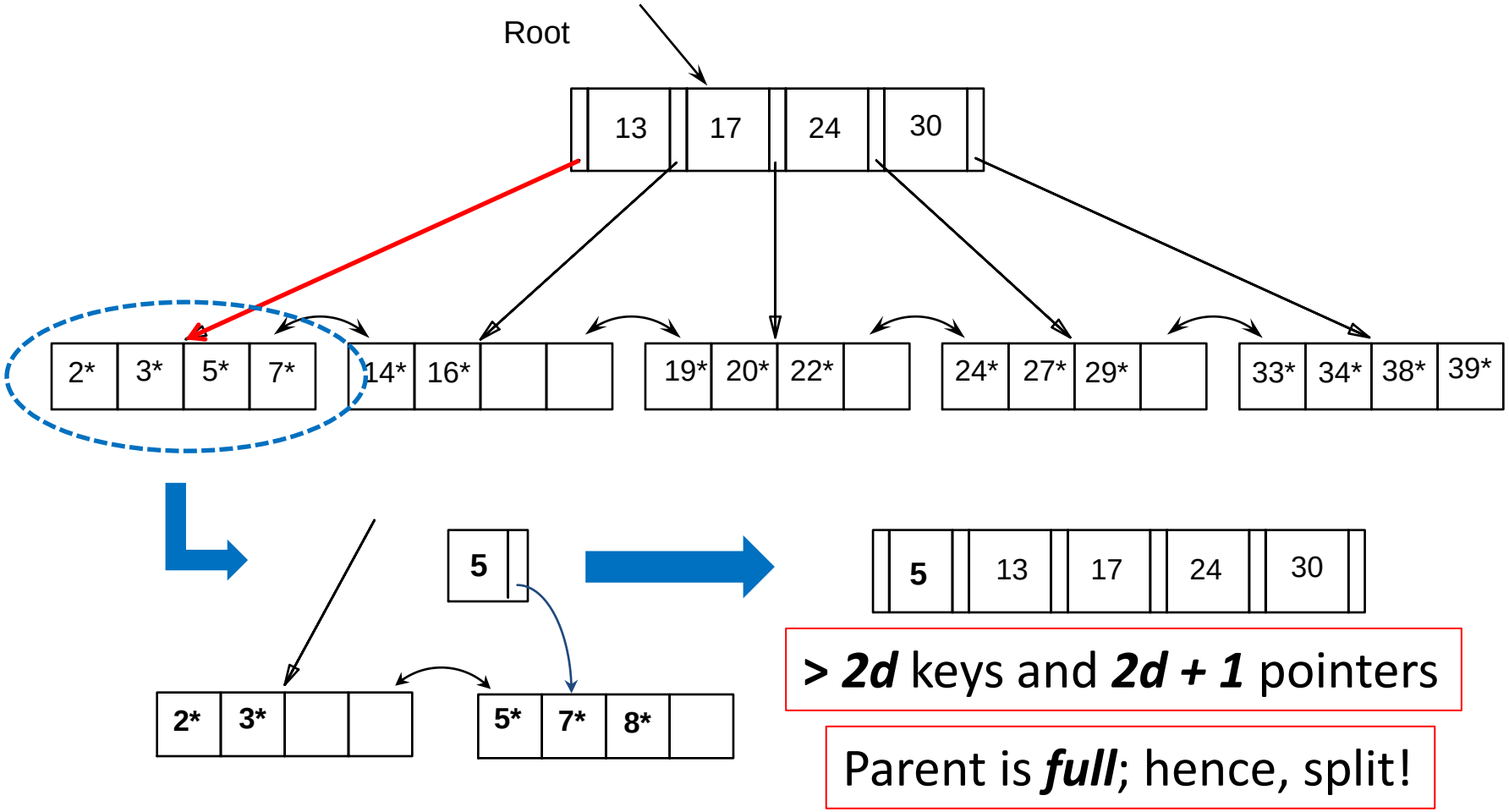
B+ Tree: Examples of Insertions

- Insert entry 8*



B+ Tree: Examples of Insertions

- Insert entry 8*

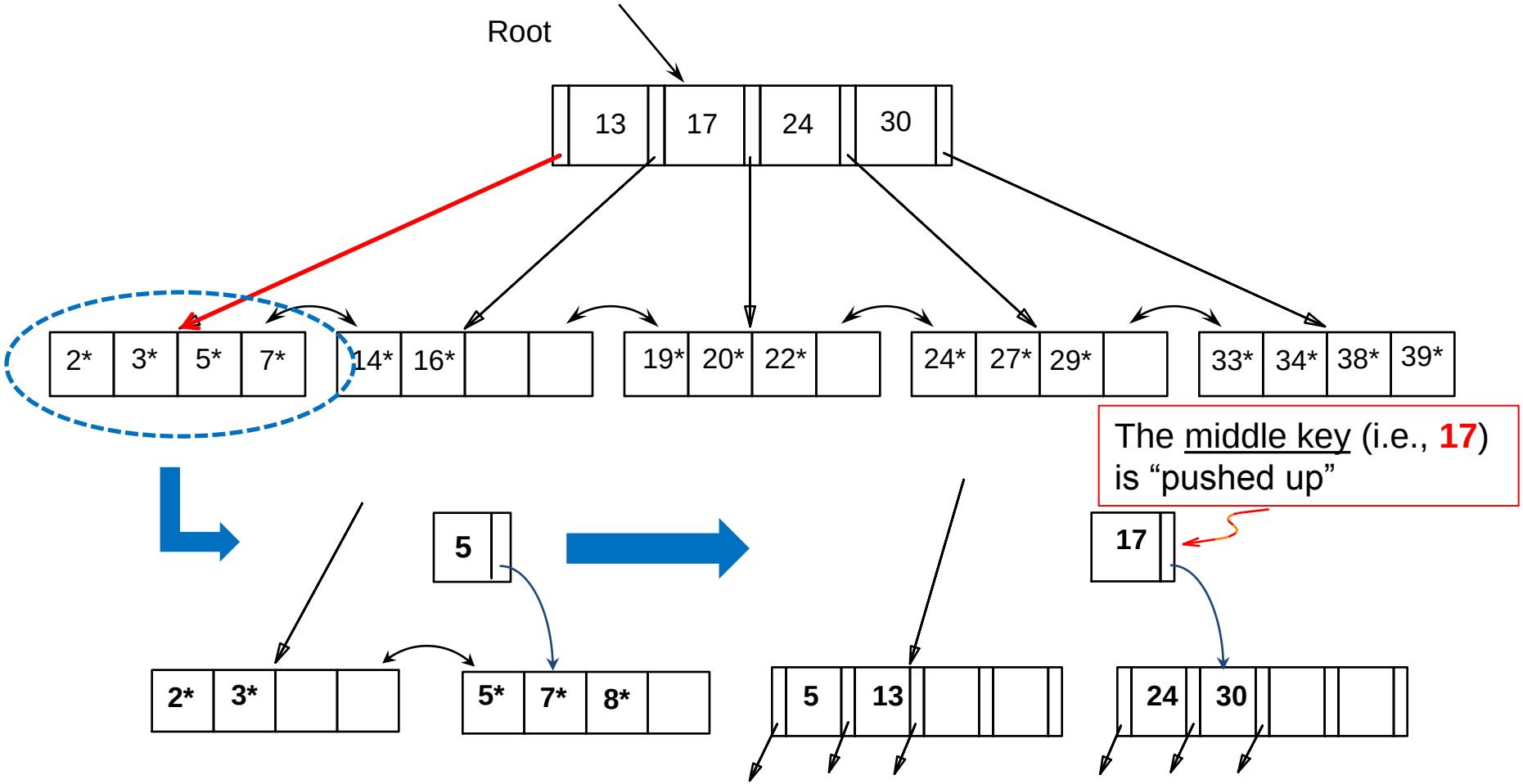


> ***2d*** keys and ***2d + 1*** pointers

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

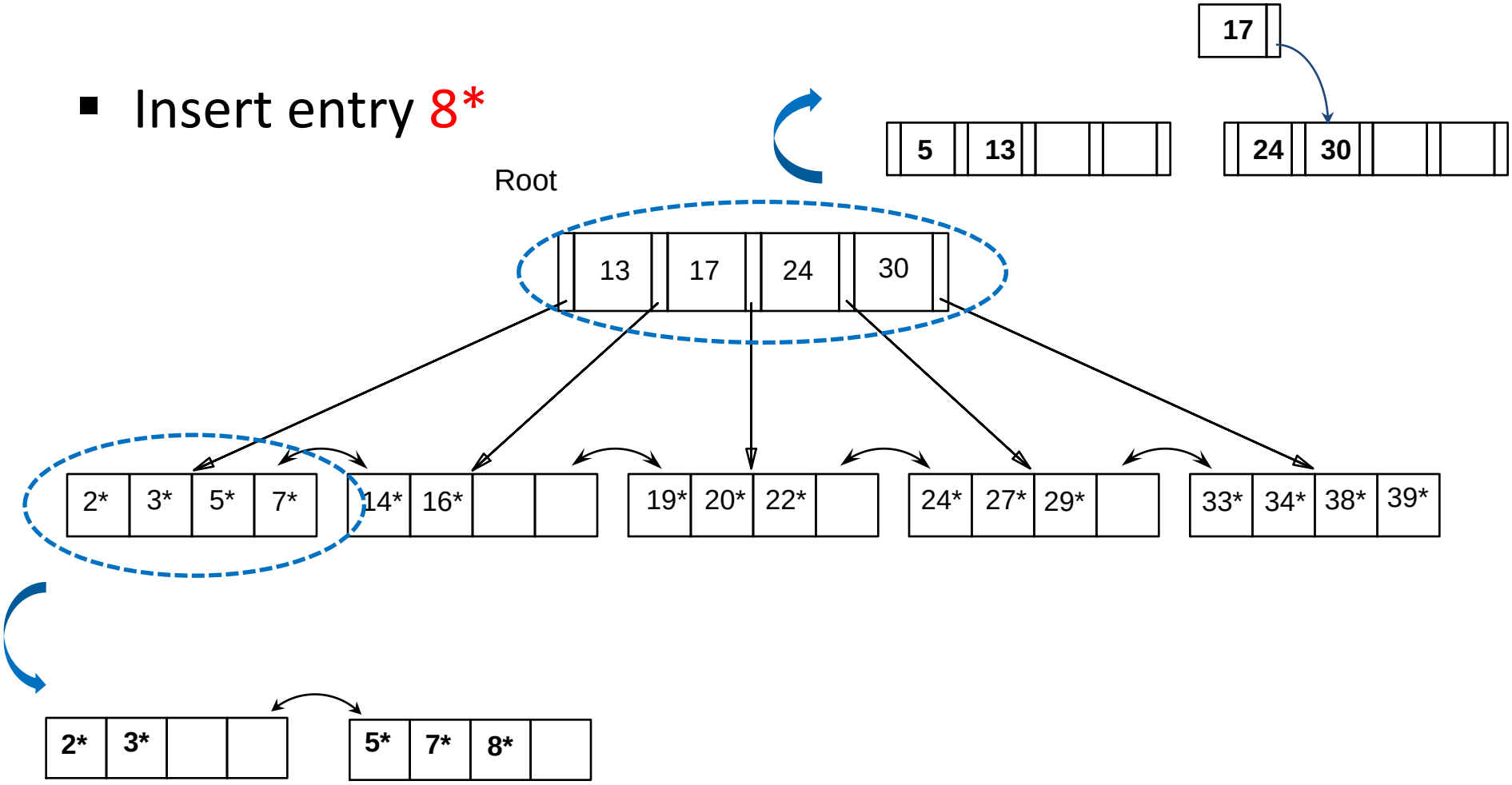
B+ Tree: Examples of Insertions

