Φροντιστήριο 10^{ο}: Φυσική Σχεδίαση Βάσεων Δεδομένων

Φίλιππος Γουίδης

Τμήμα επιστήμης υπολογιστών
Πανεπιστήμιο Κρήτης

15 Δεκεμβρίου 2023

B-trees

Ένα B-tree(Balanced Tree) τάξης μ είναι ένα δέντρο με τις παρακάτω ιδιότητες:

- Η ουρά έχει τουλάχιστον δύο παιδιά, εκτός εάν είναι φύλλο
- Κανένας κόμβος δεν έχει πάνω από μ παιδιά
- Κάθε κόμβος εκτός από την ουρά και τα φύλλα έχει τουλάχιστον $\lceil \frac{d}{2} \rceil$ παιδιά
- Όλα τα φύλλα εμφανίζονται στο ίδιο επίπεδο
- Κάθε κόμβος περιέχει κλειδιά και δείκτες.
- Ενα υπόδενδρο που ξεκινάει από ένα δείκτη που βρίσκεται αριστερά ενός κλειδιού, περιέχει αποκλειστικά κλειδιά με μικρότερες τιμές
- Αν ένας κόμβος περιέχει k κλειδιά τότε περιέχει k+1 δείκτες προς άλλους κόμβους

B-trees

- Εναλλακτικά, ένα B-tree μπορεί να οριστεί από το παράγοντα d. Συγκεκριμένα, κάθε εσωτερικός κόμβος μπορεί να έχει το πολύ 2d-1 παιδιά και να περιέχει 2d-2 εγγραφές.
- Αντίστοιχα, ο ελάχιστος αριθμός παιδιών είναι δ και ο ελάχιστος αριθμός εγγραφών d-1. Αντίστοιχα, κάθε φύλλο μπορεί να περιέχει από d μέχρι 2d-1 εγγραφές (κλειδιά).
- Σε κάποιες περιπτώσεις ο ελάχιστος αριθμός εγγραφών στα φύλλα ενδέχεται να είναι διαφορετικός από τον ελάχιστο αριθμό εγγραφών στους εσωτερικούς κόμβους. Στην προκειμένη περίπτωση το B-tree ορίζεται από τα d και e.

B-trees

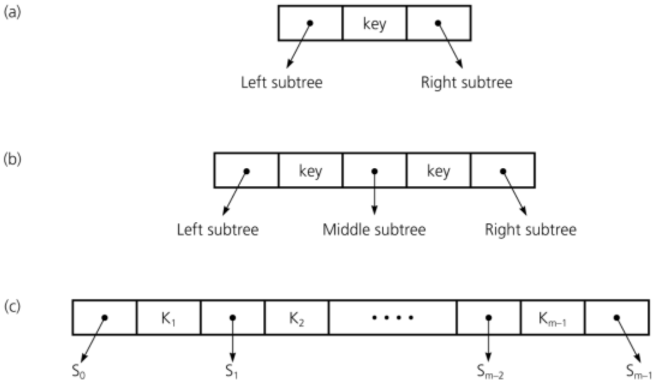
Οι επόμενοι δύο πίνακες συνοψίζουν τις διαφορετικές περιπτώσεις. Ο πρώτος αναφέρεται στην περίπτωση που είναι η γνωστή η τάξη m του δέντρου. Ενώ ο δεύτερος αφορά την περίπτωση που ξέρουμε τα d και e.

	Μέγ. αρ. παιδιών	Ελ. αρ. παιδιών	Μέγ. αρ. εγγραφών	Ελ. αρ. εγγραφών
εσ. κόμβος	m	$\lceil \frac{m}{2} \rceil$	m-1	$\lceil \frac{m}{2} \rceil - 1$
φύλλο	-	-	m-1	$\lceil \frac{m}{2} \rceil - 1$
ρίζα	m	2*	m-1	1*

	Μέγ. αρ. παιδιών	Ελ. αρ. παιδιών	Μέγ. αρ. εγγραφών	Ελ. αρ. εγγραφών
εσ. κόμβος	2d-1	d	2d-2	d-1
φύλλο	-	-	2e-1	e
ρίζα	2d-1 (2e)[1]	2*	2d-2 (2e-1)[1]	1*

[1] Όταν είναι φύλλο

Παράδειγμα B-trees



Εικόνα.: a)BTree με 2 παιδιά, b) BTree με 3 παιδιά, c) BTree με m παιδιά

B+trees

Τα B+trees αποτελούν υποκατηγορία των B-trees που χαρακτηρίζονται από την εξής ιδιότητα:

- Μόνα τα φύλλα περιέχουν δεδομένα [1] , ενώ οι υπόλοιποι κόμβοι περιέχουν δείκτες B-(pointers) προς άλλους κόμβους.
- Συνεπώς, οι εσωτερικοί κόμβοι καθοδηγούν την αναζήτηση (για εύρεση, εισαγωγή ή διαγραφή δεδομένων) και τα φύλλα περιέχουν τις αντίστοιχες εγγραφές.
- Προσοχή! Η παρουσία κάποιου κλειδιού σε εσωτερικό κόμβο δεν συνεπάγεται την ύπαρξη εγγραφής που να αντιστοιχεί σε αυτό το κλειδί.

[1] Για την ακρίβεια, δείκτες προς δεδομένα

Διεργασίες σε B+Trees

- Αναζήτηση (Lookup):
  - Αναζήτηση μιας εγγραφής με κλειδί n. Εύρεση ενός μονοπατιού από την ρίζα προς το φύλλο που περιέχει την εν λόγω εγγραφή, εάν αυτή υπάρχει.
- Εισαγωγή (Insertion):
  - Εισαγωγή μιας εγγραφής με κλειδί n
- Διαγραφή (Deletion):
  - Διαγραφή της εγγραφής με κλειδί n

Αλγόριθμος Αναζήτησης σε B+tree

func *find* (search key value K) **returns** nodepointer
// Given a search key value, finds its leaf node
return tree_search(root, K); // searches from root
endfunc

func *tree_search* (nodepointer, search key value K) **returns** nodepointer
// Searches tree for entry
if *nodepointer is a leaf, return nodepointer;
else,
    if $K < K_1$ then return tree_search(P_0, K);
    else,
        if $K \geq K_m$ then return tree_search(P_m, K); // m = # entries
        else,
            find i such that $K_i \leq K < K_{i+1}$;
            return tree_search(P_i, K)
endfunc

Αλγόριθμος Εισαγωγής σε B+tree

```
proc inseTt (nodepointel', entry, newchildentry)
// InseTts entry into subtree with TOot '*nodepointer'; degree is d;
/'/'newchildentry' null initially, and null on return unless child is split

if *nodepointer is a non-leaf node, say N,
    find'i such that K_i ≤ entry's key value < J(i+1;     // choose subtree
    insert(P_i, entry, newchildentry);           // recurs'ively, insert entry
    if newchildentry is null, return;        // usual case; didn't split child
    else,            // we split child, must insert *newchildentry in N
        if N has space,                                        // usual case
            put *newchildentry on it, set newchildentry to null, return;
        else,                  // note difference wrt splitting of leaf page!
            split N:        // 2d + 1 key values and 2d + 2 nodepointers
            first d key values and d + 1 nodepointers stay,
            last d keys and d + 1 pointers move to new node, N2;
            // *newchildentry set to guide searches between Nand N2
            newchildentry = & ((smallest key value on N2,
                                 pointer to N2));
            if N is the root,                  // root node was just split
                create new node with (pointer to N, *newchildentry);
                make the tree's root-node pointer point to the new node;
            return;

if *nodepointer is a leaf node, say L,
    if L has space,                                            // usual case
    put entry on it, set newchildentry to null, and return;
    else.                              // once in a while, the leaf is full
        split L: first d entries stay, rest move to brand new node L2;
        newchildentry = & ((smallest key value on L2, pointer to L2));
        set sibling pointers in Land L2;
        return;
endproc
```

Αλγόριθμος Διαγραφής σε B+tree

```
proc delete (parentpointer, nodepointer, entry, oldchildentry)
// Deletes entry from s'ubtree with TOot '*nodepointer'; degree is d;
// 'oldchildentry' null initially, and null upon return unless child deleted
if *nodepointer is a non-leaf node, say N,
    find i such that Ki ≤ entry's key value < Ki+l;   // choose subtree
    delete(nodepointer, Pi, entry, oldchildentry);    // recursive delete
    if oldchildentry is null, return;            // usual case: child not deleted
    else,                         // we discarded child node (see discussion)
        remove *oldchildentry from N,        // next, check for underflow
        if N has entries to spare,                          // usual case
            set oldchildentry to null, return; // delete doesn't go further
        else,                  // note difference wrt merging of leaf pages!
            get a sibling S of N:     // parentpointer arg used to find S
            if S has extra entries,
                redistribute evenly between Nand S through parent;
                set oldchildentry to null, return;
            else, merge Nand S                    // call node on rhs M
                oldchildentry = & (current entry in parent for M);
                pull splitting key from parent down into node on left;
                move all entries from M to node on left;
                discard empty node M, return;

if *nodepointer is a leaf node, say L,
    if L has entries to spare,                              // usual case
        remove entry, set oldchildentry to null, and return;
    else,                  // once in a while, the leaf becomes underfull
        get a sibling S of L;            // parentpointer used to find S
        if S has extra entries,
            redistribute evenly between Land S;
            find entry in parent for node on right;          // call it M
            replace key value in parent entry by new low-key value in M;
            set oldchildentry to null, return;
        else, merge Land S                            // call node on rhs M
            oldchildentry = & (current entry in parent for M);
            move all entries from M to node on left;
            discard empty node M, adjust sibling pointers, return;
endproc
```

Εισαγωγή σε B+tree : Περιπτώσεις

The `insert` algorithm for B+ Trees

Leaf Page Full	Index Page FULL	Action
NO	NO	Place the record in sorted position in the appropriate leaf page
YES	NO	1. Split the leaf page 2. Place Middle Key in the index page in sorted order. 3. Left leaf page contains records with keys below the middle key. 4. Right leaf page contains records with keys equal to or greater than the middle key.
YES	YES	1. Split the leaf page. 2. Records with keys < middle key go to the left leaf page. 3. Records with keys >= middle key go to the right leaf page. 4. Split the index page. 5. Keys < middle key go to the left index page. 6. Keys > middle key go to the right index page. 7. The middle key goes to the next (higher level) index. IF the next level index page is full, continue splitting the index pages.

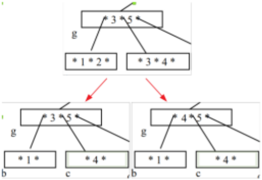
Διαγραφή σε B+tree : Περιπτώσεις

The `delete` algorithm for B+ Trees

Leaf Page Below Fill Factor	Index Page Below Fill Factor	Action
NO	NO	Delete the record from the leaf page. Arrange keys in ascending order to fill void. If the key of the deleted record appears in the index page, use the next key to replace it.
YES	NO	Combine the leaf page and its sibling. Change the index page to reflect the change.
YES	YES	1. Combine the leaf page and its sibling. 2. Adjust the index page to reflect the change. 3. Combine the index page with its sibling. Continue combining index pages until you reach a page with the correct fill factor or you reach the root page.

Διαγραφή σε B+tree

- Σε κάποιες παραλλαγές του αλγορίθμου διαγραφής[2] , σε περίπτωση διαγραφής της πρώτης εγγραφής σε ένα φύλλο, ανανεώνεται και η τιμή του κλειδιού που δείχνει στο φύλο. Εντούτοις, το συγκεκριμένο βήμα δεν είναι απαραίτητο και μπορεί να παραληφθεί.
- Στο επόμενο παράδειγμα για την διαγραφή της εγγραφής 3, παρουσιάζονται οι δύο εναλλακτικοί τρόποι. Στα αριστερά η διαγραφή πραγματοποιείται χωρίς ανανέωση του κλειδιού, ενώ στα δεξιά με ανανέωση αντίστοιχα.



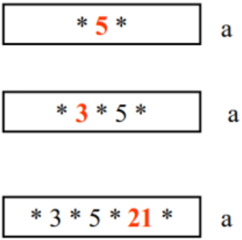
[2] Η συγκεκριμένη προσέγγιση ακολουθείται στις σημειώσεις του μαθήματος

Παράδειγμα διεργασιών σε B+tree

- Εστω το B+Tree τάξεως 4.
- Δείξτε βήμα-βήμα το B+ δένδρο μετά την εισαγωγή των 5, 3, 21, 9, 1, 13, 2, 7, 1 12, 4, 8 και την διαγραφή 2, 21, 10, 3, 4