- Insert entry 8*



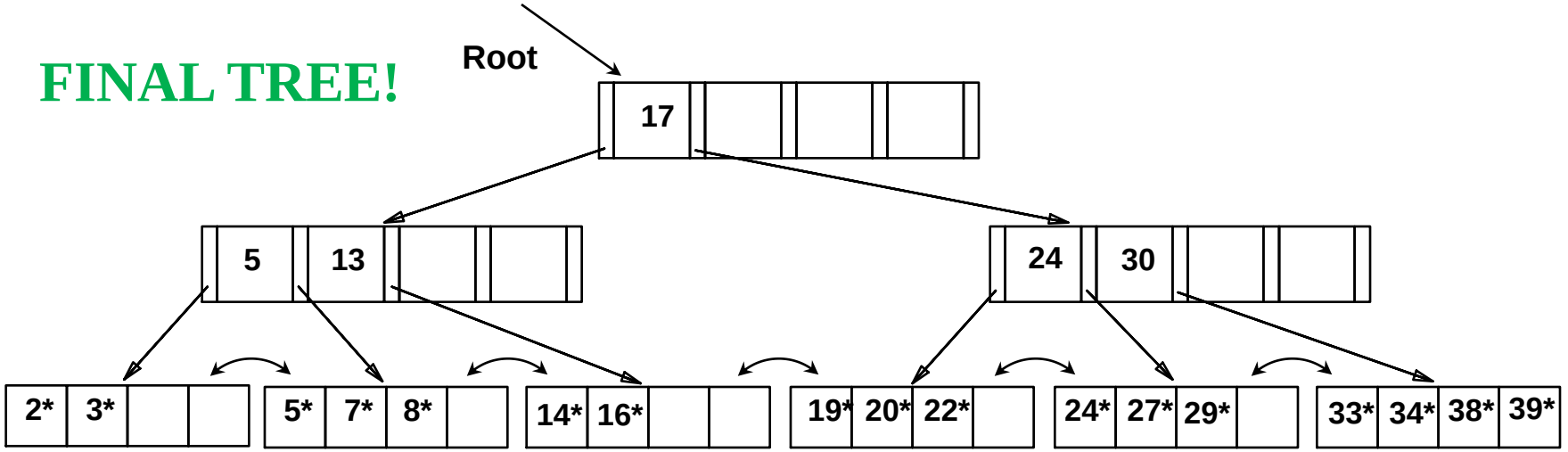
B+ Tree: Examples of Insertions

- Insert entry 8*



B+ Tree: Examples of Insertions

- Insert entry 8*

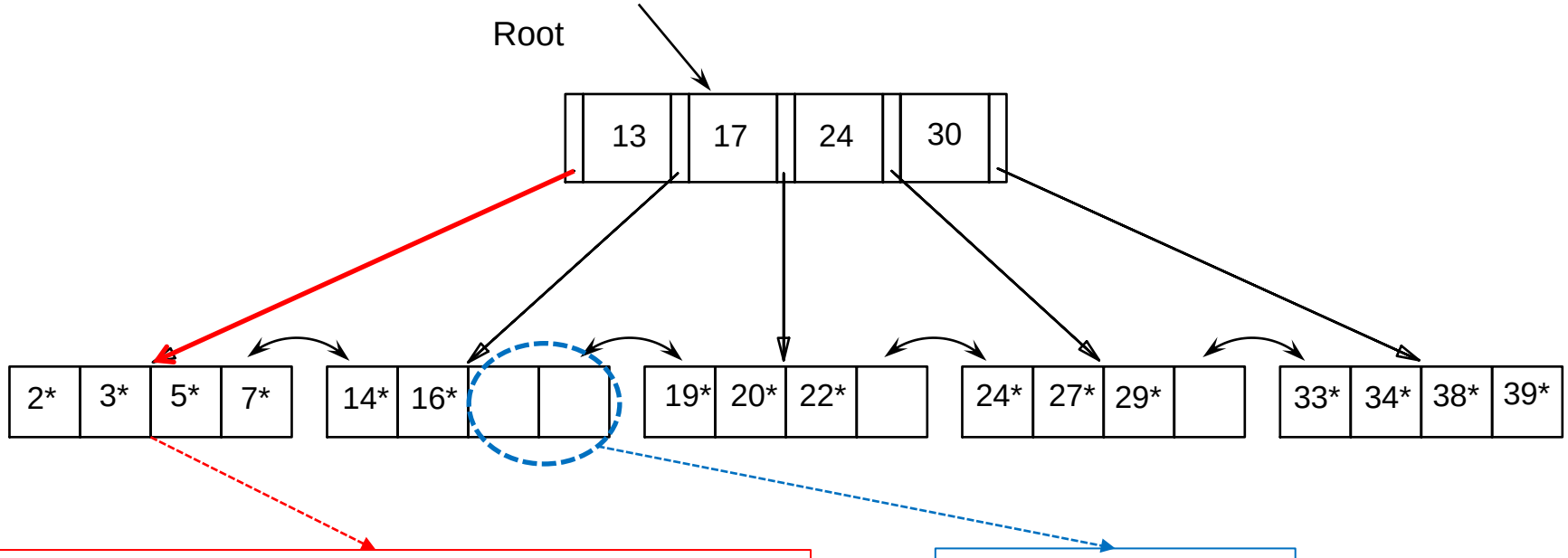


Splitting the root lead to an increase of height by 1!

What about re-distributing entries instead of splitting nodes?

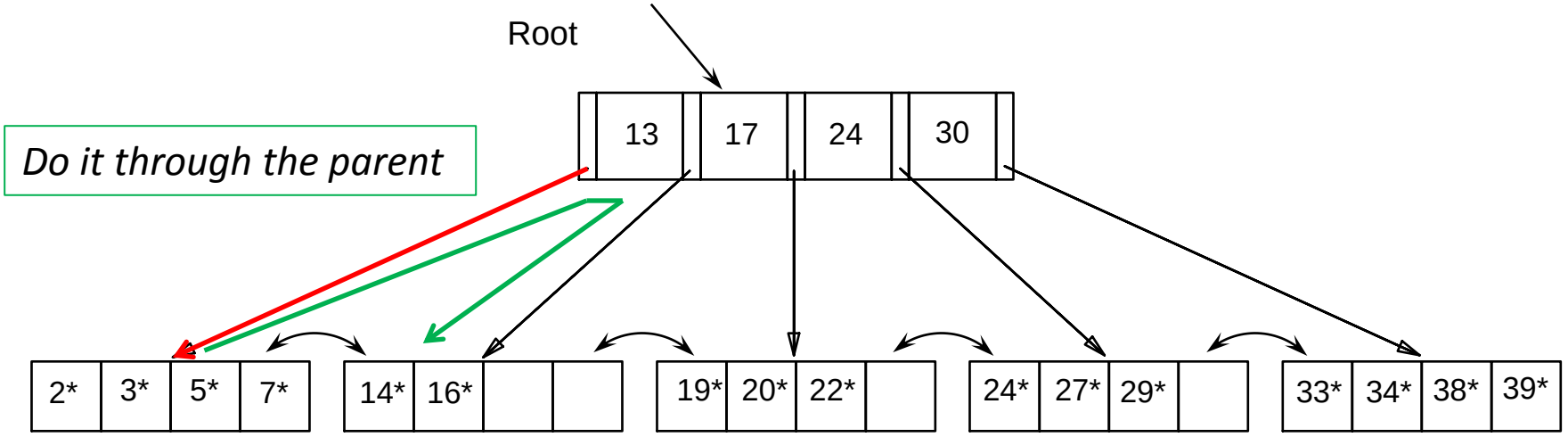
B+ Tree: Examples of Insertions

- Insert entry 8*



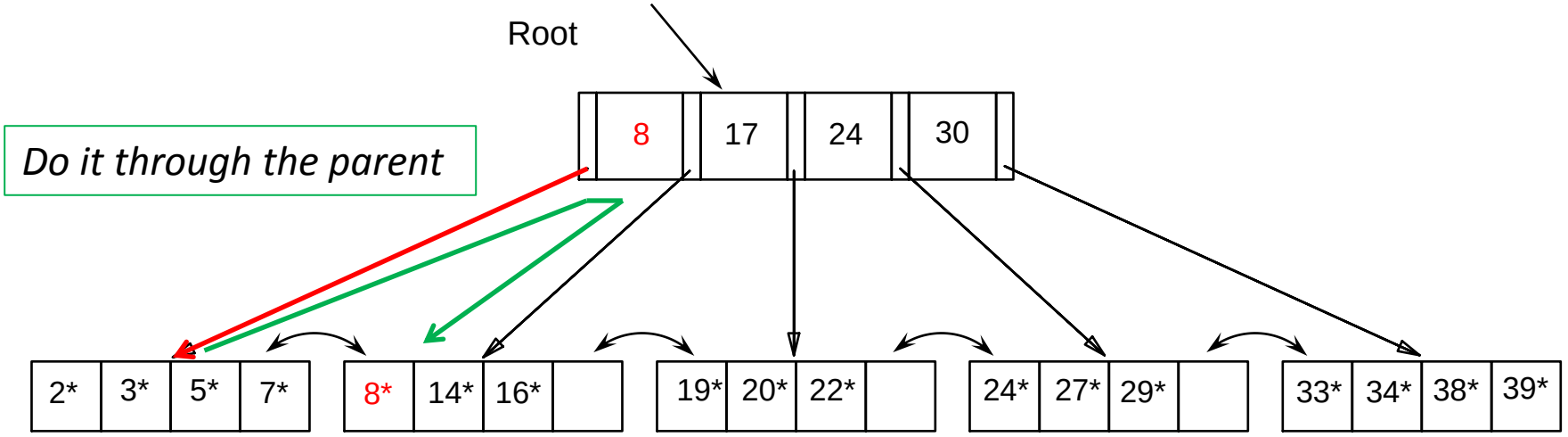
B+ Tree: Examples of Insertions

- Insert entry 8*



B+ Tree: Examples of Insertions

- Insert entry 8*



"Copy up" the new low key value!

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

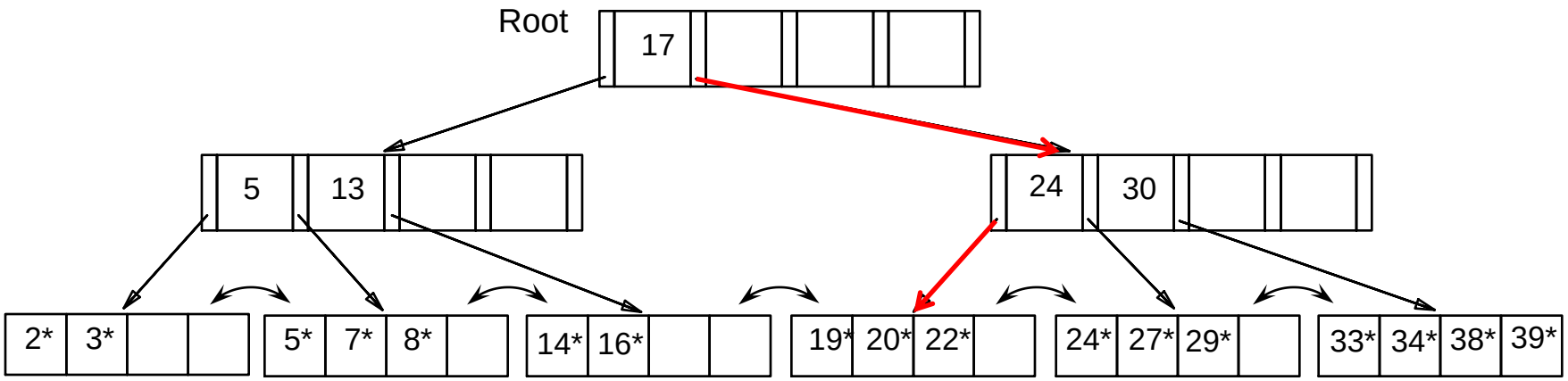
- Every data entry must appear in a leaf node; hence, “copy up” the middle key upon splitting
- When splitting index entries, simply “push up” the middle key
- Apply splitting and/or redistribution on leaf nodes
- Apply only splitting on non-leaf nodes

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

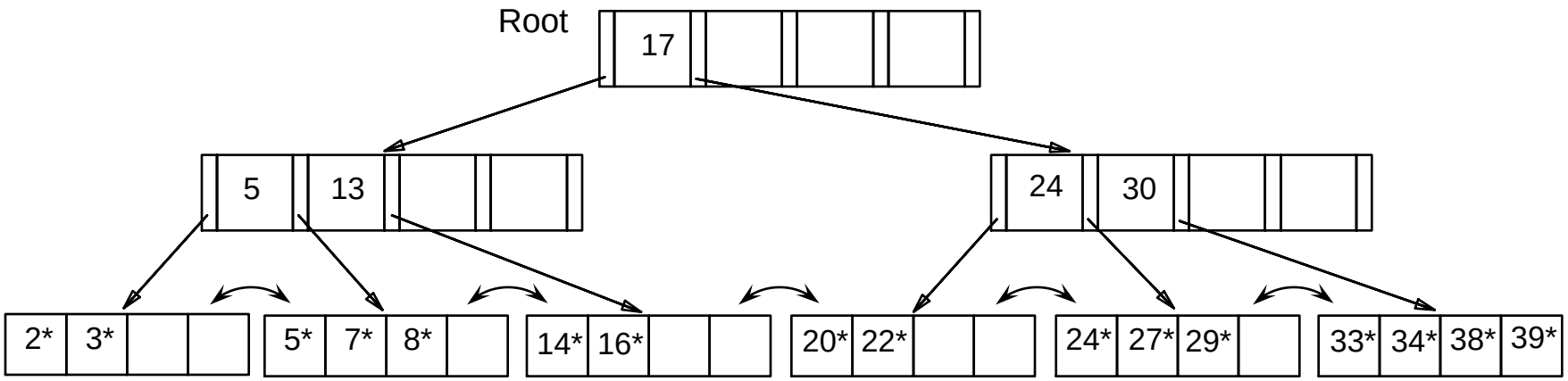
- Delete 19*



Removing 19* does not entail an underflow

B+ Tree: Examples of Deletions

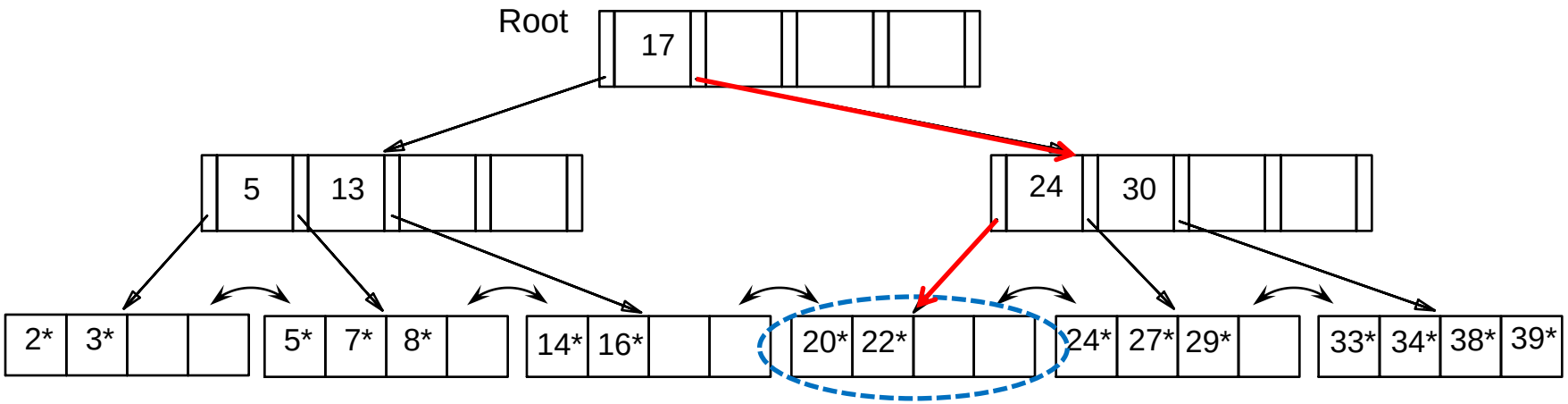
- Delete 19*



FINAL TREE!