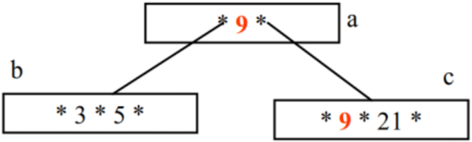
Παράδειγμα διεργασιών σε B+tree

Εισαγωγή 5, 3, 21



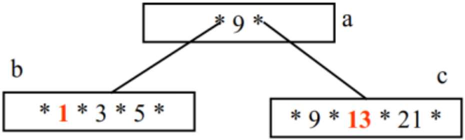
Παράδειγμα διεργασιών σε B+tree

Εισαγωγή 9



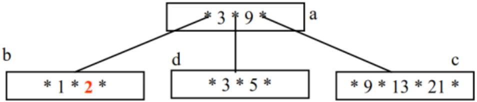
Παράδειγμα διεργασιών σε B+tree

Εισαγωγή 1,13



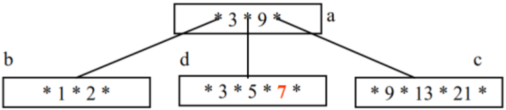
Παράδειγμα διεργασιών σε B+tree

Εισαγωγή 2



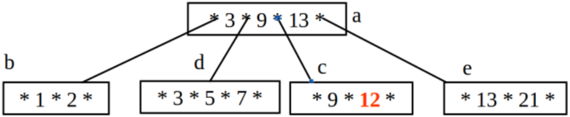
Παράδειγμα διεργασιών σε B+tree

Εισαγωγή 7



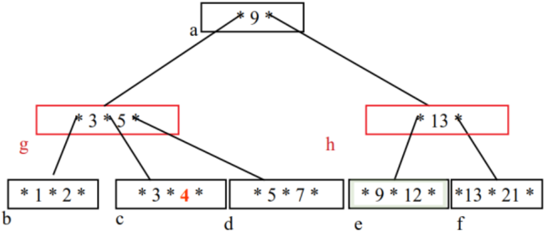
Παράδειγμα διεργασιών σε B+tree

Εισαγωγή 12



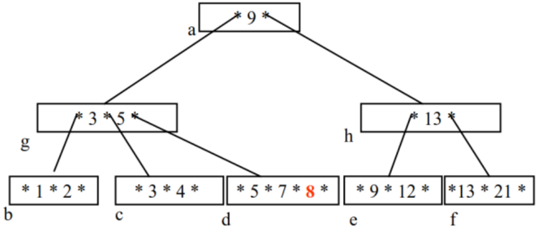
Παράδειγμα διεργασιών σε B+tree

Εισαγωγή 4



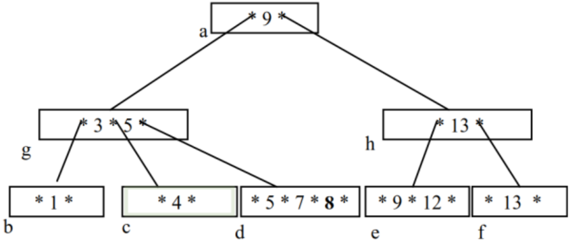
Παράδειγμα διεργασιών σε B+tree

Εισαγωγή 8



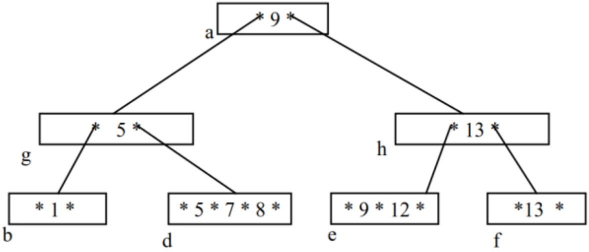
Παράδειγμα διεργασιών σε B+tree

Διαγραφή 2,21,3

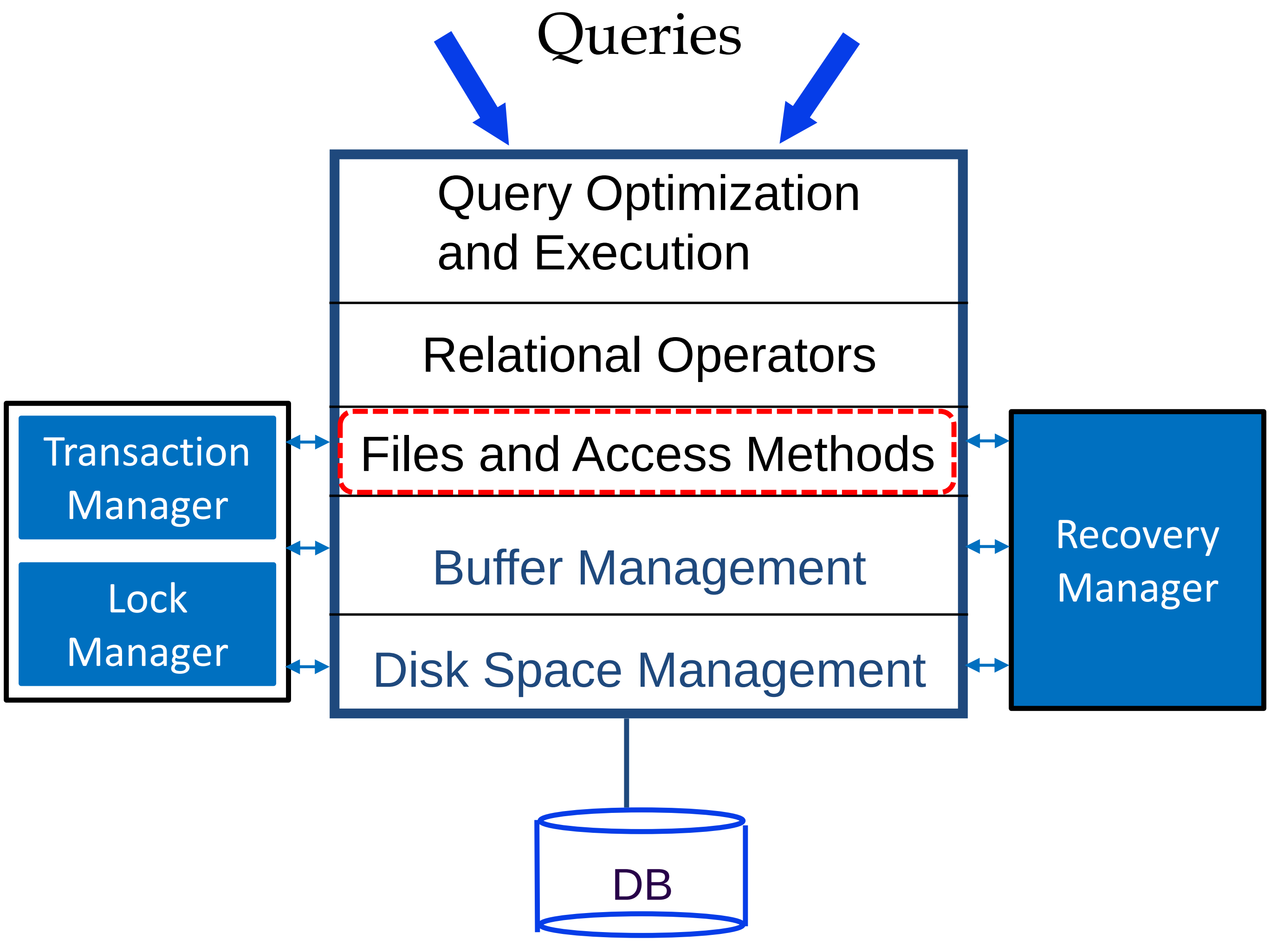


Παράδειγμα διεργασιών σε B+tree

Διαγραφή 4



DBMS Layers



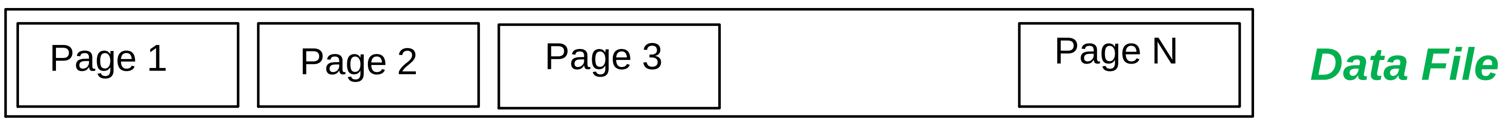
Queries
Query Optimization
and Execution
Relational Operators
Files and Access Methods
Buffer Management
Disk Space Management
Transaction
Manager
Lock
Manager
Recovery
Manager
DB

Outline

- Why Indexing? ✓
- Storing Data Records and Index Types
- Indexed Sequential Access Method (ISAM) Trees
- B+ Trees

Motivation

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

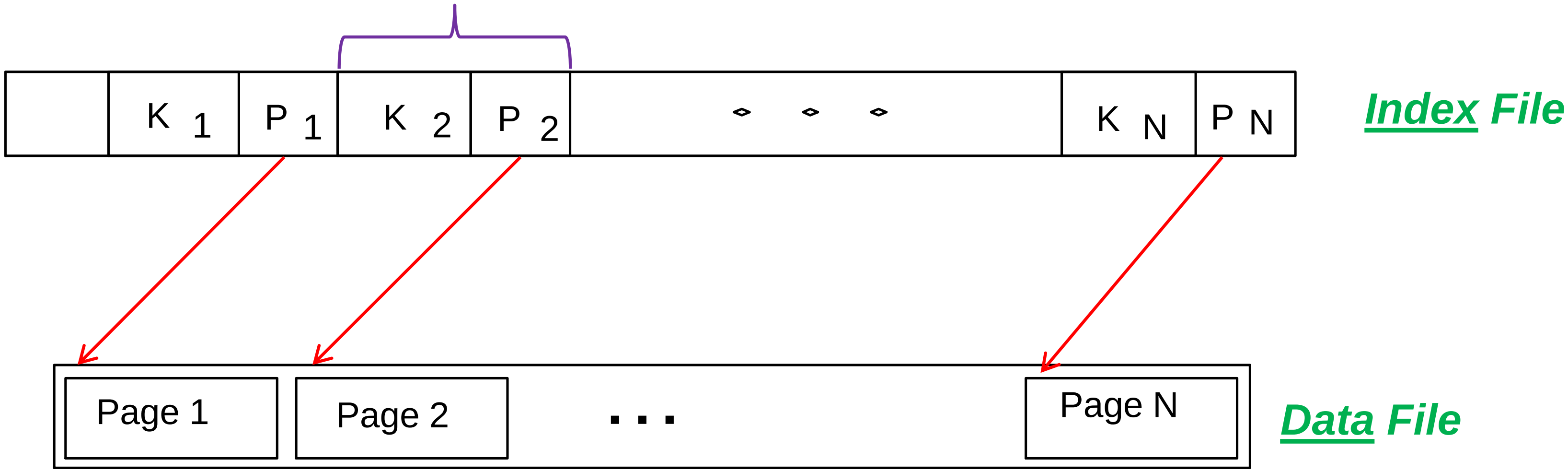


- How can we answer a *range selection* (E.g., *"Find all students with a GPA higher than 3.0"*)?
  - What about doing a *binary search* followed by a *scan*?
    - Yes, but…
  - What if the file becomes "very" large?
    - Cost is proportional to the number of pages fetched
    - Hence, may become very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

Motivation

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

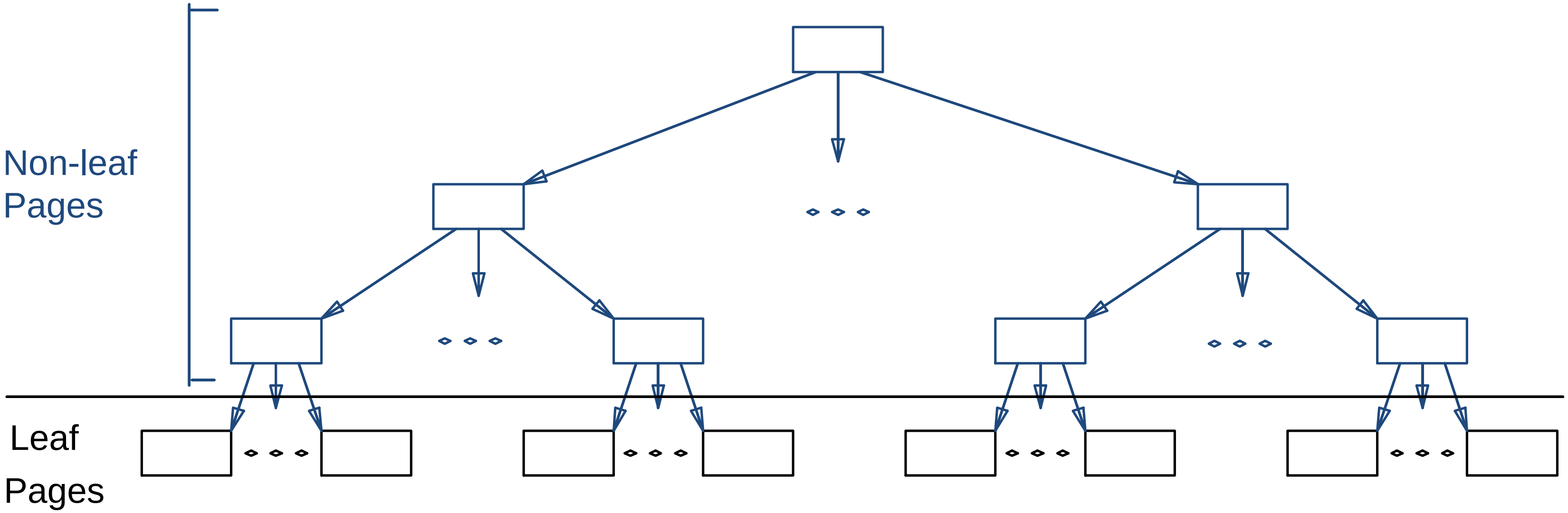
Index Entry = <first key on the page, pointer to the page>



- 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

- Why Indexing?
- Storing Data Records and Index Types ✓
- Indexed Sequential Access Method (ISAM) Trees
- B+ Trees

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

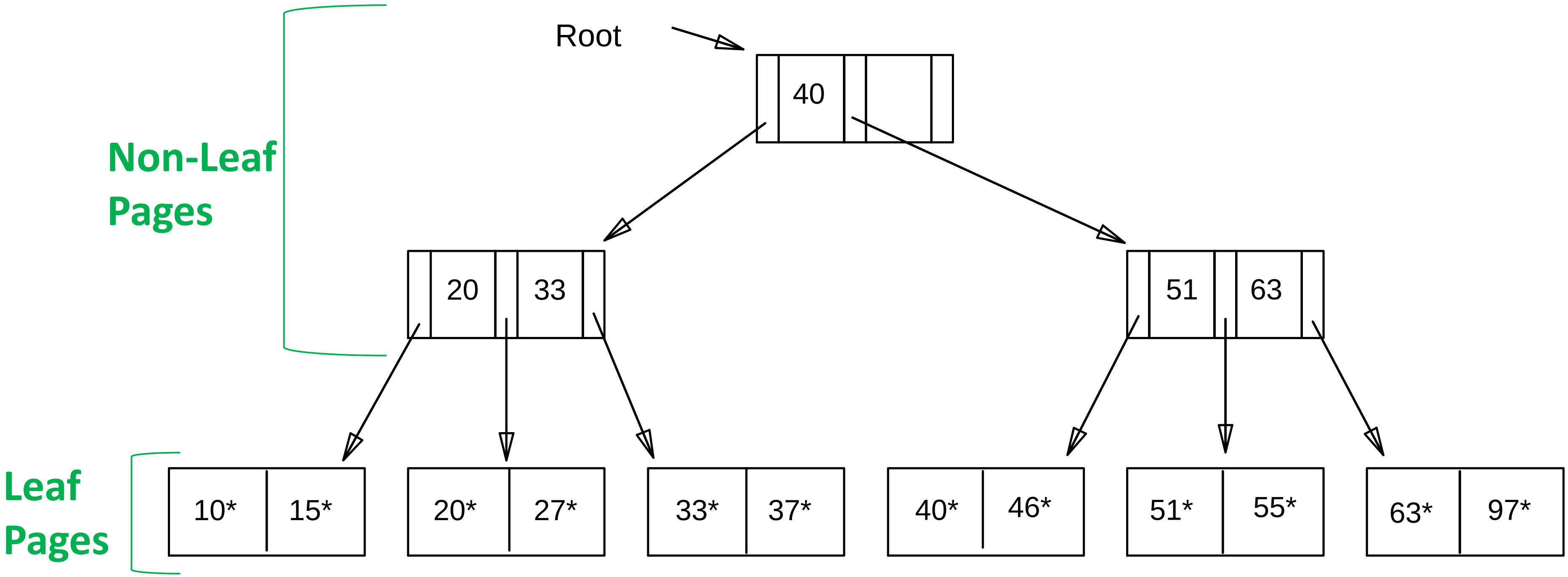
- Is an index that uses Alternative (1) clustered or un-clustered?
  - Clustered
- Is an index that uses Alternative (2) or (3) clustered or un-clustered?
  - Clustered "only" if data records are sorted on the search key field
- In practice:
  - A clustered index is an index that uses Alternative (1)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

Outline

- Why Indexing?
- Storing Data Records and Index Types
- Indexed Sequential Access Method (ISAM) Trees ✓
- B+ Trees

ISAM Trees

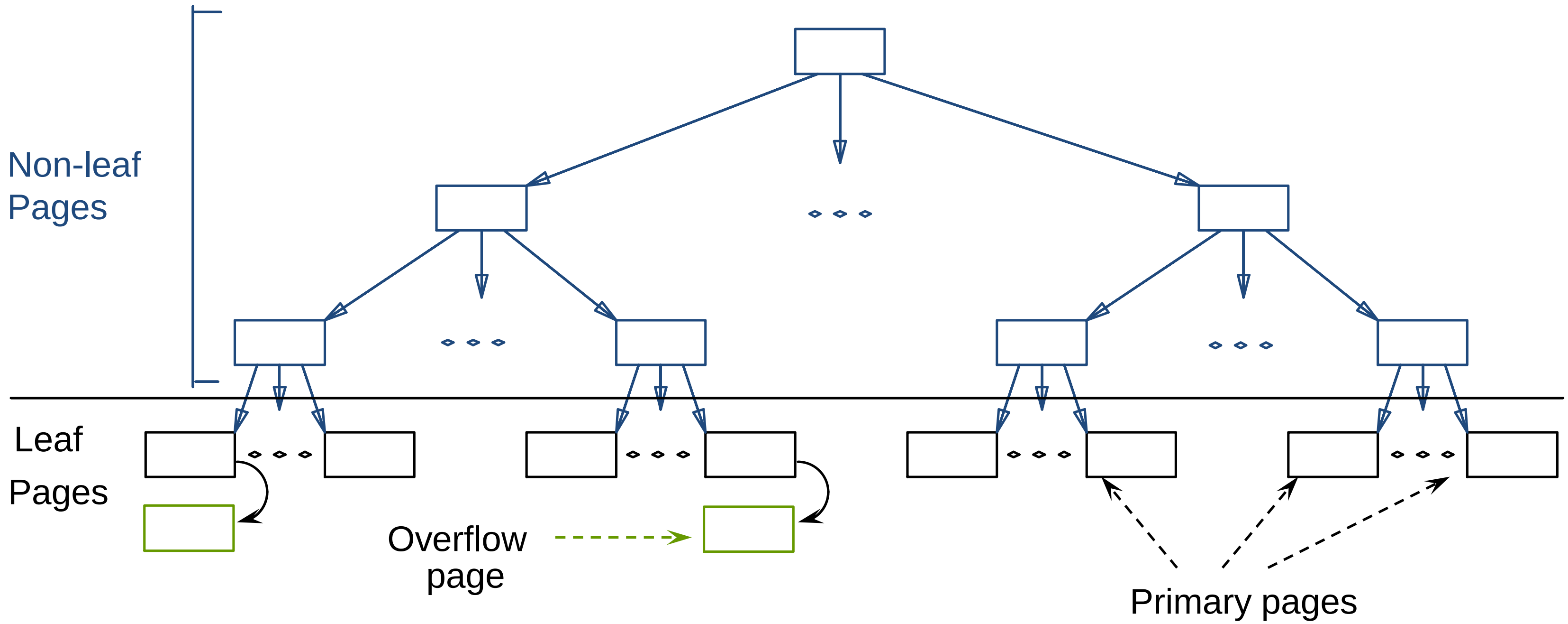
- Indexed Sequential Access Method (ISAM) trees are *static*



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

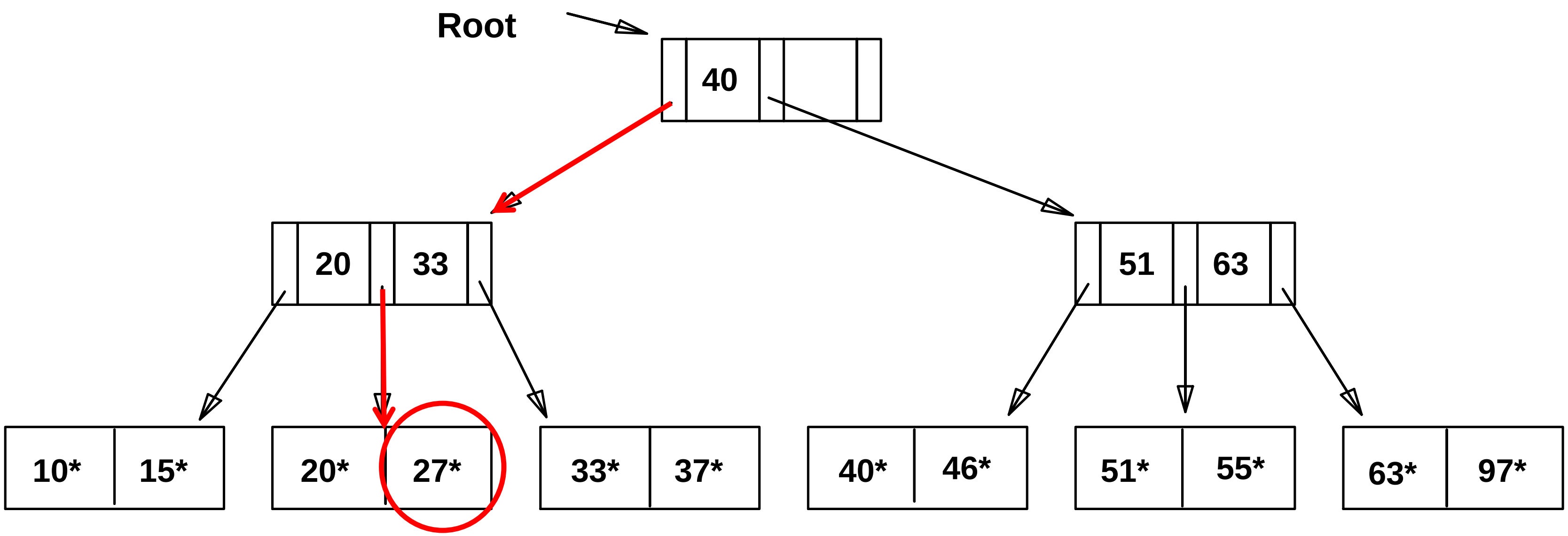
  - If Alternative (2) or (3) is used, the data records are created and *sorted* before allocating leaf pages

  - The non-leaf pages are subsequently allocated

ISAM: Searching for Entries

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

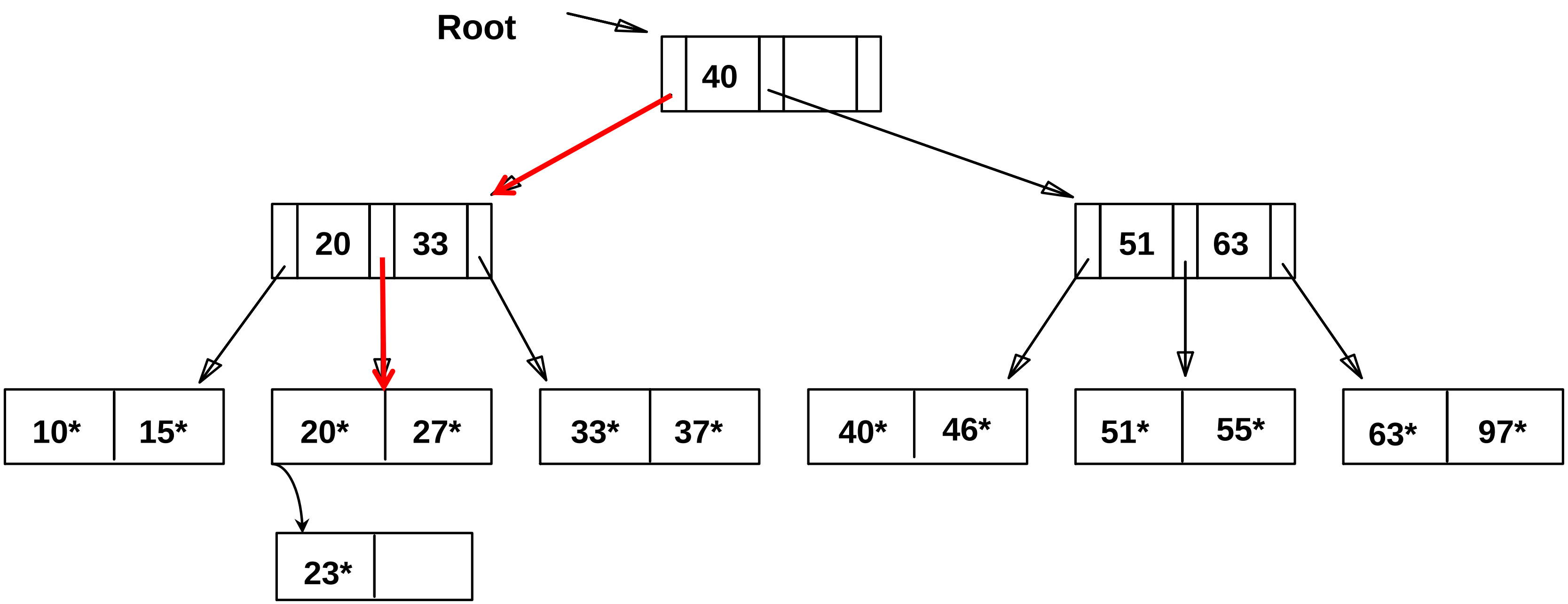
- Search for 27*



ISAM: Inserting Entries

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

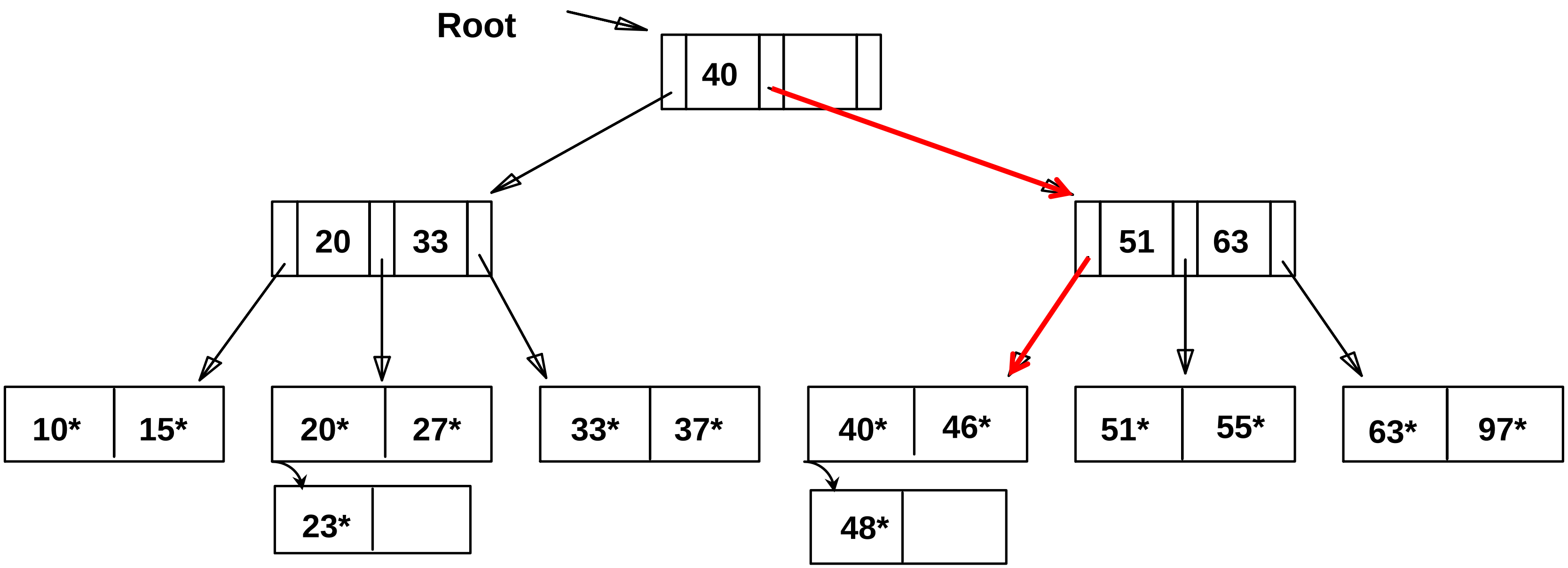
- Insert 23*



ISAM: Inserting Entries

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

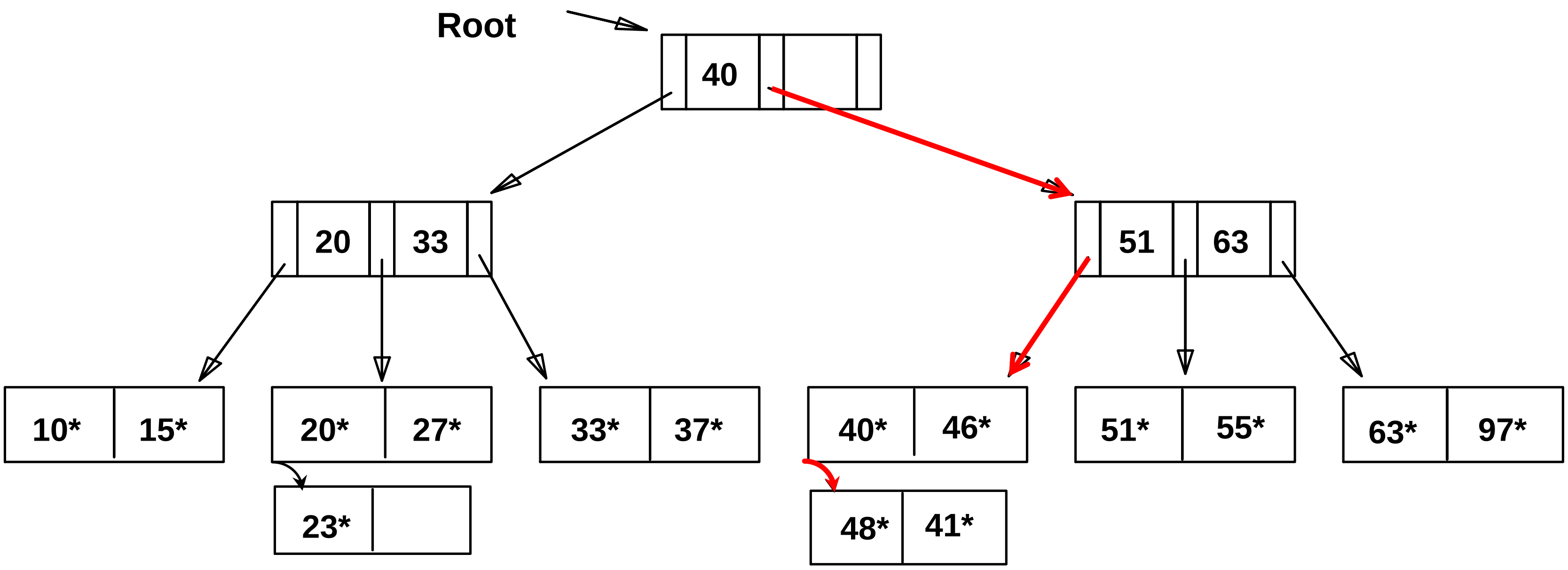
- Insert 48*



ISAM: Inserting Entries

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

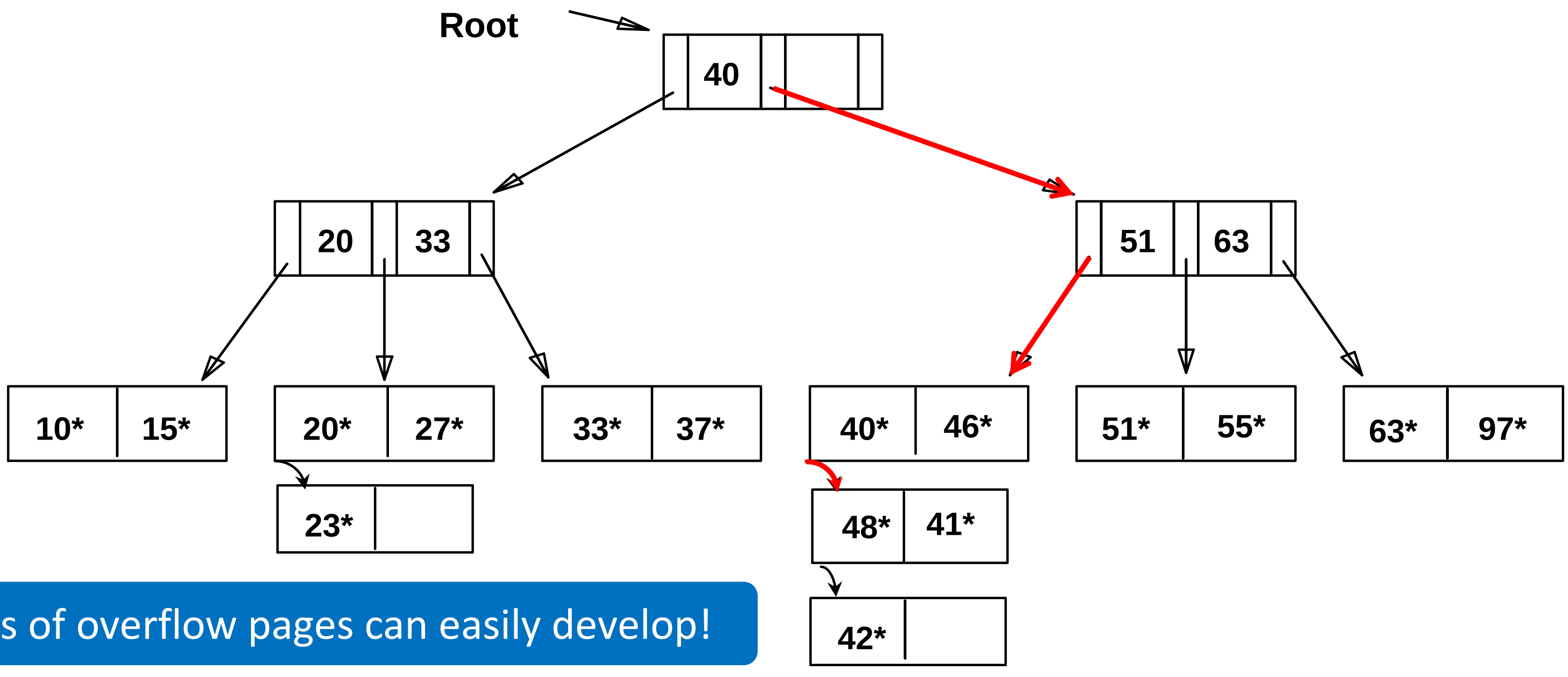
- Insert 41*



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*

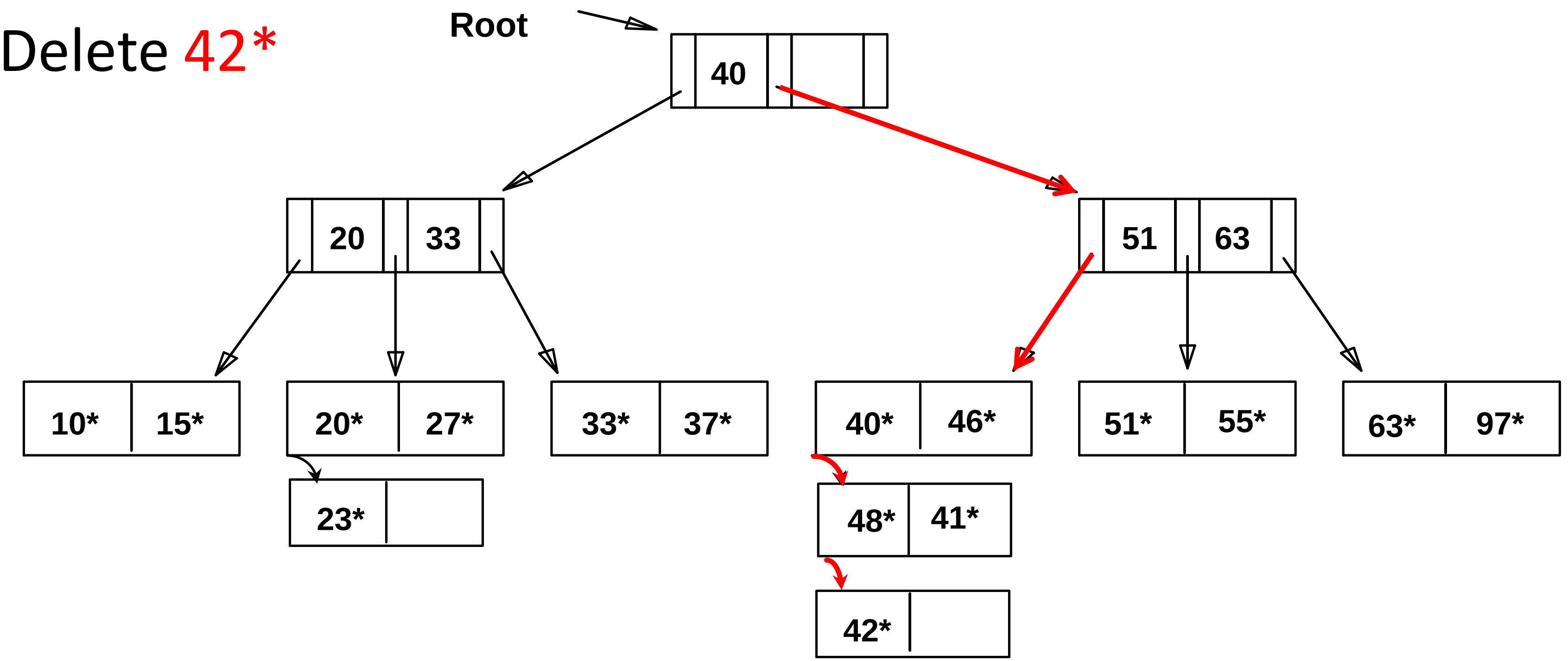


Chains of overflow pages can easily develop!

ISAM: Deleting Entries

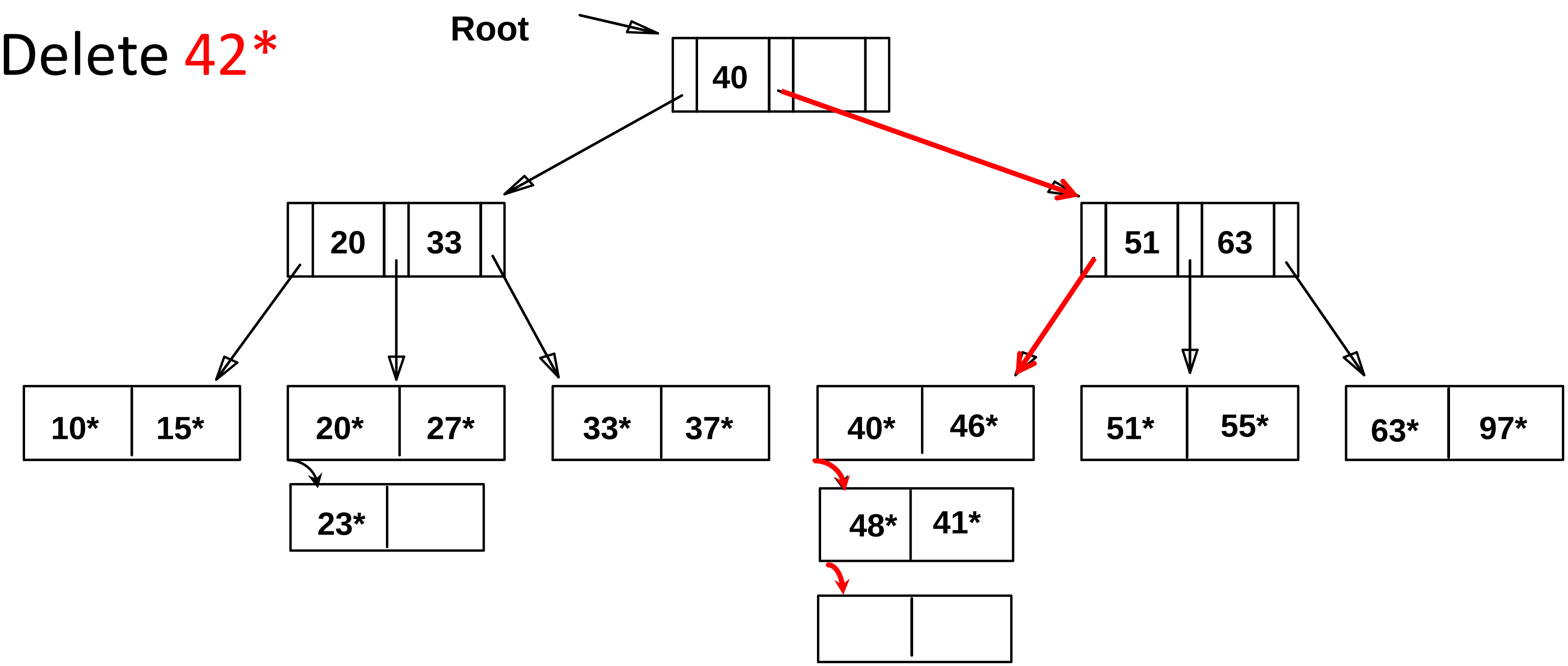
- The appropriate page is determined as for a search, and the entry is deleted (*with ONLY overflow pages removed when becoming empty*)