B+ Tree: Examples of Deletions

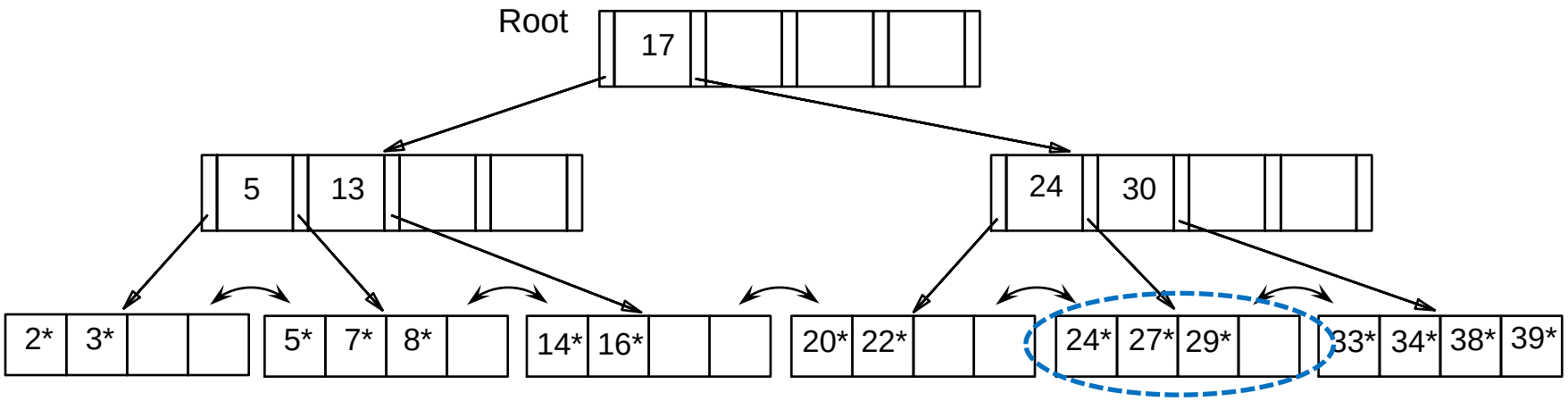
- Delete 20*



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

B+ Tree: Examples of Deletions

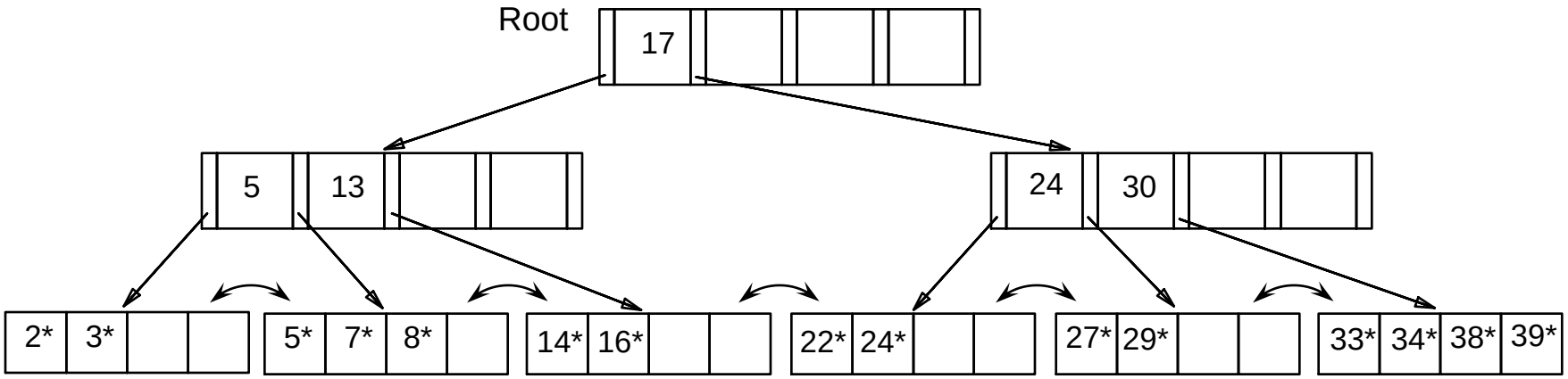
- Delete 20*



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

B+ Tree: Examples of Deletions

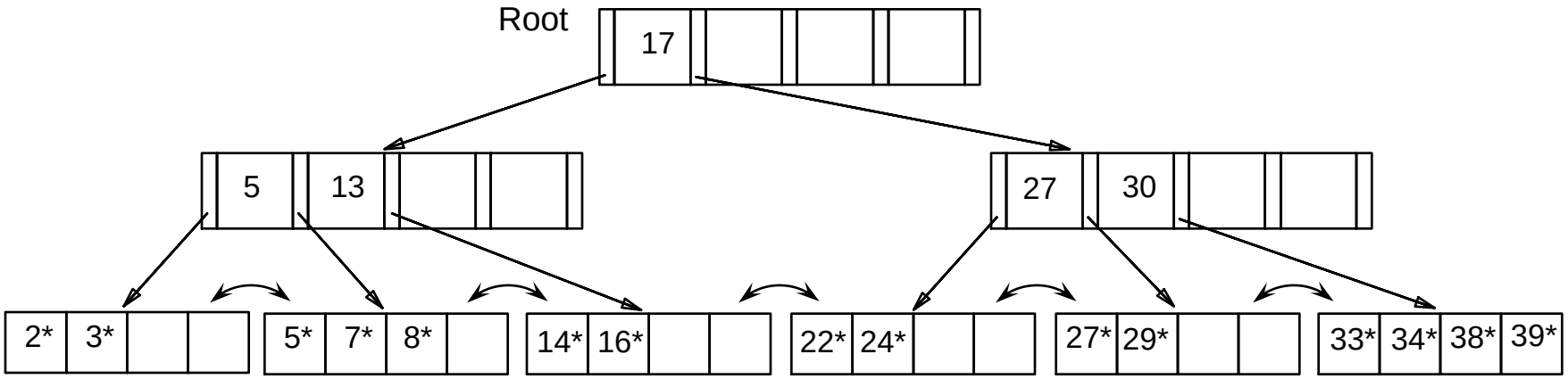
- Delete 20*



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

B+ Tree: Examples of Deletions

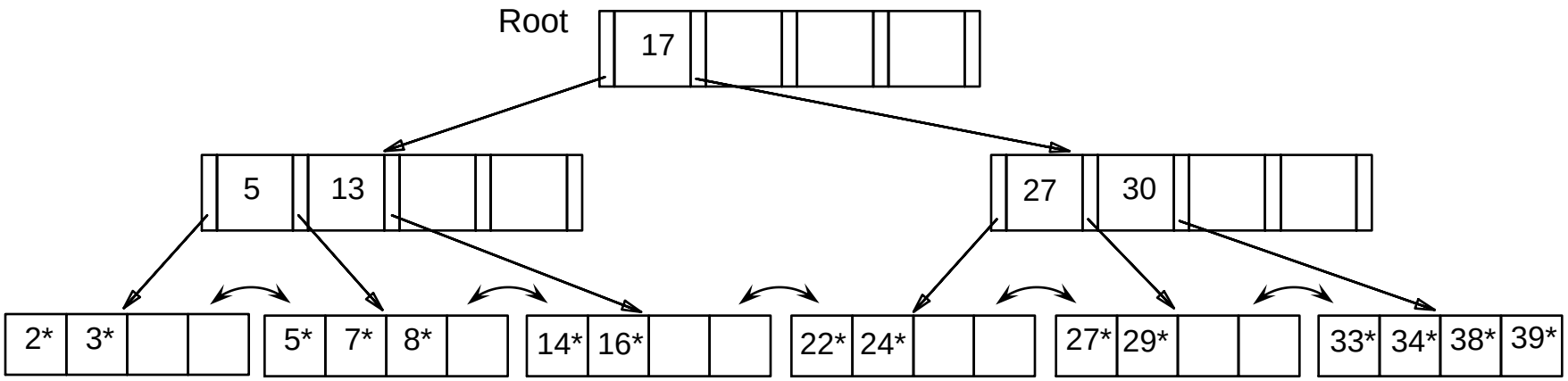
- Delete 20*



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

B+ Tree: Examples of Deletions

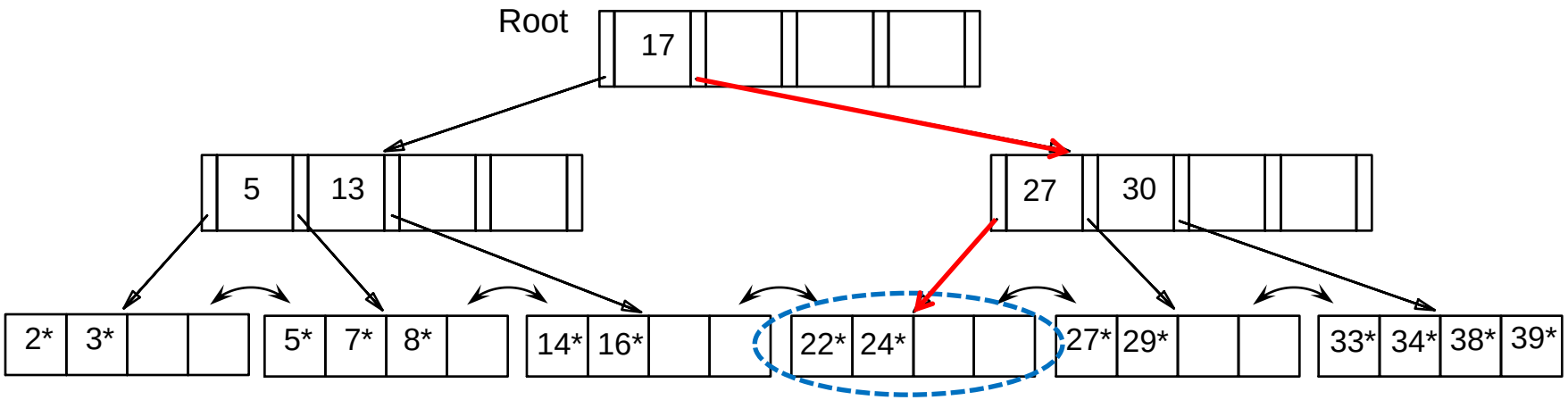
- Delete 20*



FINAL TREE!

B+ Tree: Examples of Deletions

- Delete 24*

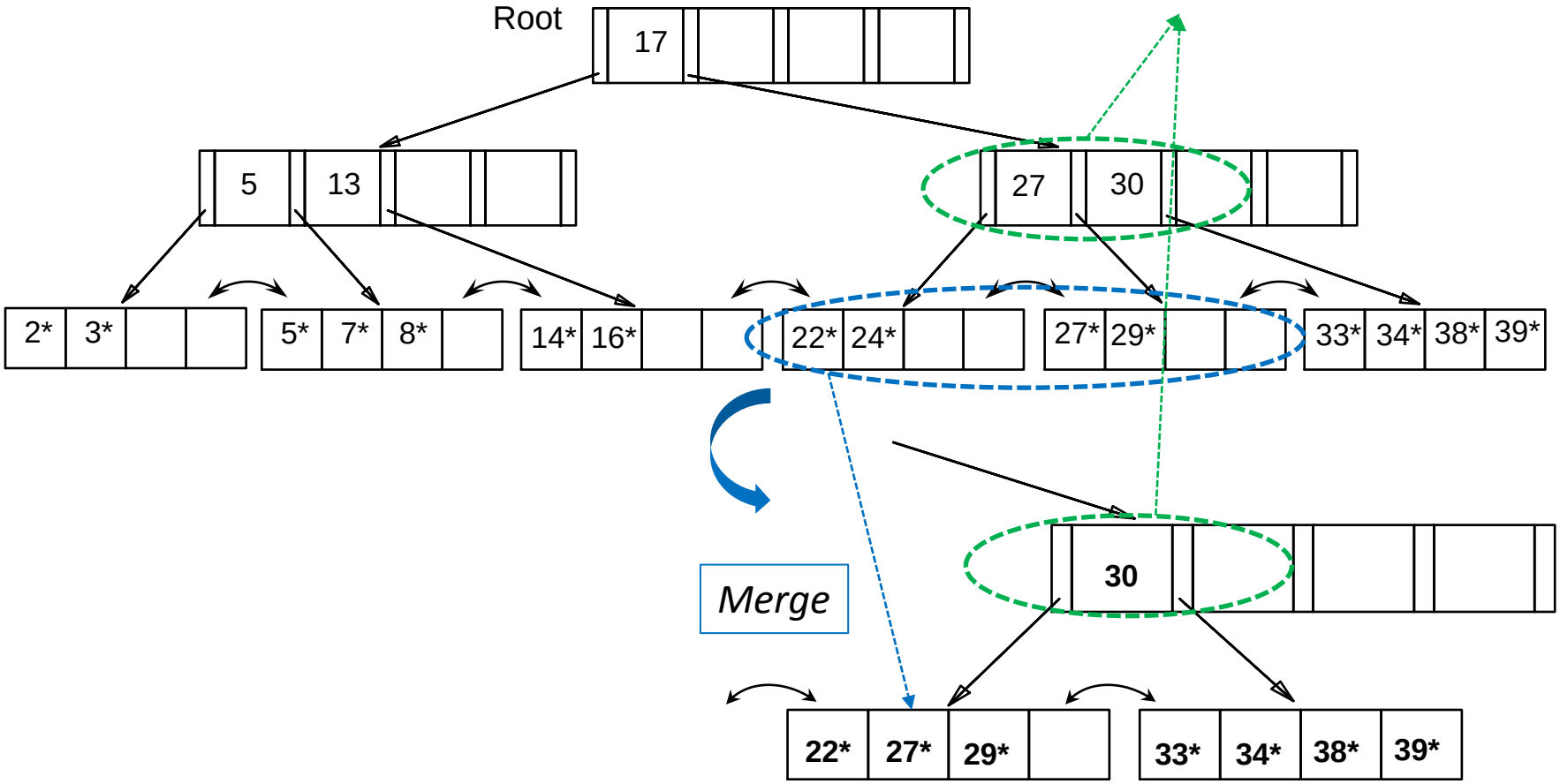


The affected leaf will contain only 1 entry and the sibling cannot lend any entry (i.e., redistribution is not applicable); hence, *merge*!