- Delete 42*



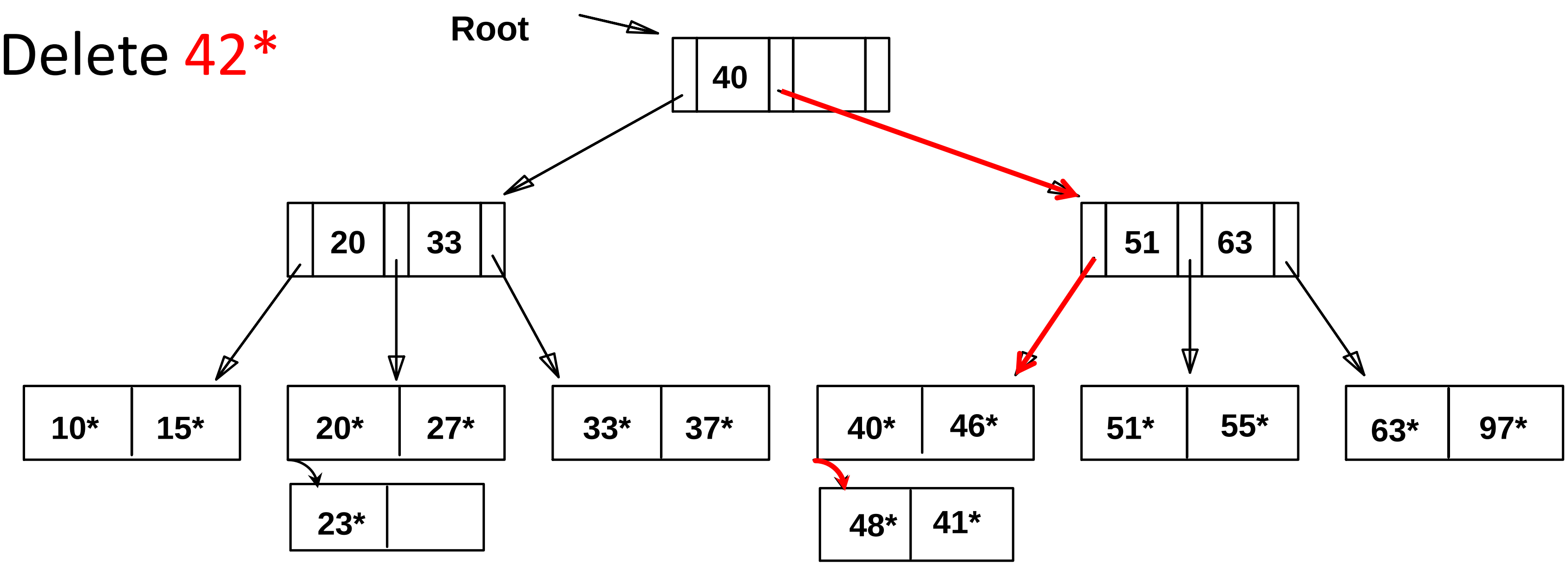
ISAM: Deleting Entries

- 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

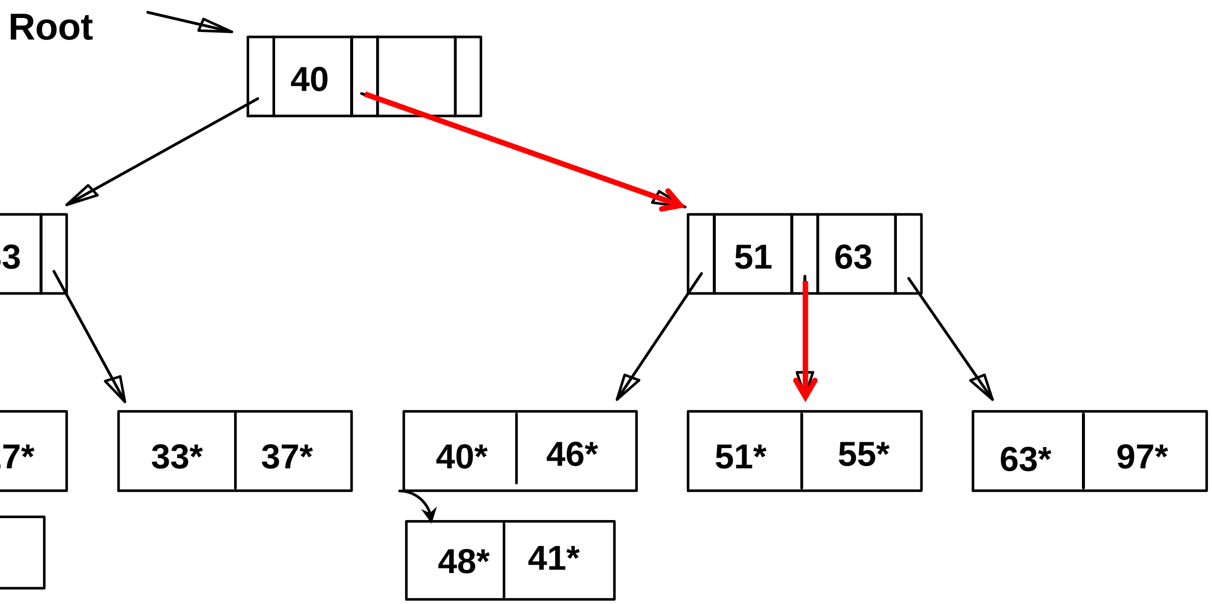
- The appropriate page is determined as for a search, and the entry is deleted (*with ONLY overflow pages removed when becoming empty*)
- Delete 42*



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*

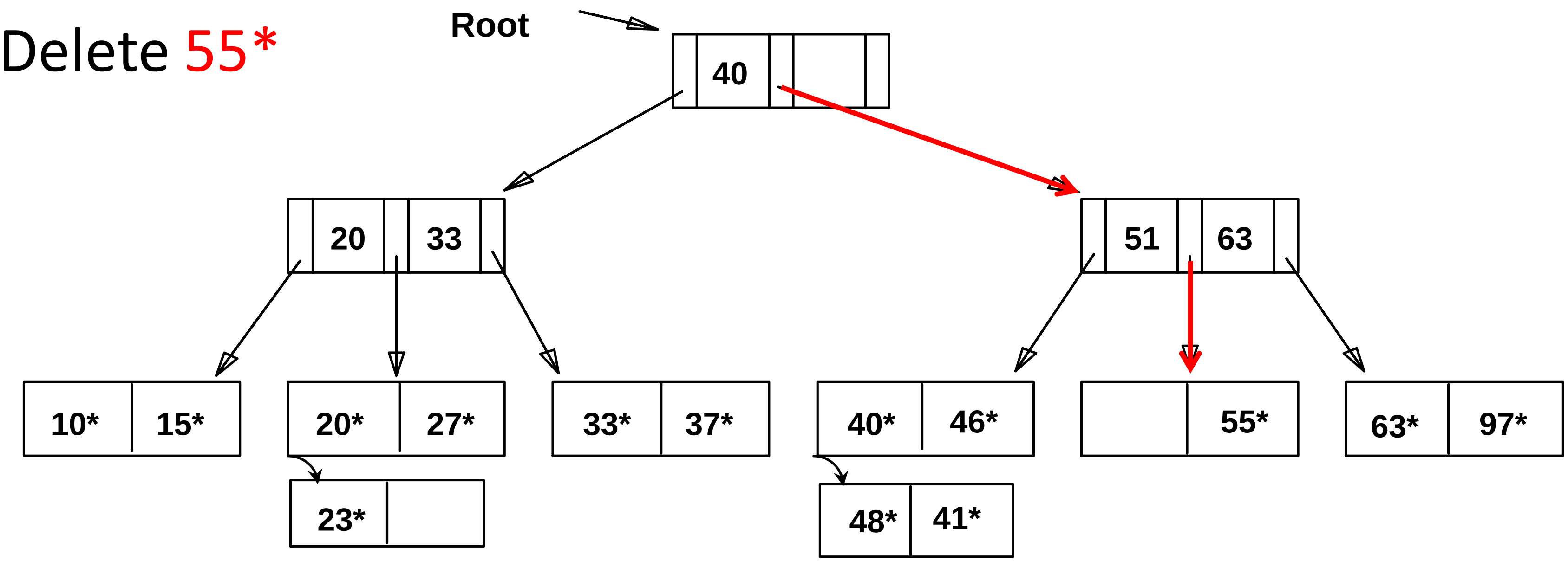


Note that 51 still appears in an index entry, but not in the leaf!

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

- Why Indexing?
- Storing Data Records in Indexes and Index Types
- Indexed Static Access Method (ISAM) Trees
- B+ Trees ✓

Dynamic Trees

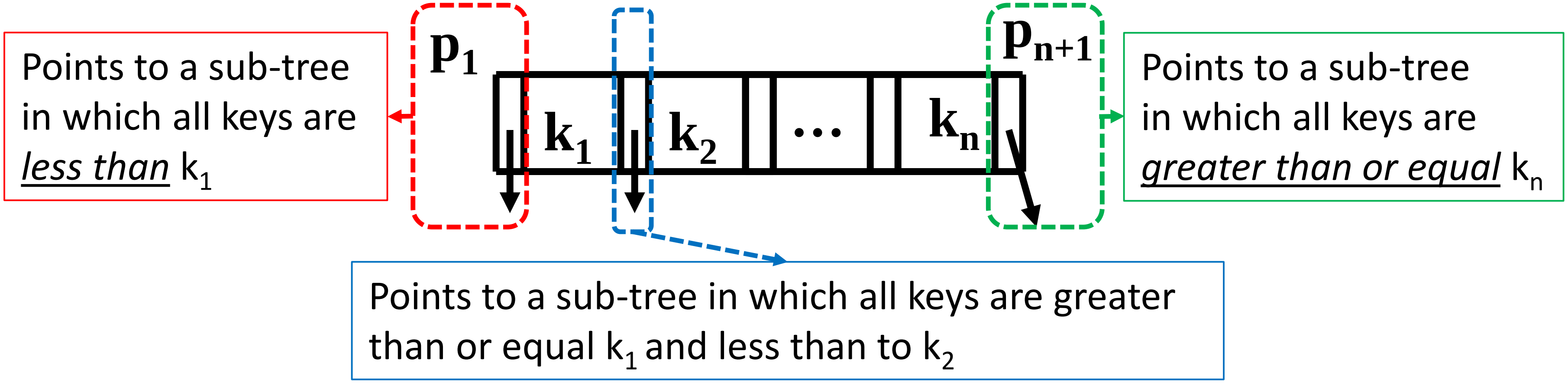
- ISAM indices are static
  - Long overflow chains can develop as the file grows, leading to poor performance

- 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

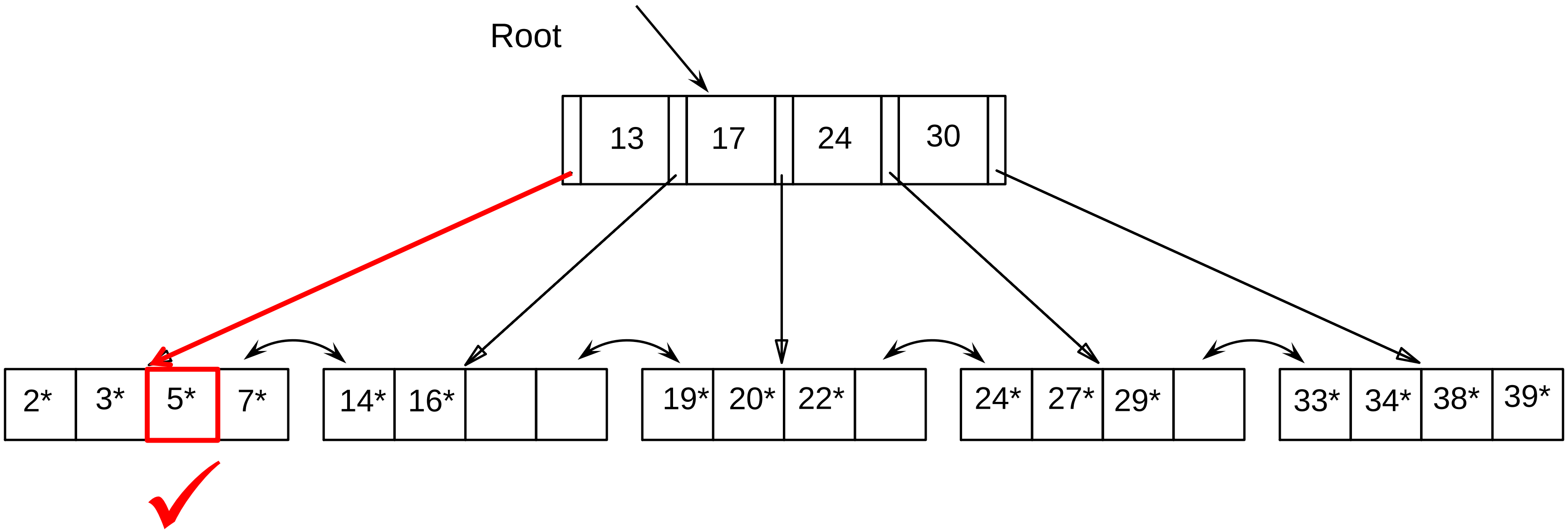
- Each node in a B+ tree of order ***d*** (this is a measure of the capacity of a tree):
  - Has at most ***2d*** keys
  - Has at least ***d*** keys (except the root, which may have just 1 key)
  - All leaves are on the same level
  - Has exactly ***n-1*** keys if the number of pointers is ***n***



B+ Tree: Searching for Entries

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

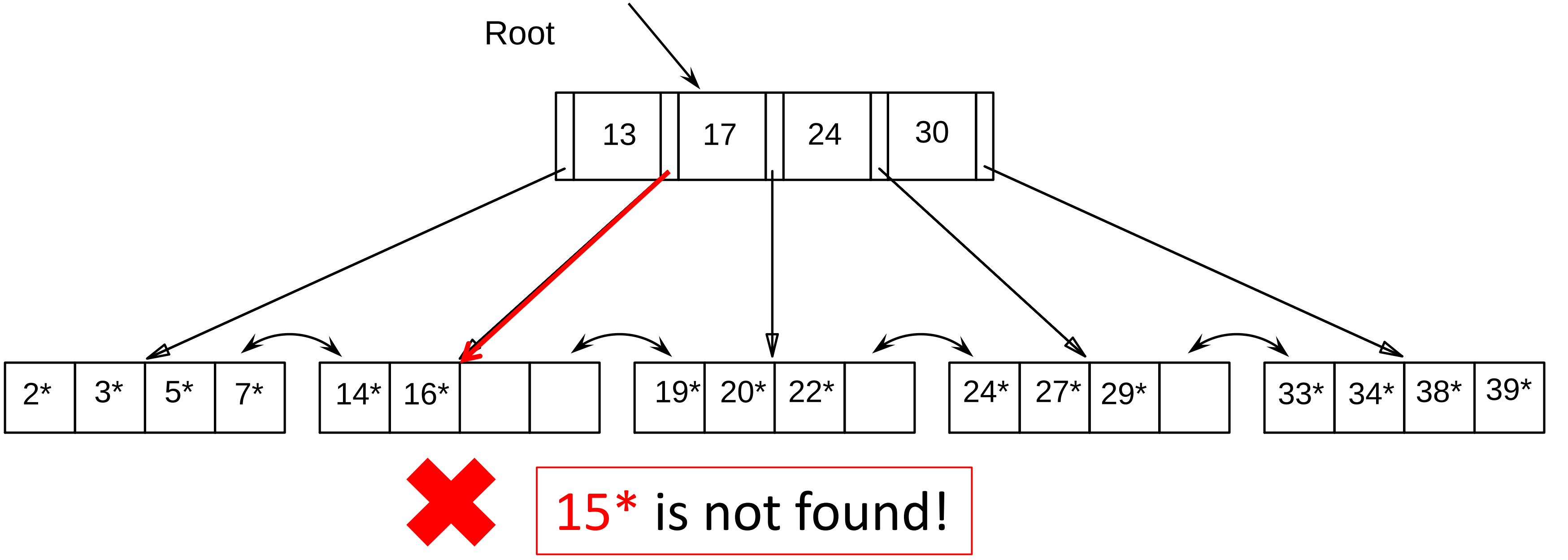
- Example 1: Search for entry 5*



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*



15* is not found!

B+ Trees: Inserting Entries

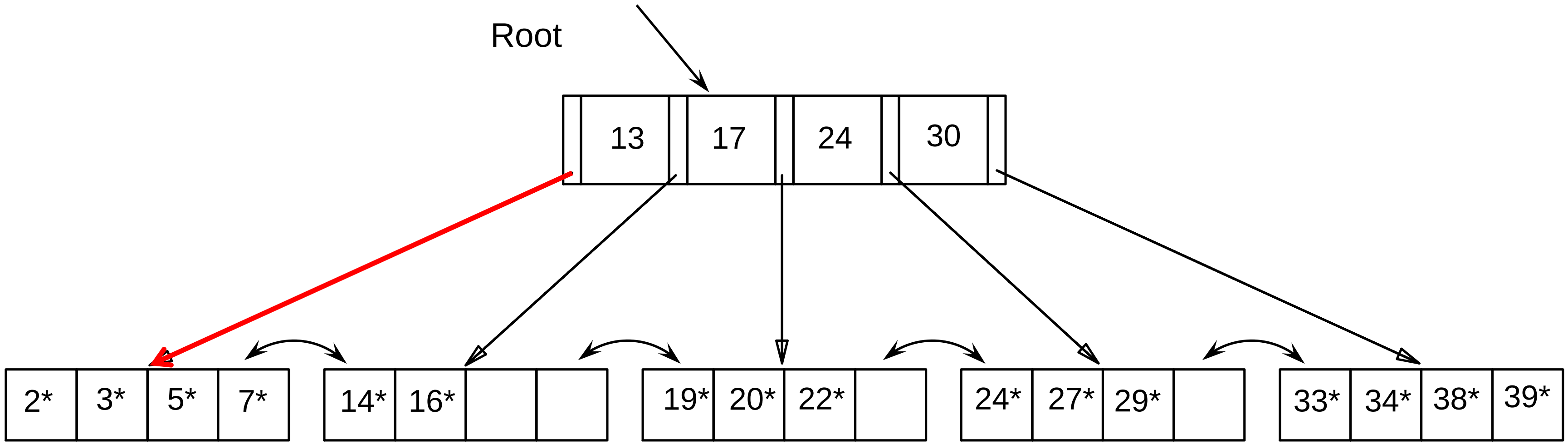
- Find correct leaf ***L***

- Put data entry onto ***L***
  - If ***L*** has enough space, *done*!
  - Else, *split* ***L*** *into* ***L*** *and a new node* $\boldsymbol{L_2}$
    - 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

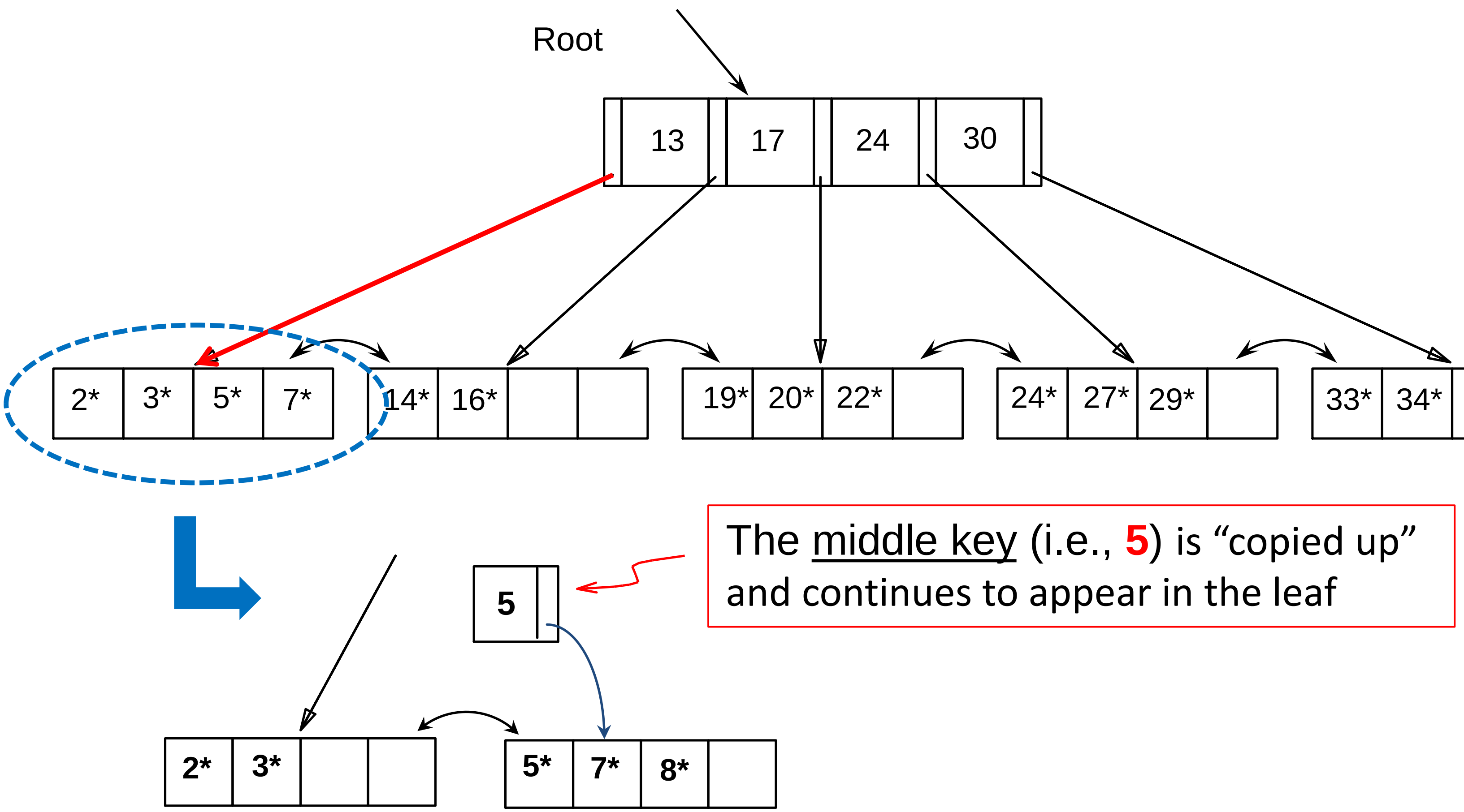
- Insert entry 8*



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

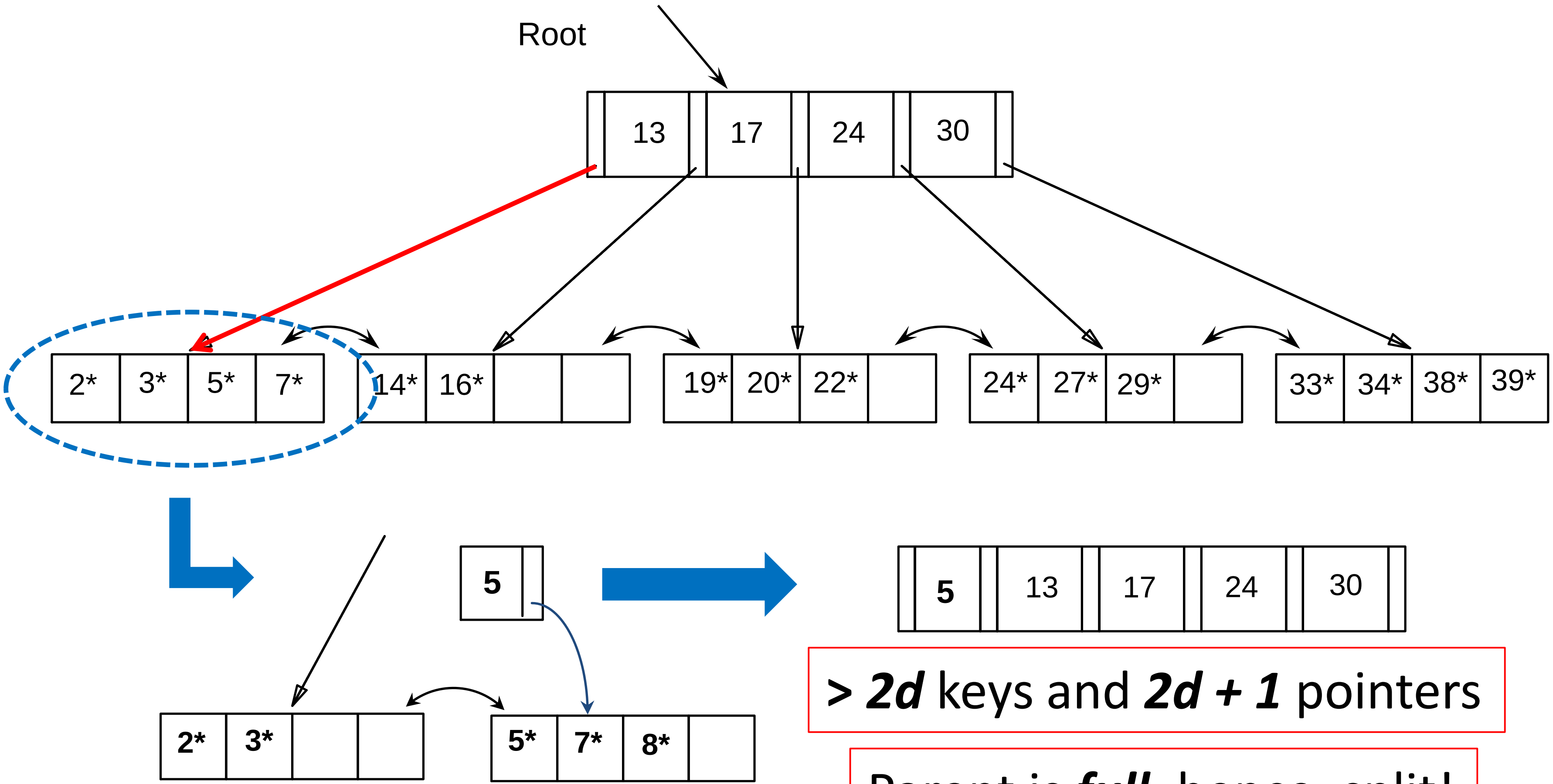
B+ Tree: Examples of Insertions

- Insert entry 8*



B+ Tree: Examples of Insertions

- Insert entry 8*

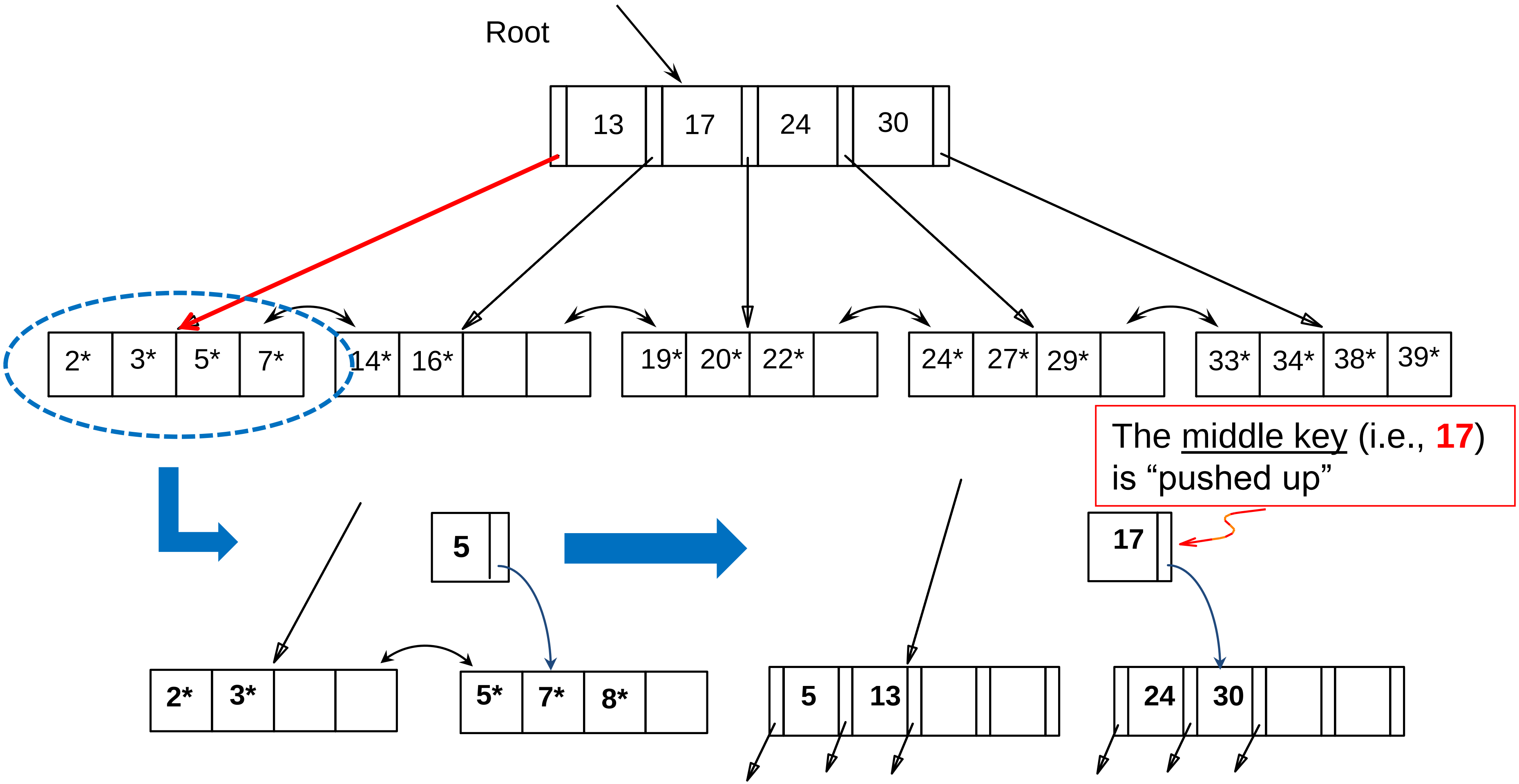


> ***2d*** keys and ***2d + 1*** pointers

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

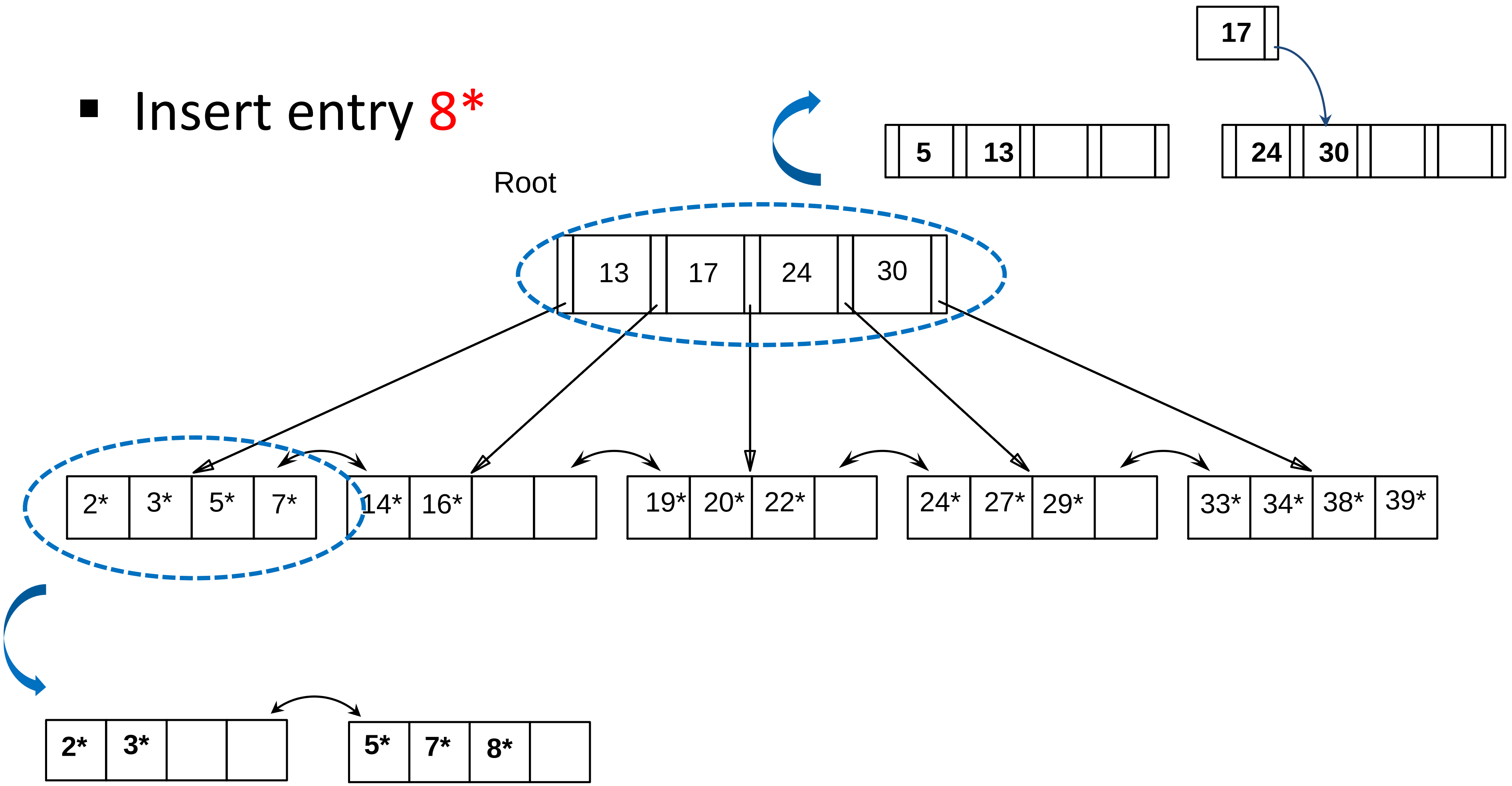
B+ Tree: Examples of Insertions

- Insert entry 8*



B+ Tree: Examples of Insertions

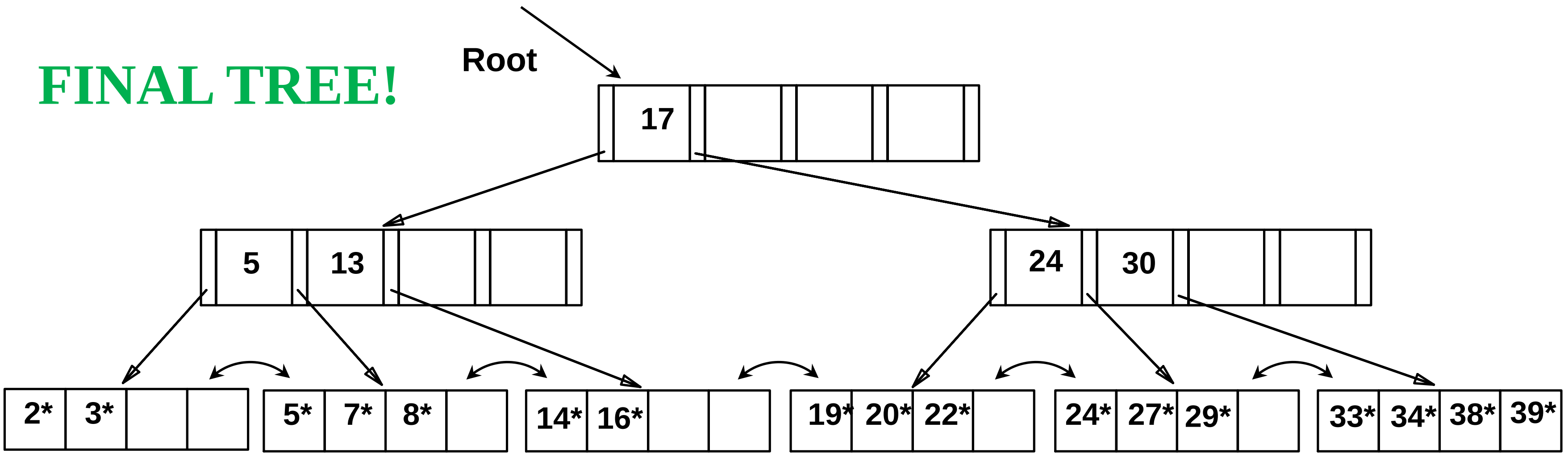
- Insert entry 8*



B+ Tree: Examples of Insertions

- Insert entry 8*

FINAL TREE!