B+ Tree: Examples of Deletions

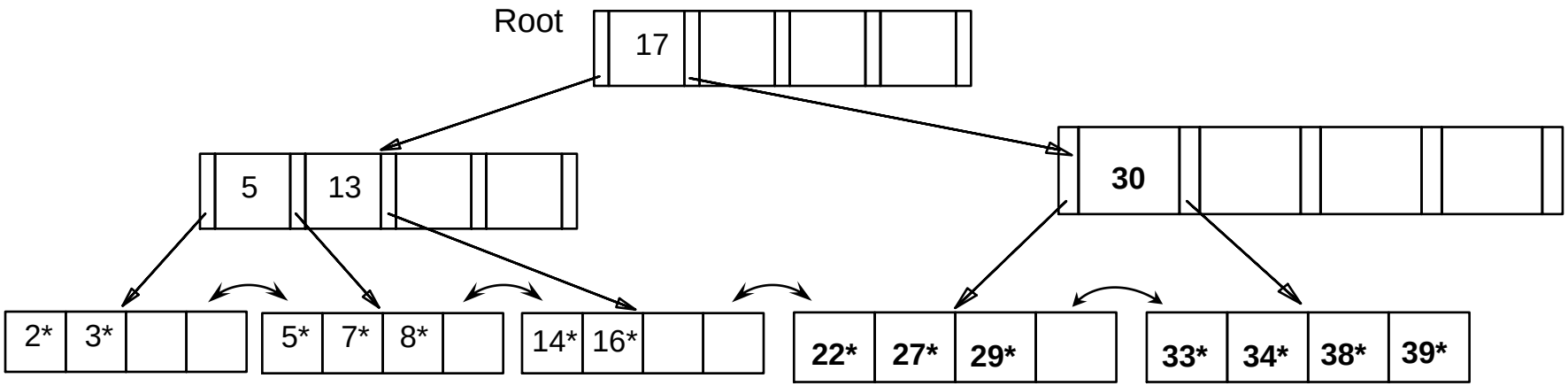
- Delete 24*

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



B+ Tree: Examples of Deletions

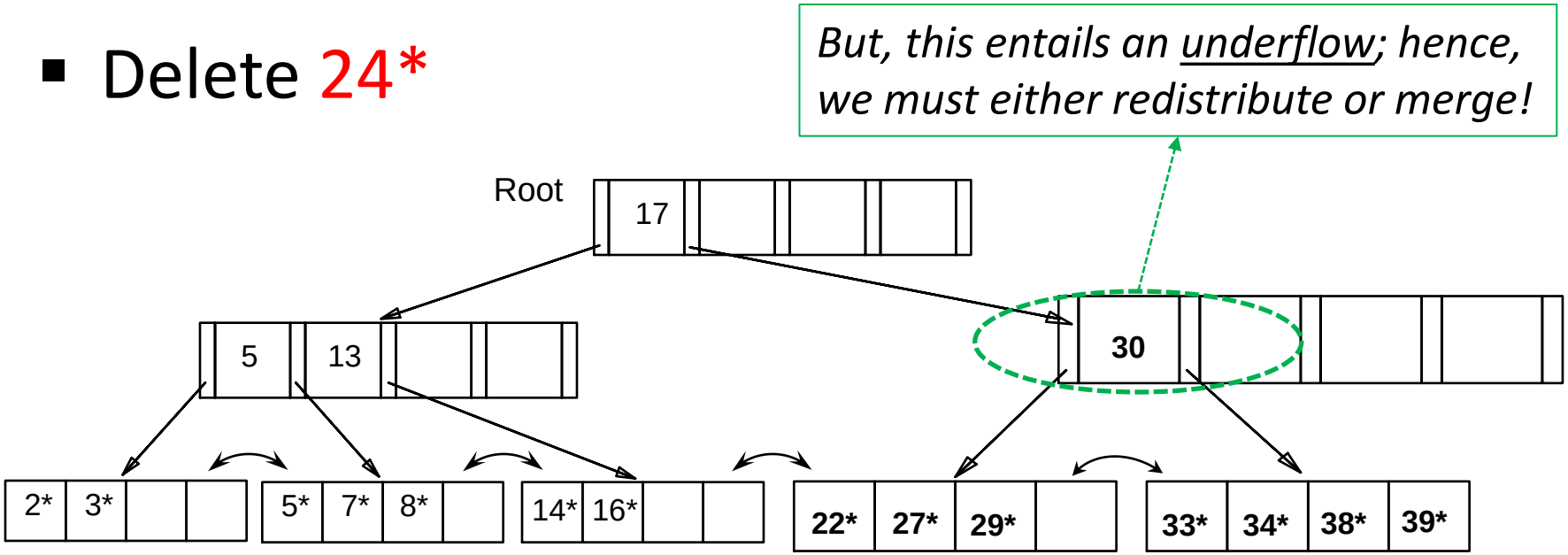
- Delete 24*



Almost there…

B+ Tree: Examples of Deletions

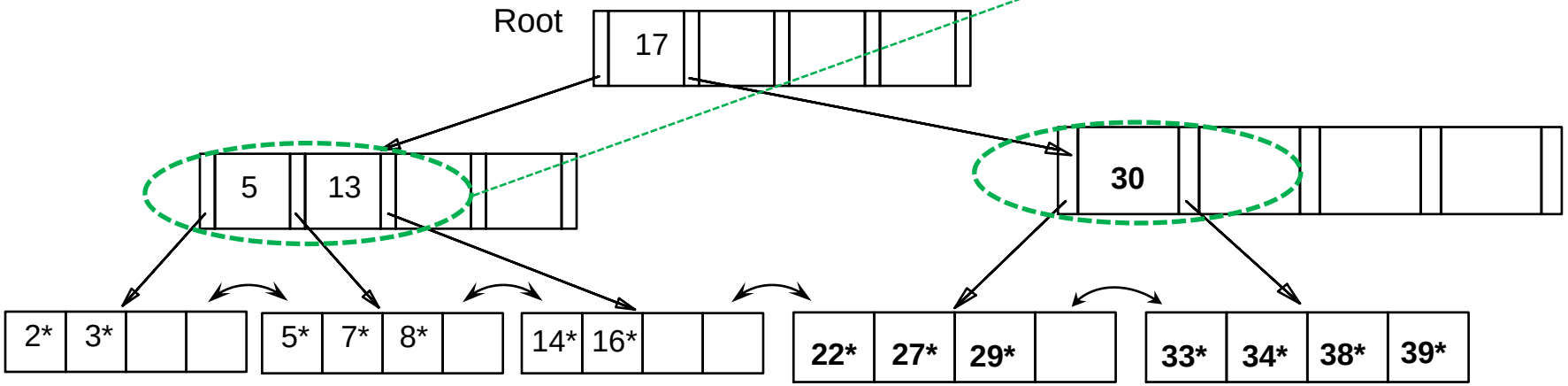
- Delete 24*



B+ Tree: Examples of Deletions

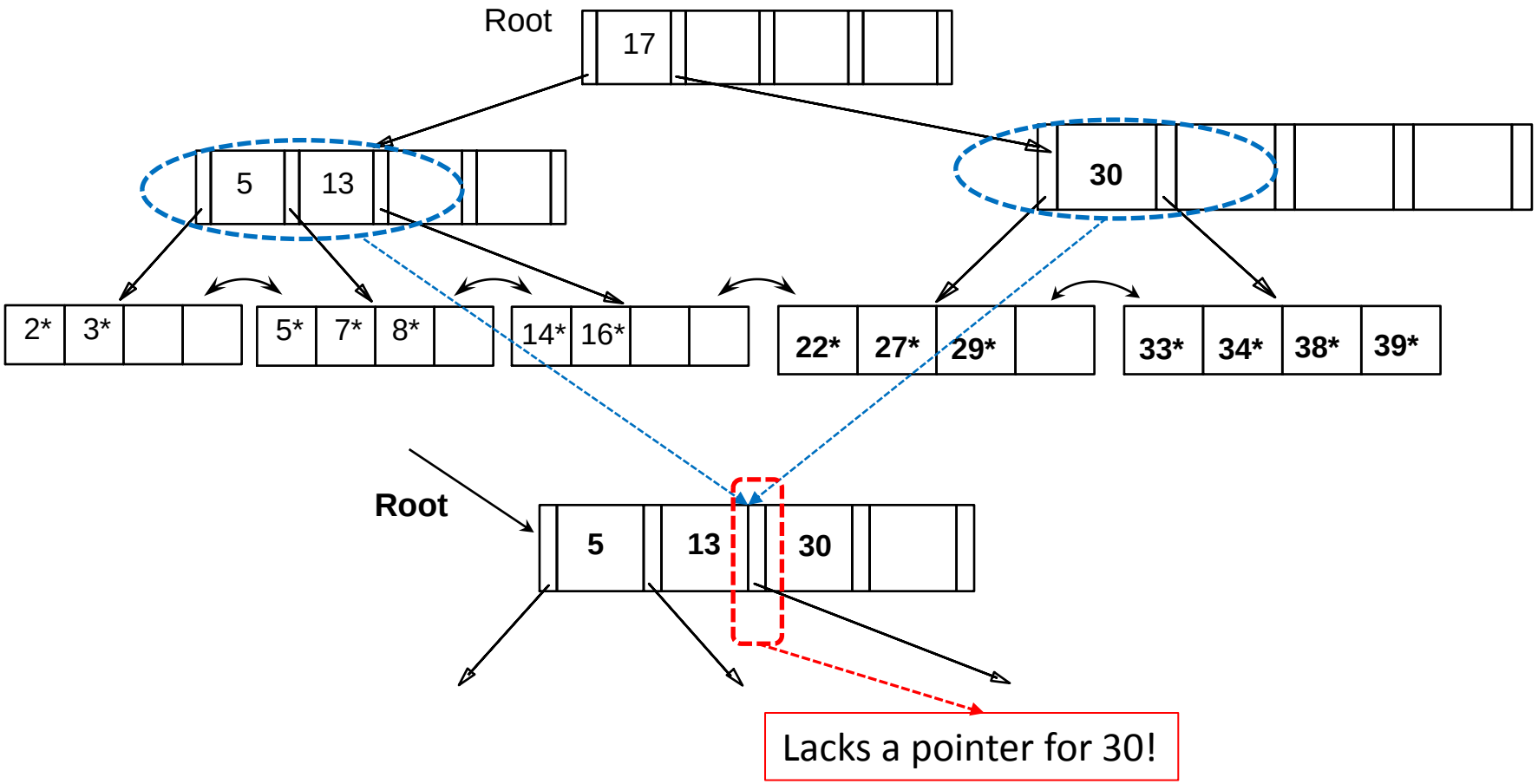