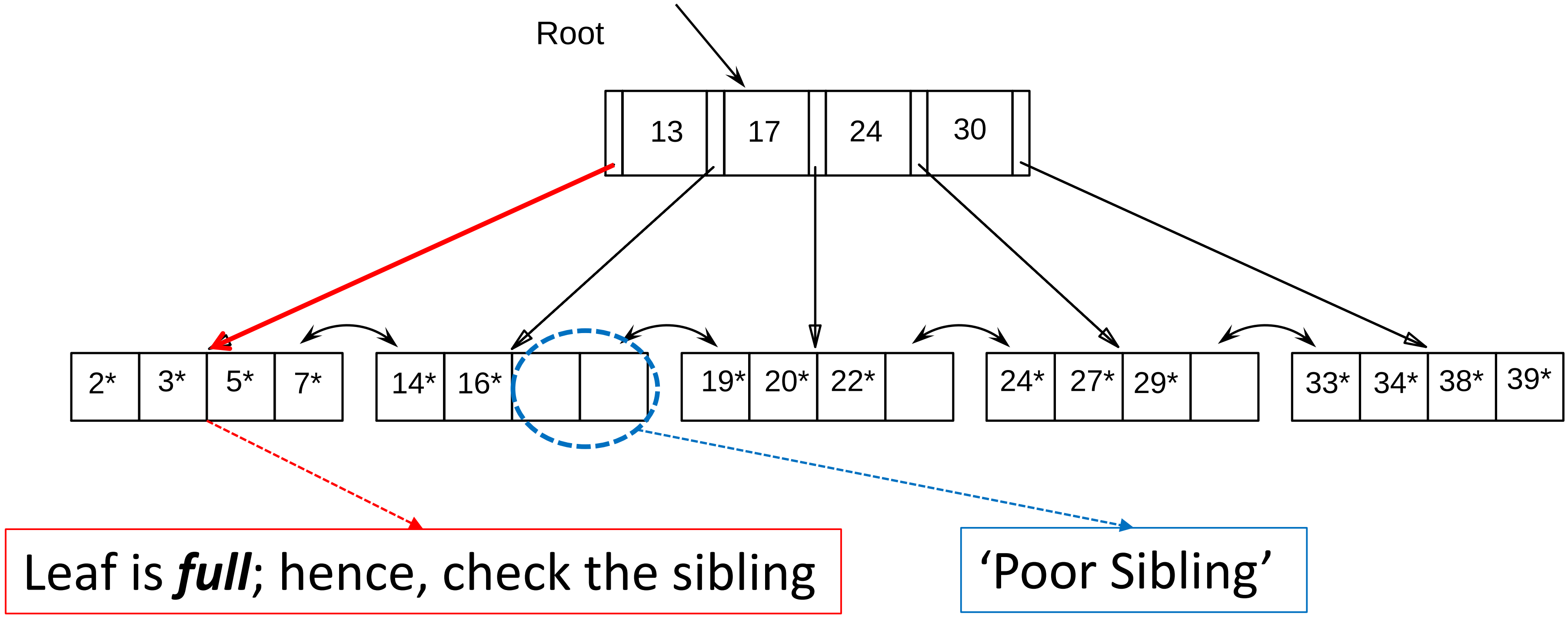


Splitting the root lead to an increase of height by 1!

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

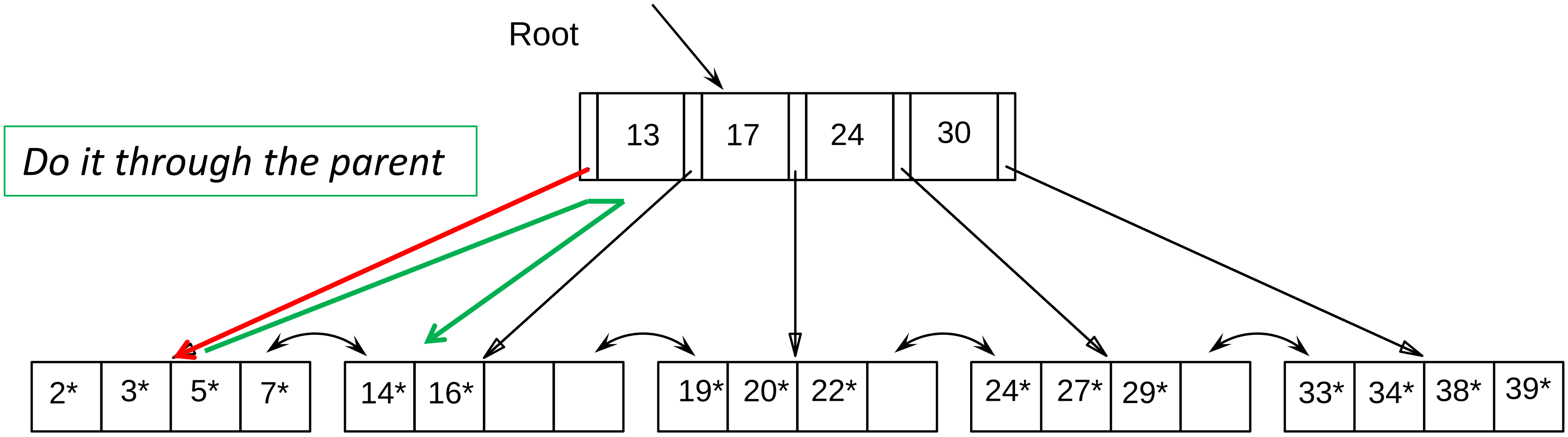
B+ Tree: Examples of Insertions

- Insert entry 8*



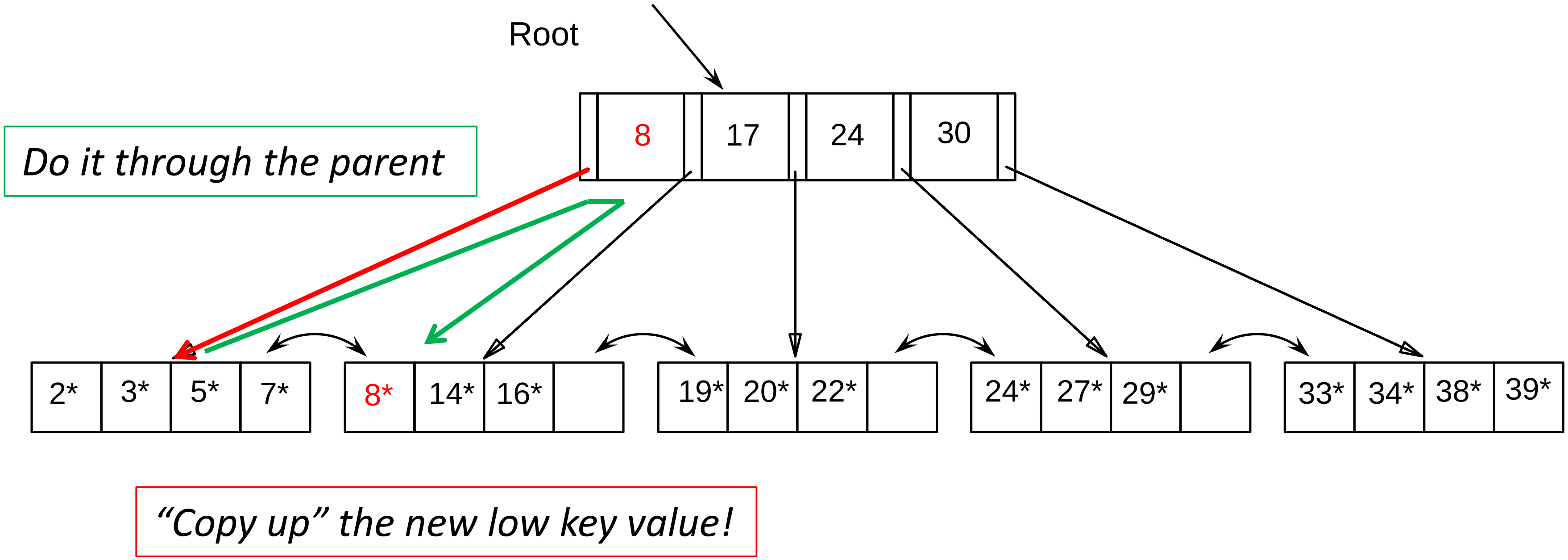
B+ Tree: Examples of Insertions

- 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

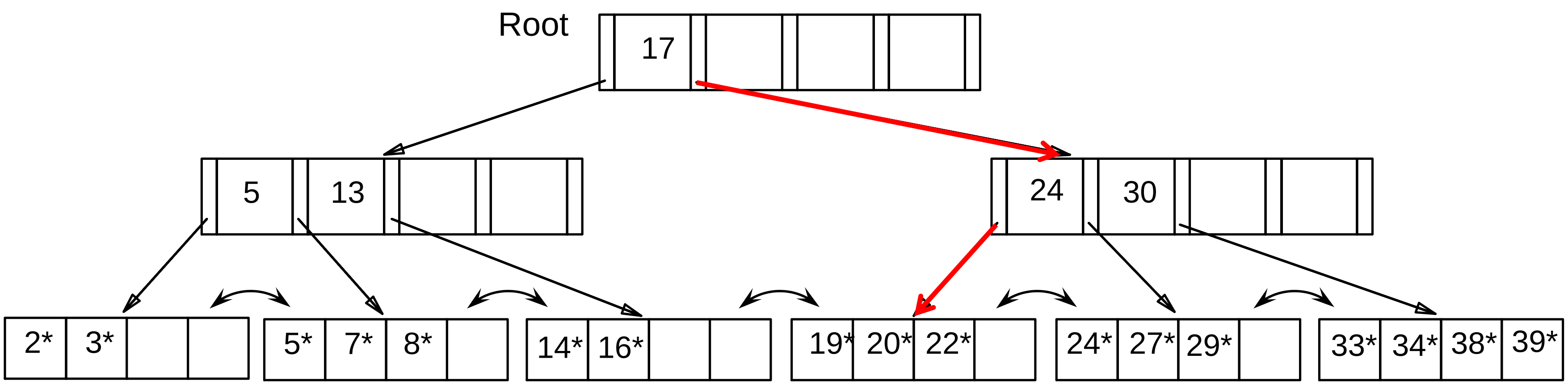
- Every data entry must appear in a leaf node; hence, "copy up" the middle key upon splitting
- When splitting index entries, simply "push up" the middle key
- Apply splitting and/or redistribution on leaf nodes
- Apply only splitting on non-leaf nodes

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

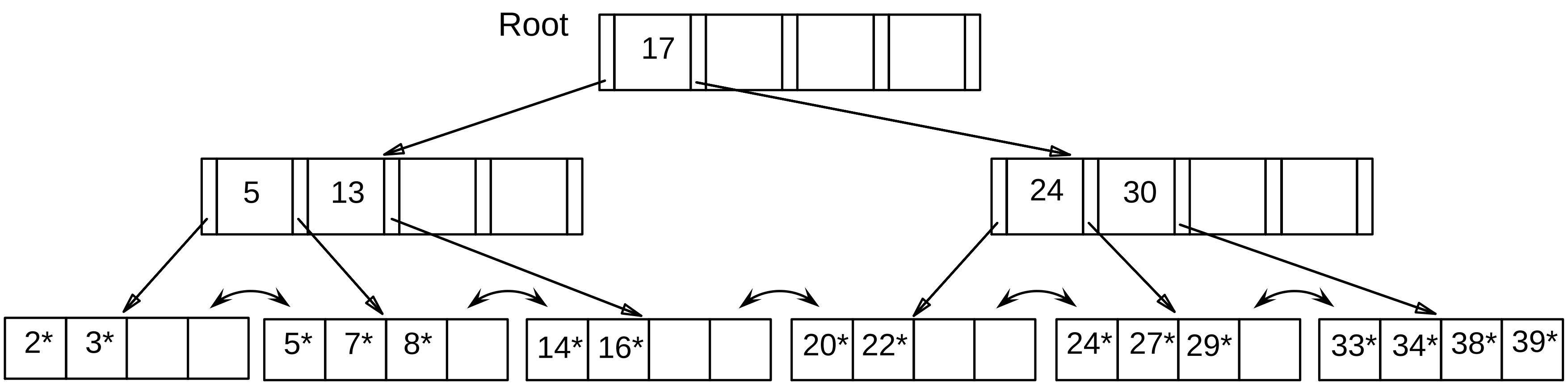
- Delete 19*



Removing 19* does not cause an underflow

B+ Tree: Examples of Deletions

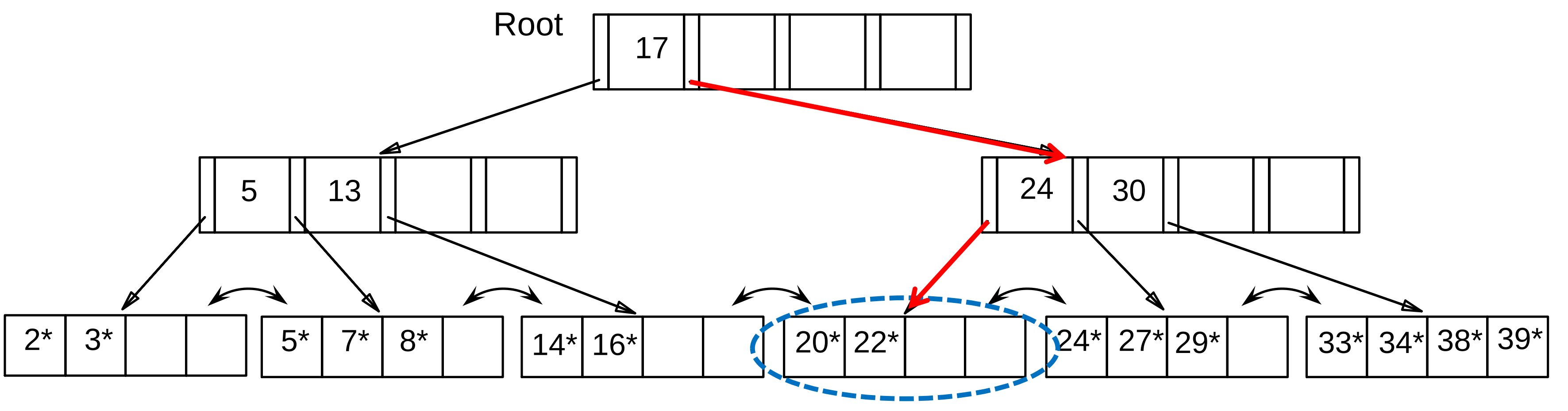
- Delete 19*



FINAL TREE!

B+ Tree: Examples of Deletions

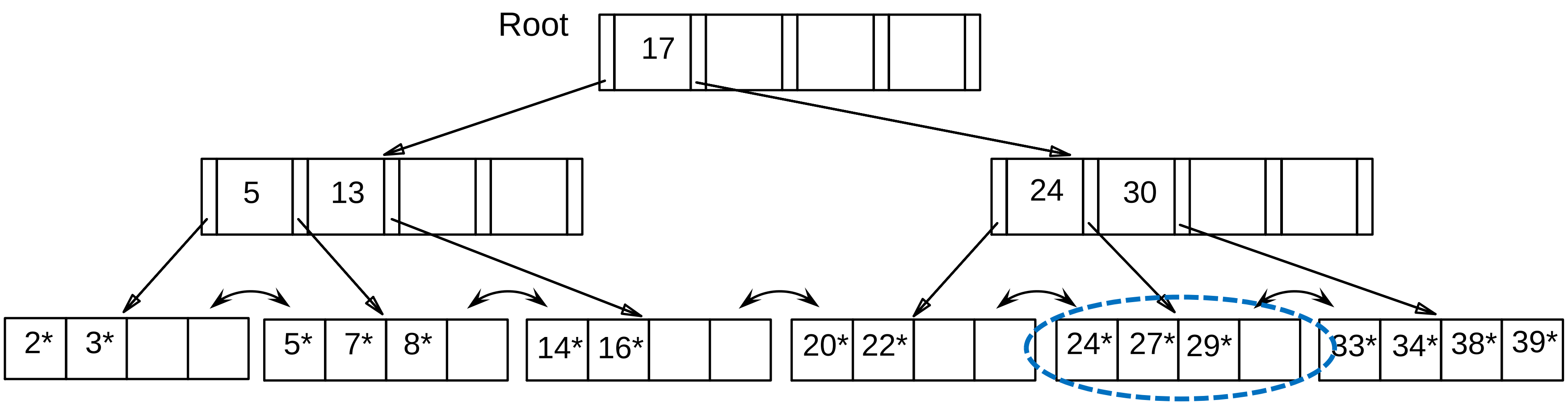
- Delete 20*



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

B+ Tree: Examples of Deletions

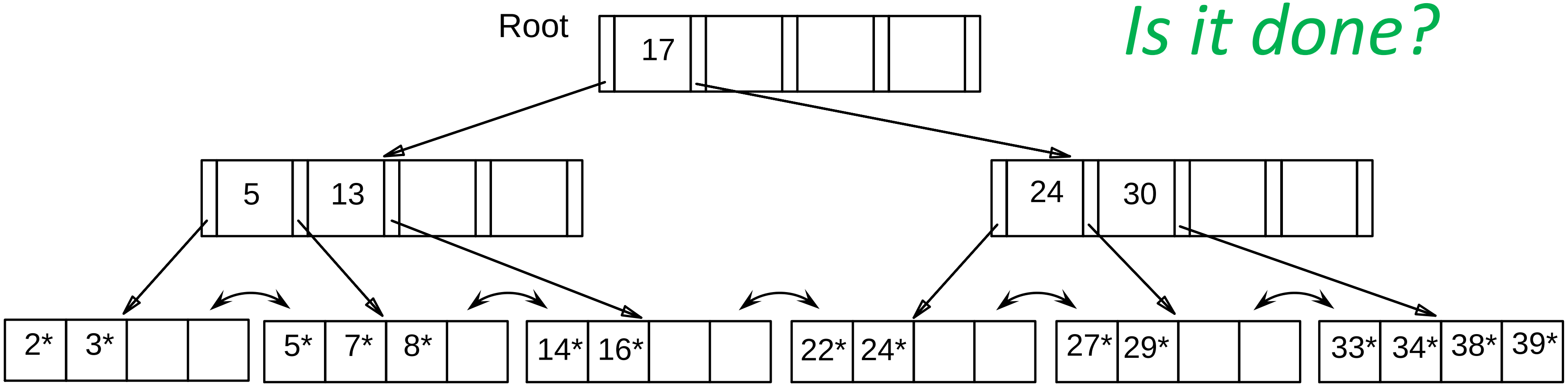
- Delete 20*



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

B+ Tree: Examples of Deletions

- Delete 20*

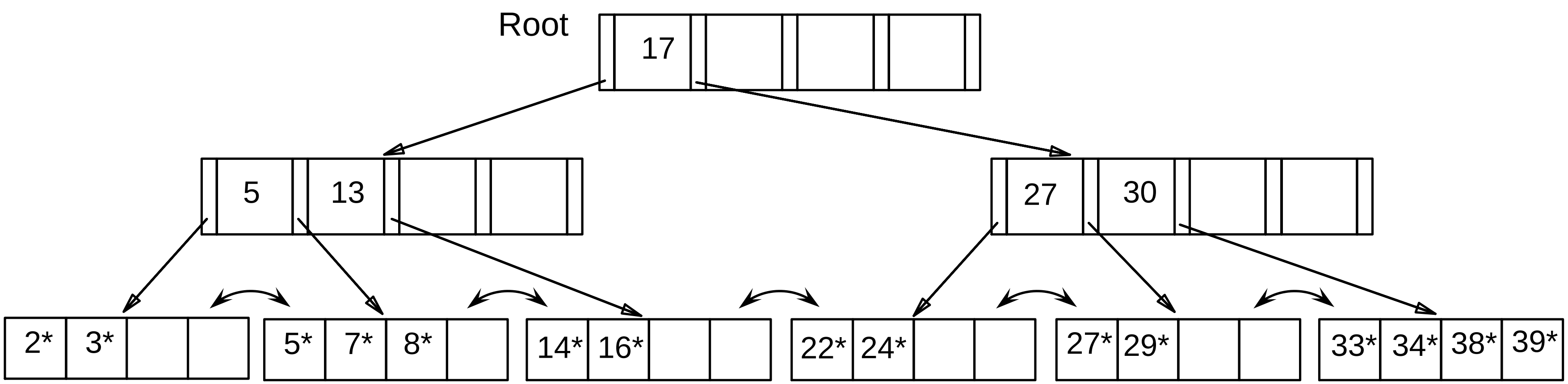


Is it done?

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

B+ Tree: Examples of Deletions

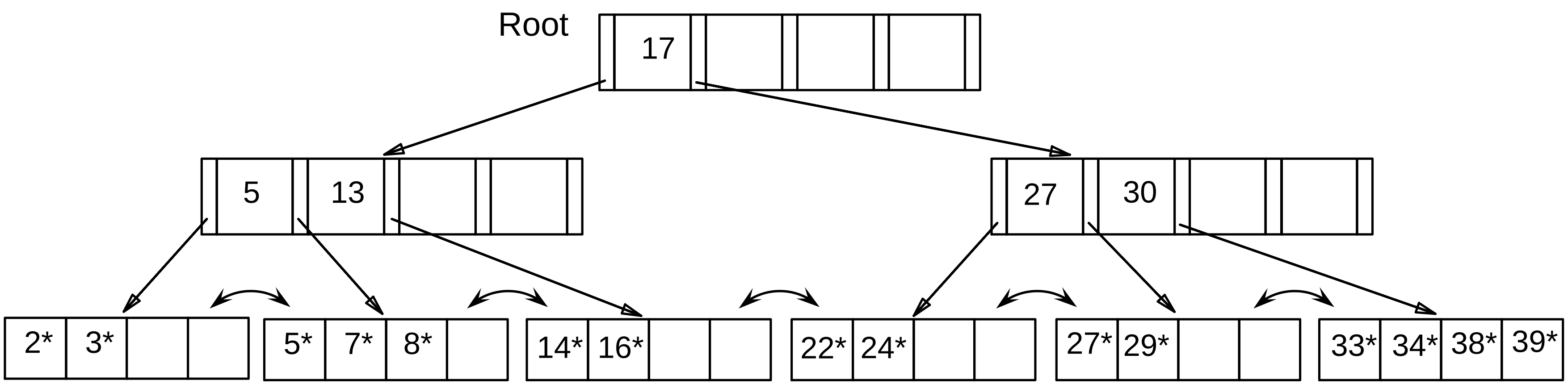
- Delete 20*



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

B+ Tree: Examples of Deletions

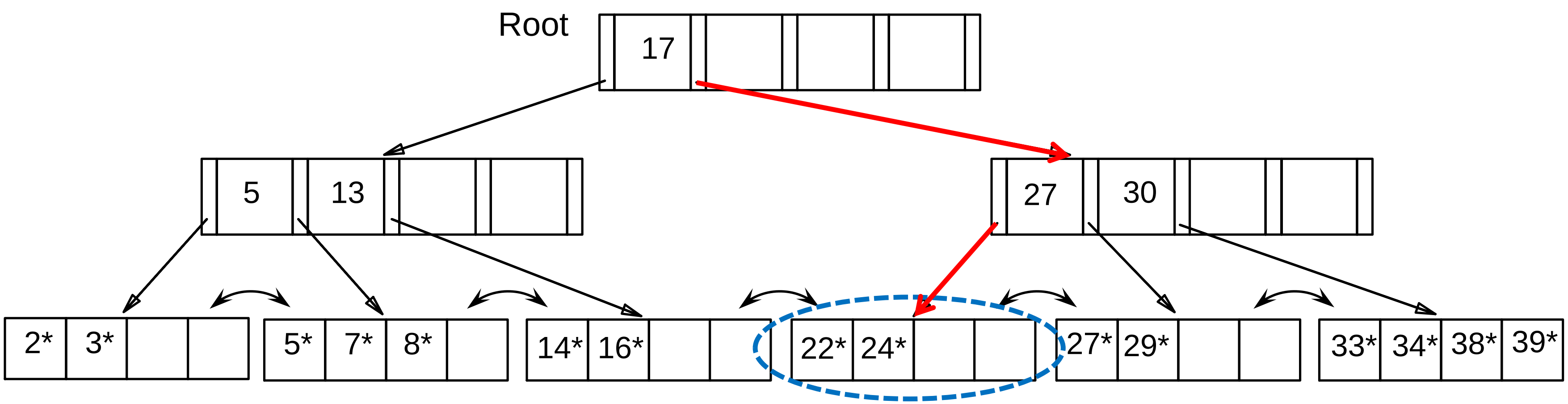
- Delete 20*



FINAL TREE!

B+ Tree: Examples of Deletions

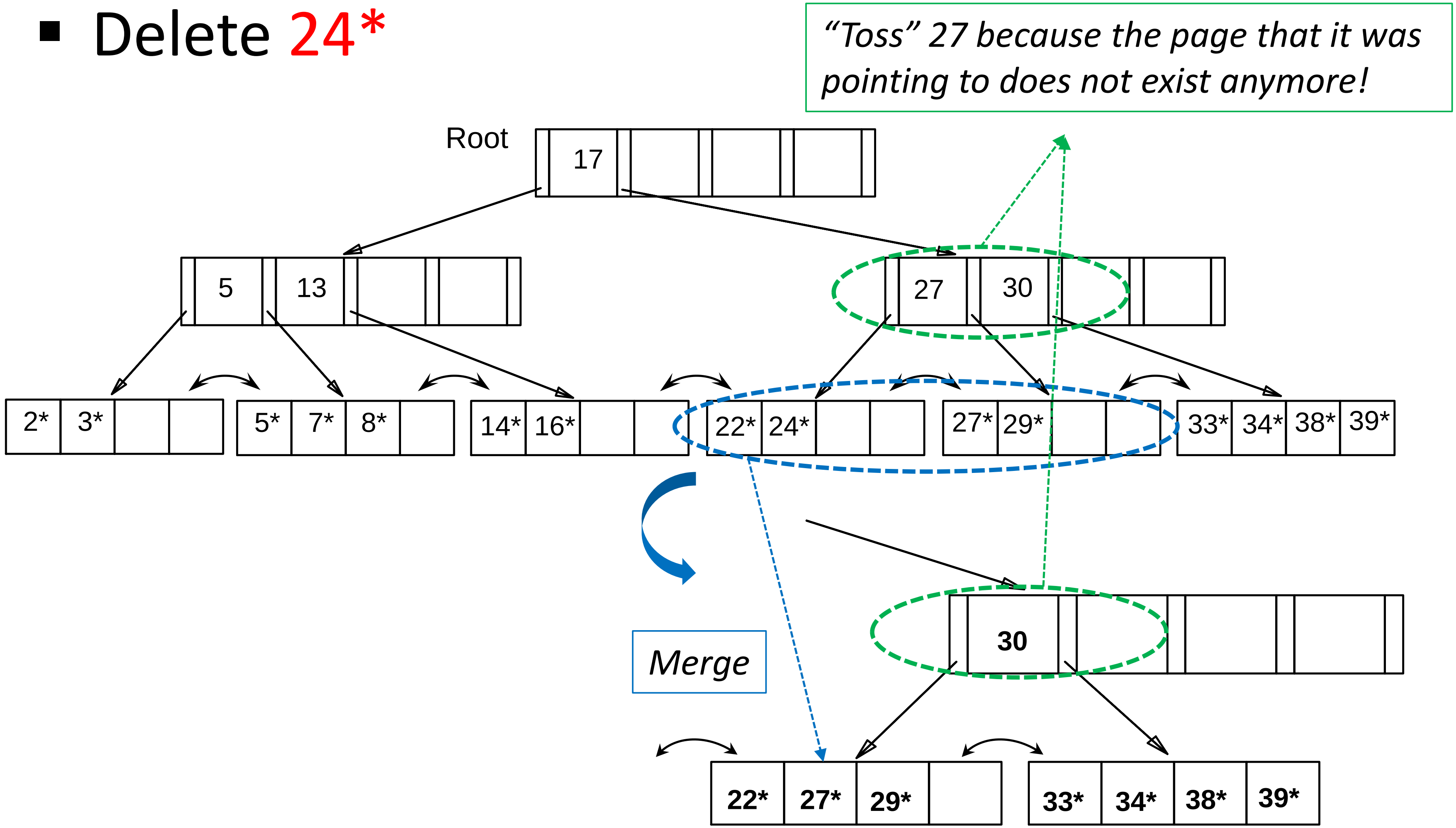
- Delete 24*



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*

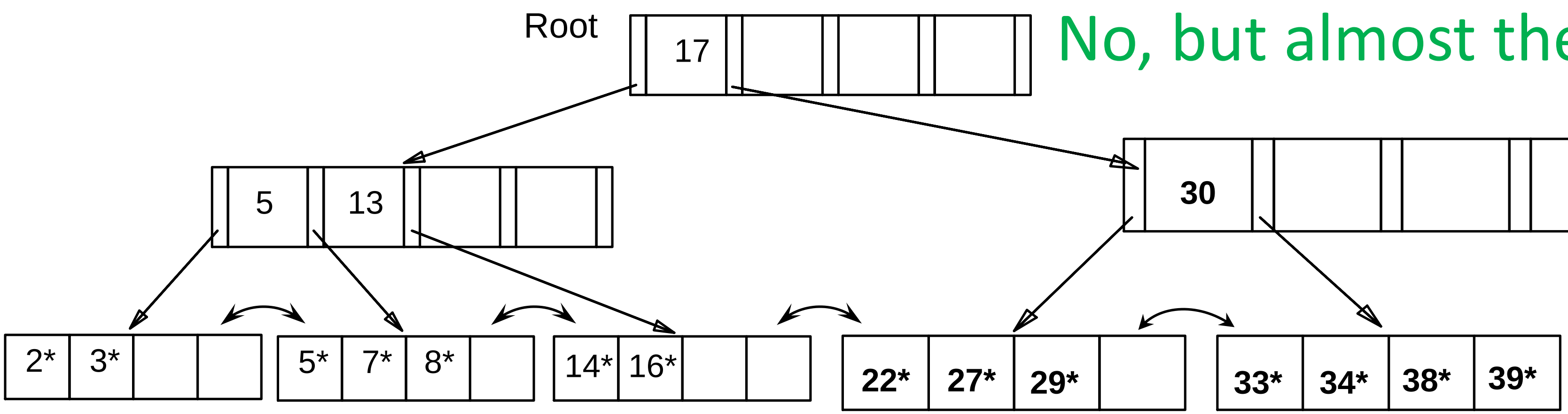


B+ Tree: Examples of Deletions

- Delete 24*

Is it done?

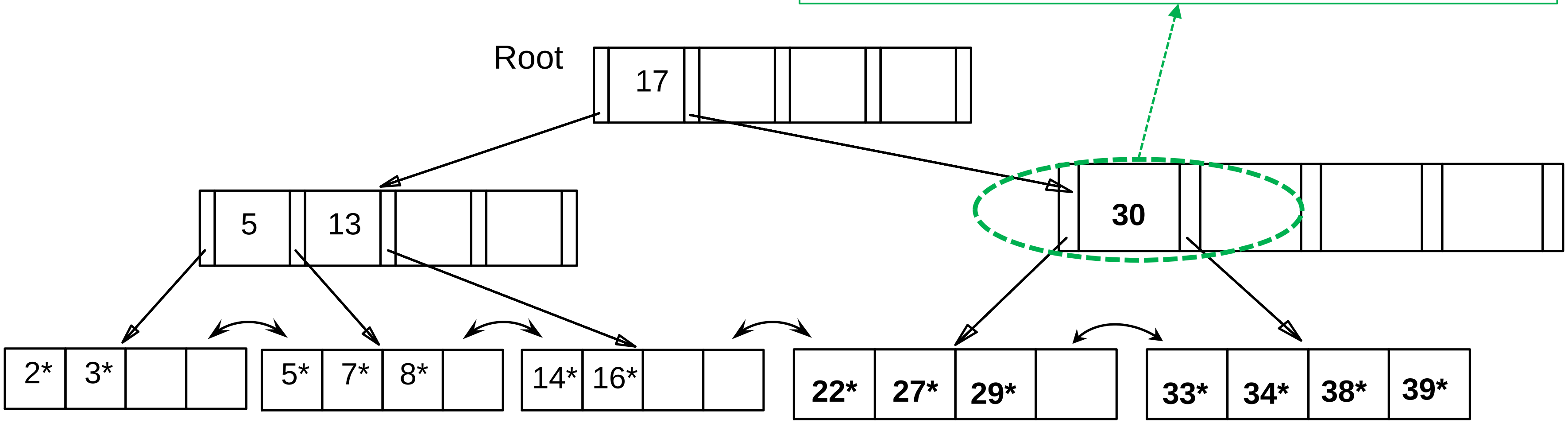
No, but almost there…



B+ Tree: Examples of Deletions

- Delete 24*

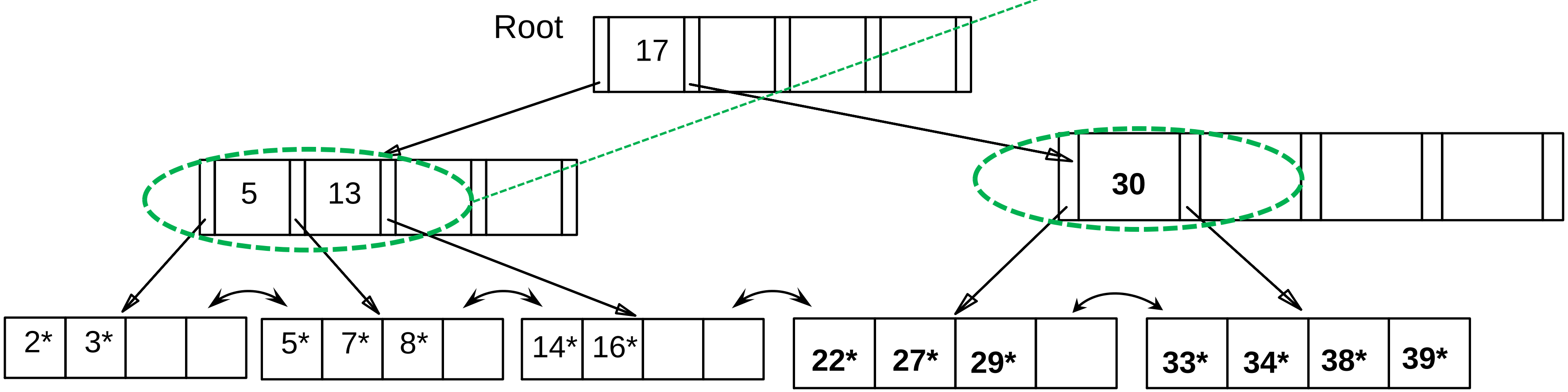
This entails an underflow; hence, we must either redistribute or merge!