- Delete 24*

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



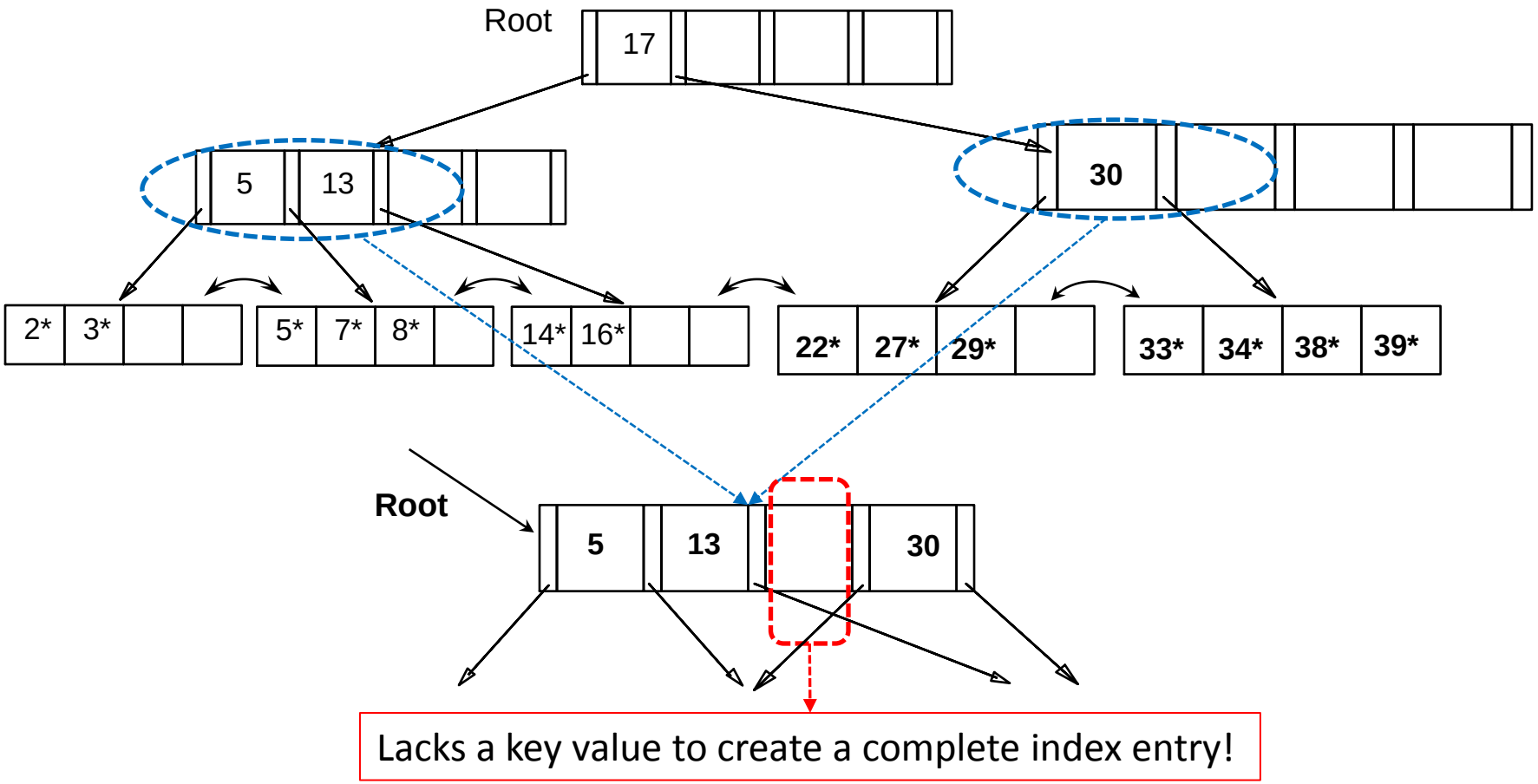
B+ Tree: Examples of Deletions

- Delete 24*



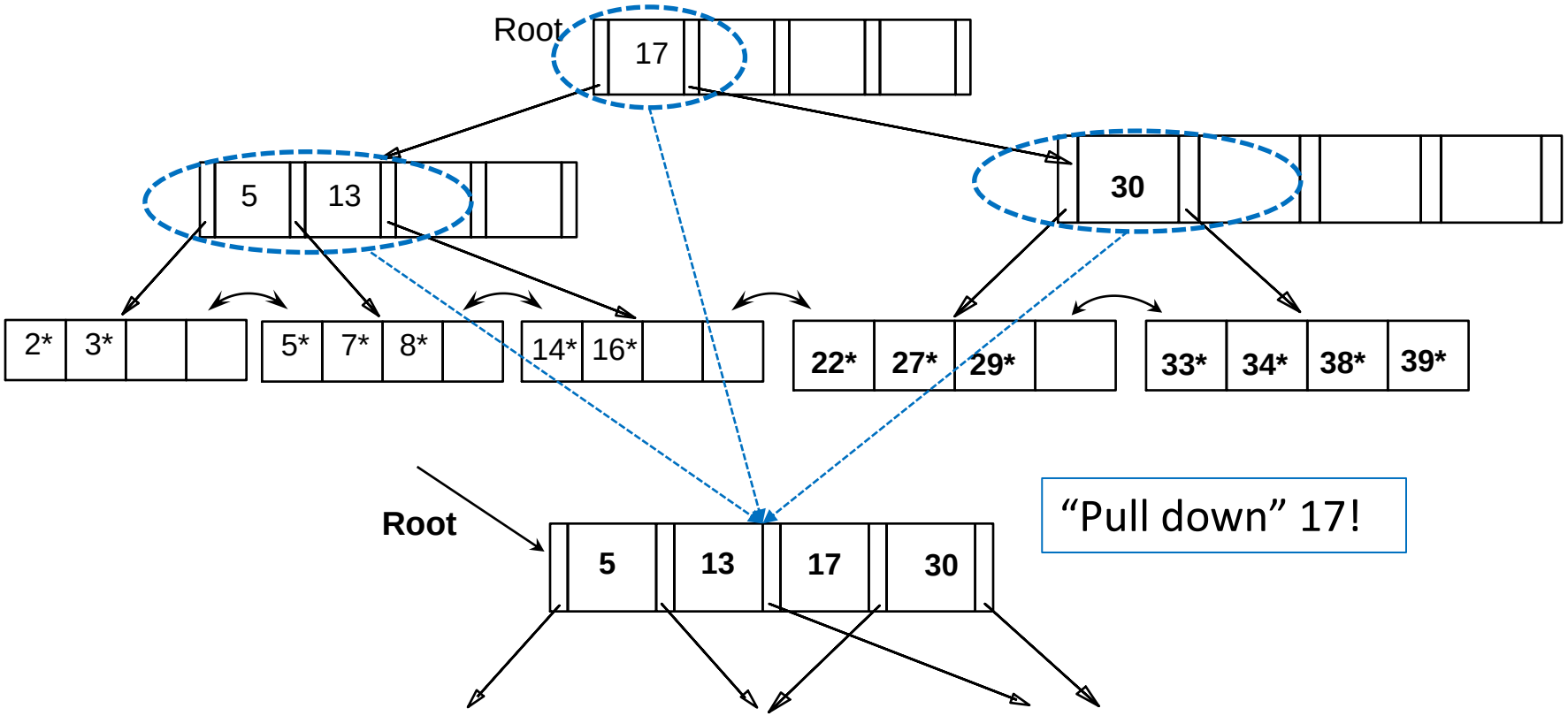
B+ Tree: Examples of Deletions

- Delete 24*



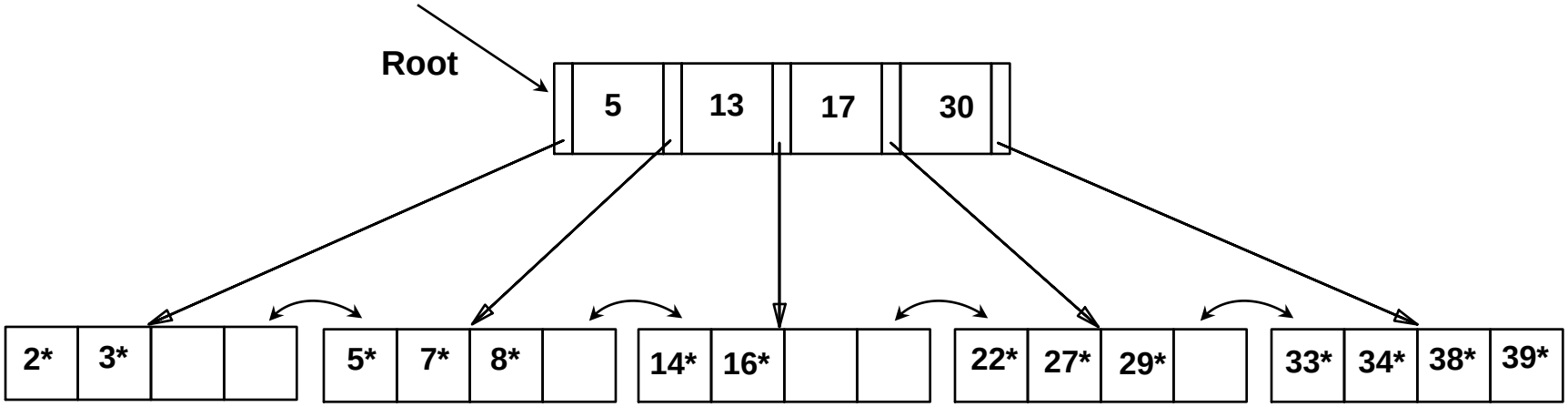
B+ Tree: Examples of Deletions

- Delete 24*



B+ Tree: Examples of Deletions