B+ Tree: Examples of Deletions

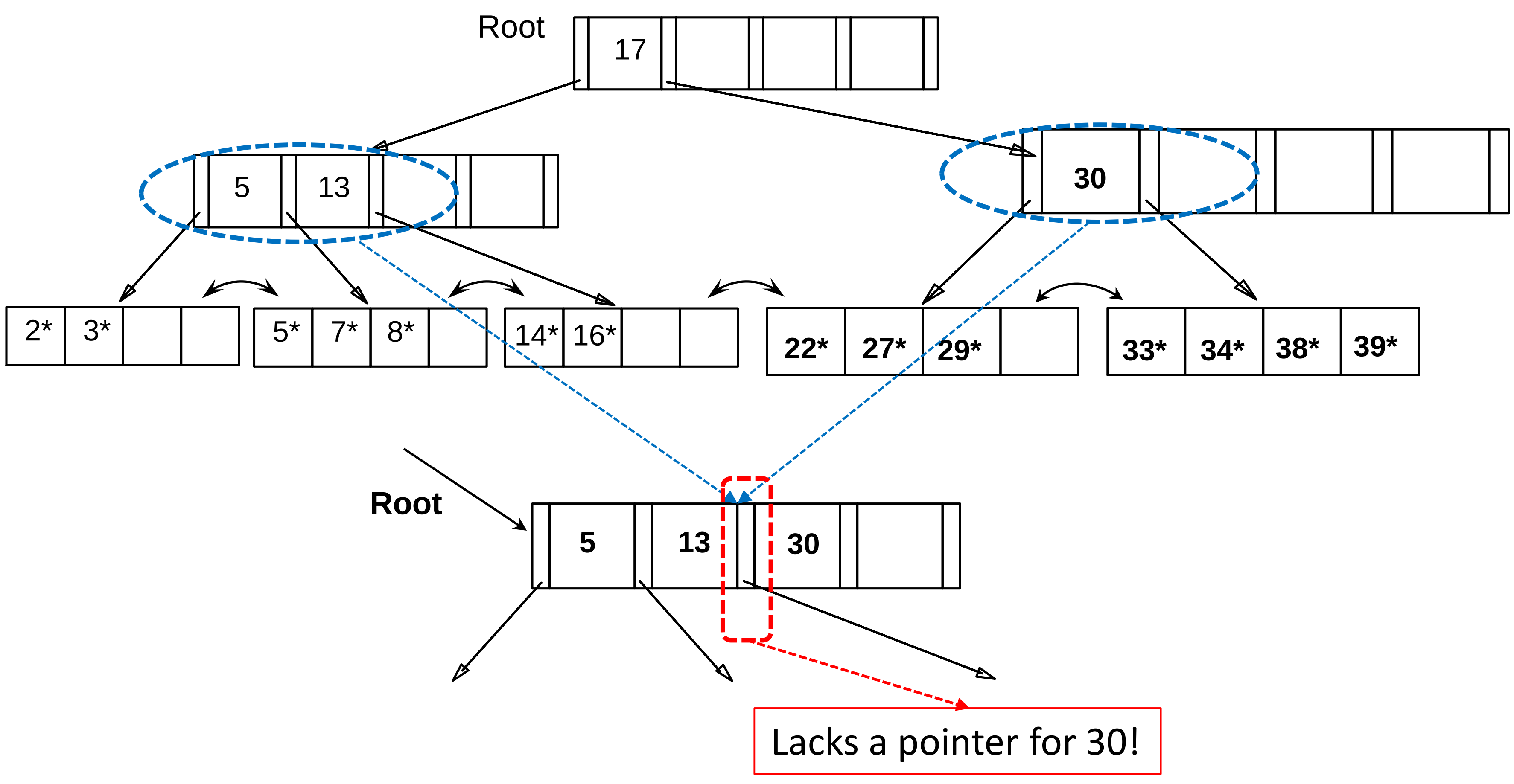
- Delete 24*

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



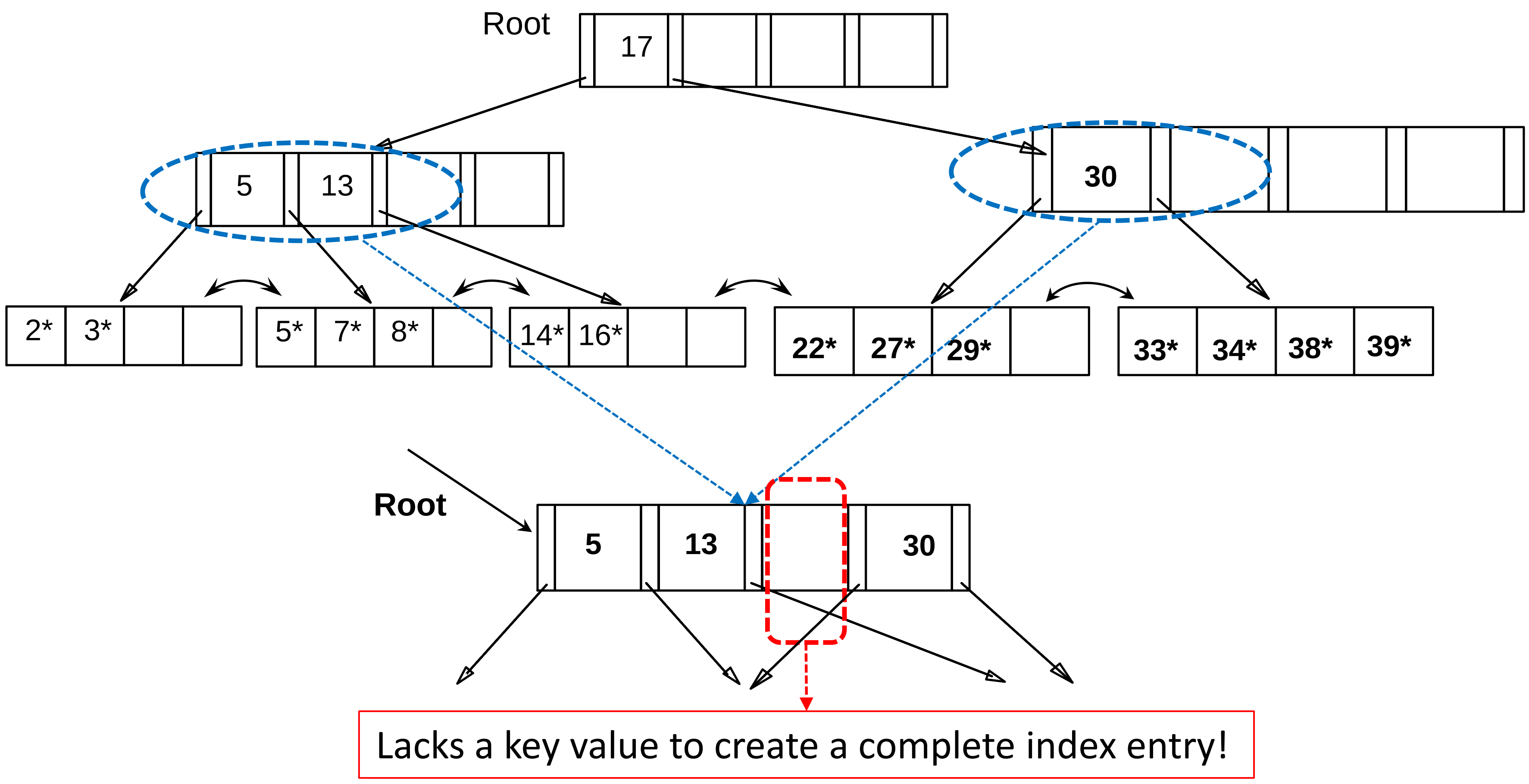
B+ Tree: Examples of Deletions

- Delete 24*



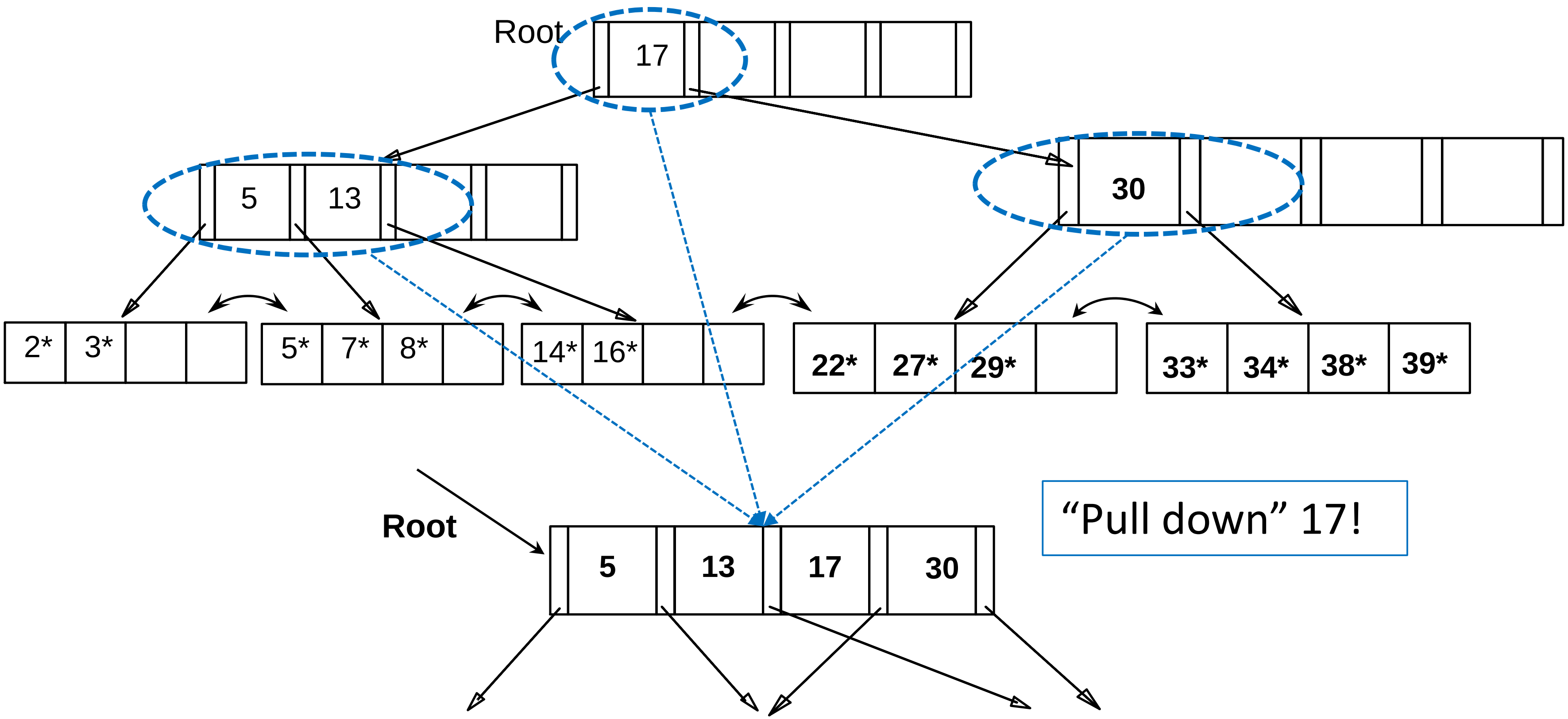
B+ Tree: Examples of Deletions

- Delete 24*



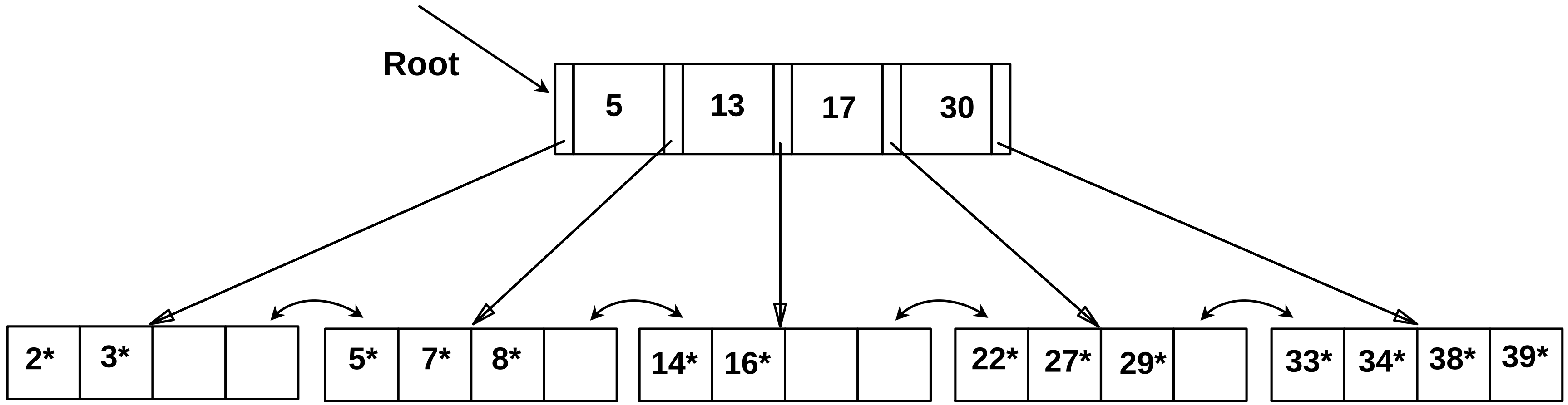
B+ Tree: Examples of Deletions

- Delete 24*



B+ Tree: Examples of Deletions

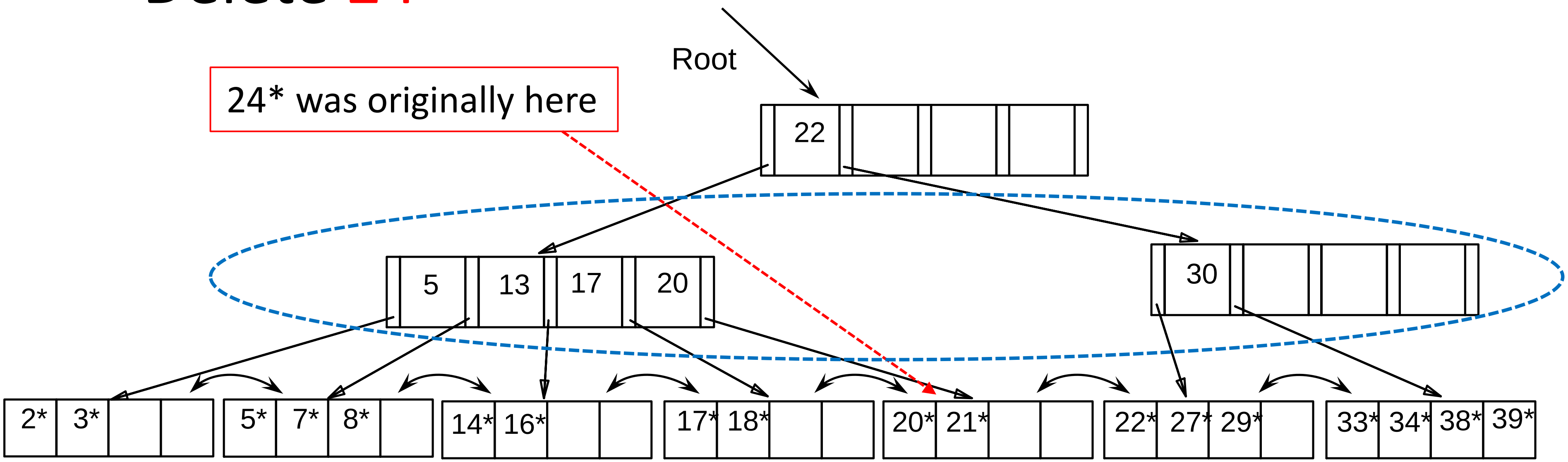
- Delete 24*



FINAL TREE!

B+ Tree: Examples of Deletions

- Delete 24*

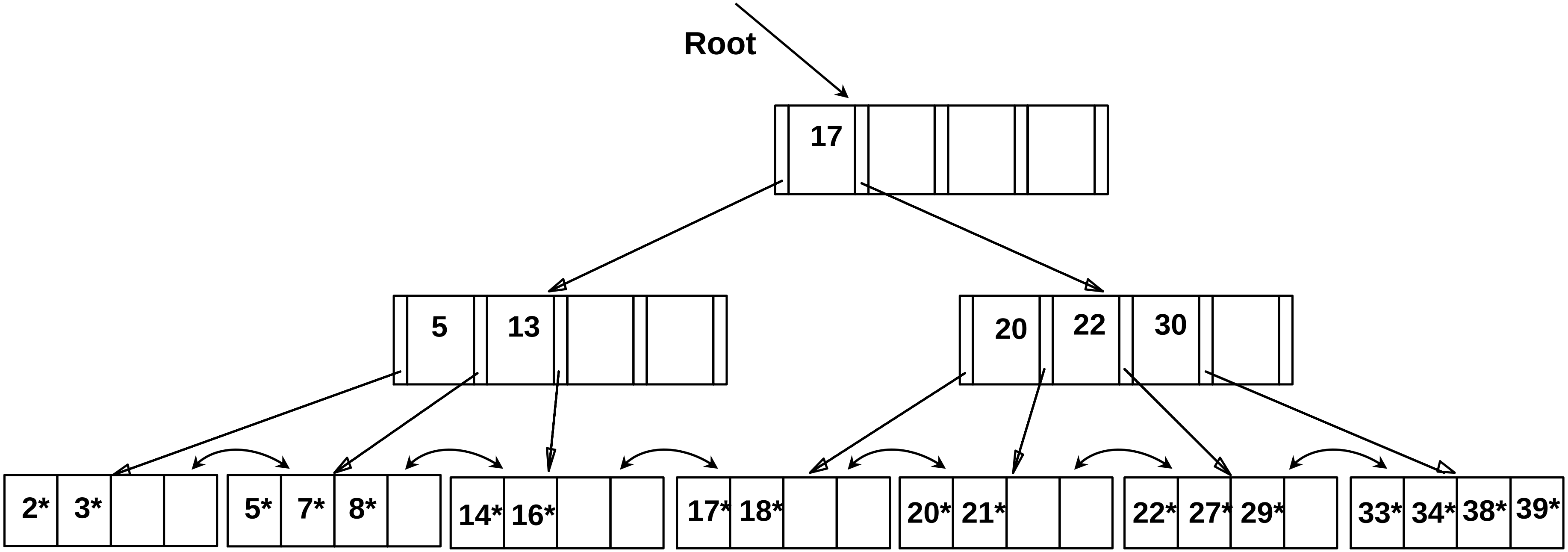


Assume (instead) the above tree *during* deleting 24*

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.