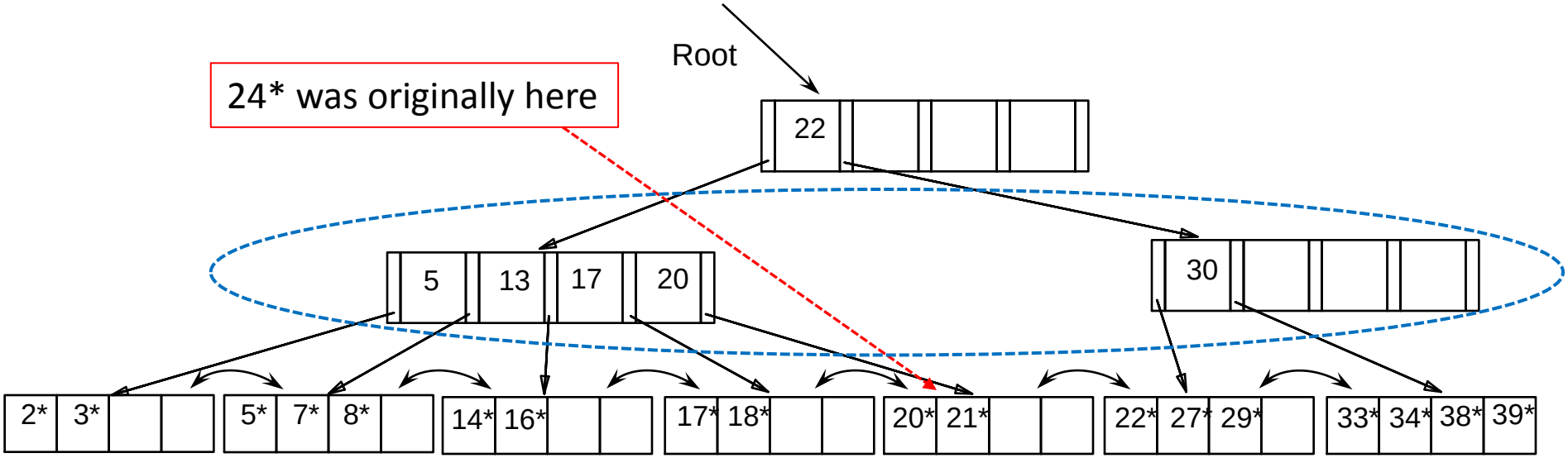
- Delete 24*



FINAL TREE!

B+ Tree: Examples of Deletions

- Delete 24*

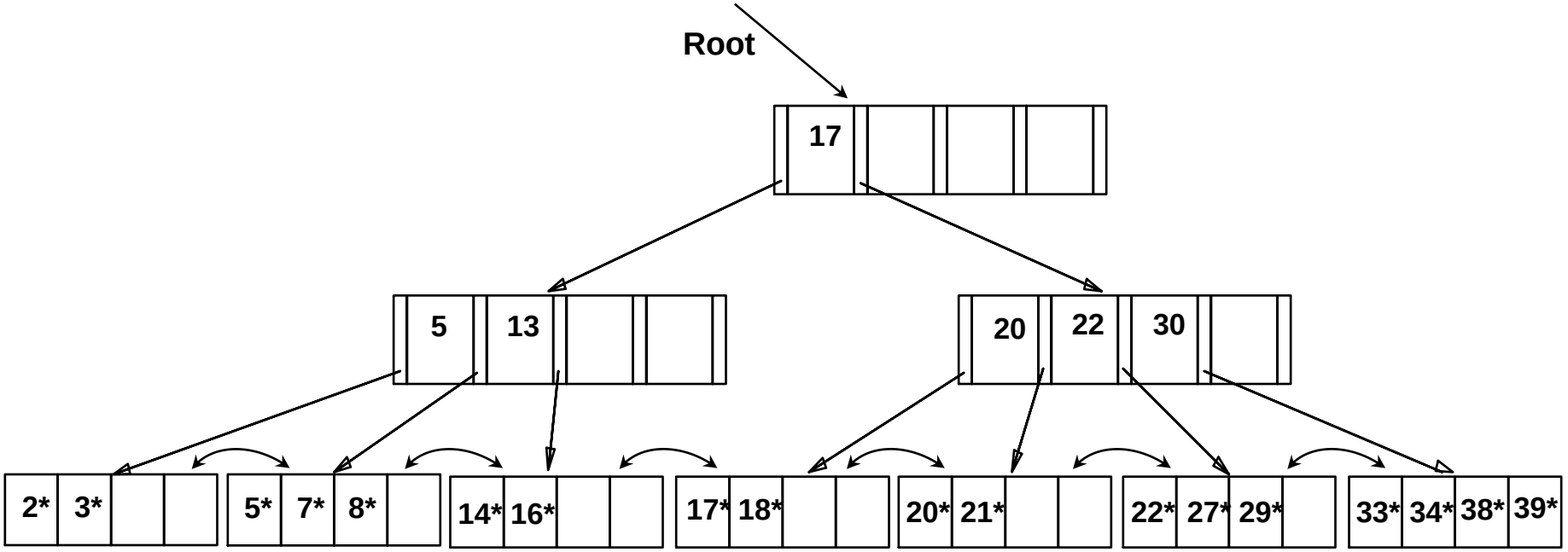


Assume (instead) the above tree *during* deleting 24*

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

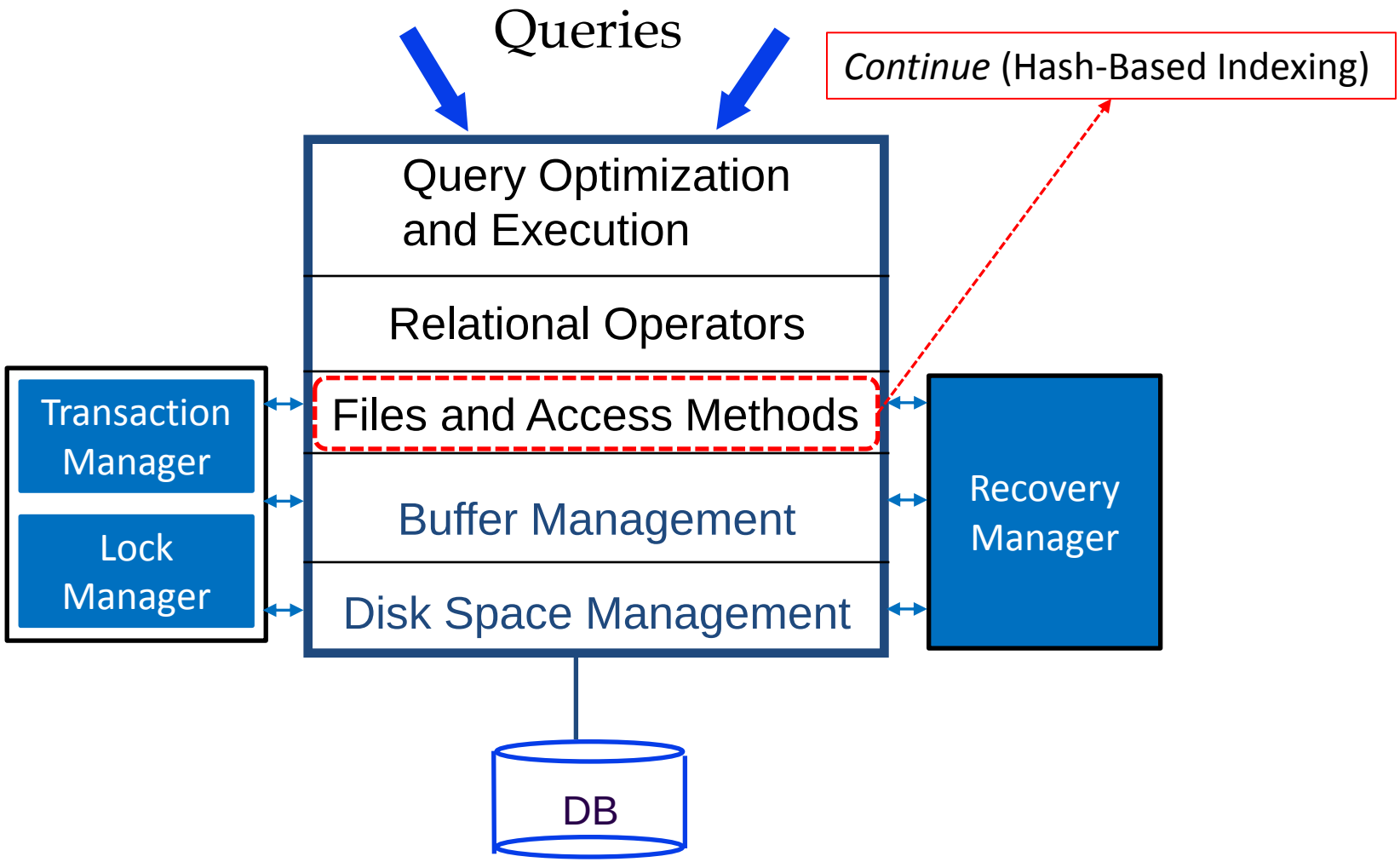
B+ Tree: Examples of Deletions

- 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