Department of Computer Science
University of Cyprus

EPL646 – Advanced Topics in Databases

Lecture 4

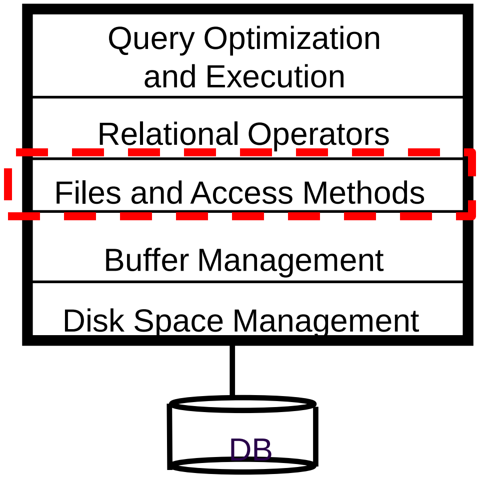
Indexing II: Tree-Structured Indexing and ISAM Indexes

Chap. 10.1-10.8: Ramakrishnan & Gehrke

Demetris Zeinalipour
http://www.cs.ucy.ac.cy/~dzeina/courses/epl446

Lecture Outline
Tree-Structured Indexing

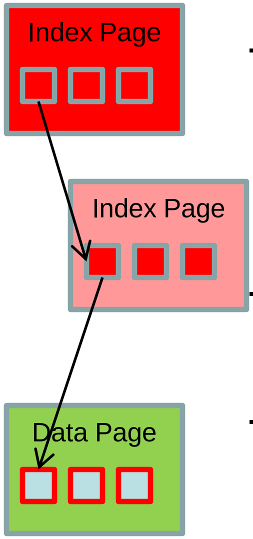
- **Note:** In prior lectures we gave an overview of **Storage and Indexing**. In this and the following lecture we will explore **Indexing** in more detail.
- 10.1) Introduction to Tree Indexes
- 10.2) The ISAM Index
  - Structure of Nodes in Trees,
  - Binary Search over Sorted Files,
  - Binary vs. N-ary Search Trees,
  - ISAM: Indexed Sequential Access Method (Outline, Search, Insert, Delete, Examples)



Indexes (Access Methods)
(Ευρετήρια Δευτερεύουσας Μνήμης)

- An *index* is a data structure that has **index records** that **point to** certain **data records**.
- An index can **optimize** certain kinds of retrieval operations (depending on the index).

- **Definitions**



  - **Index Page (Σελίδες Ευρετηρίου) vs. Data Pages (Σελίδες Δεδομένων):** Index Pages store index records to data records. Both reside on disk because we might have many of these pages!
  - **Data Record (Εγγραφή Δεδομένων):** Stores the actual data e.g., (59,Mike,3.14) .
  - **Index Record (Εγγραφή Ευρετηρίου):** Stores the RID of another index record (then called **index entry**) or a data record (then called **data entry**)

Data Entry **k*** Examples (Παραδείγματα Καταχώρησης **k***)

- **Alternative 1: <k>**

Results in a Index File Organization!

59, Mike, 3.14 — Index Data Entry

- **Alternative 2: <k, RID>**

59, RID#10 — Index Data Entry

59	Mike	3.14

RID#10 — Data Record

- **Alternative 3: <k, [RID,…,RID]>**

59, RID#10, RID#61, #RID82 — Index Data Entry

59	Mike	3.14

RID#10

59	Chris	33.14

RID#61

59	Jim	53.14

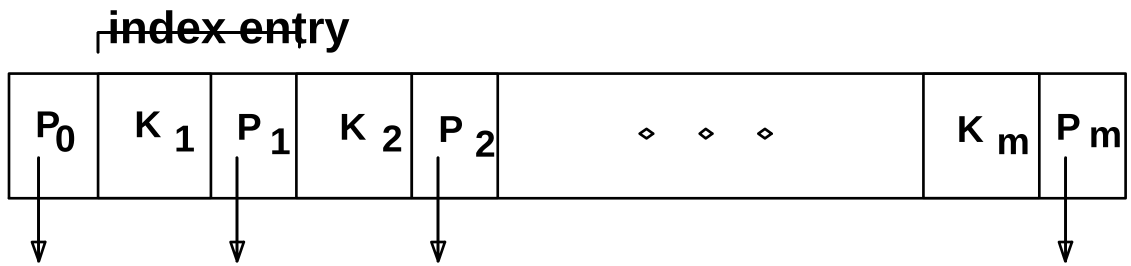
RID#82 — Data Record

Introduction to Tree Structures (Εισαγωγή σε Δενδρικές Δομές)

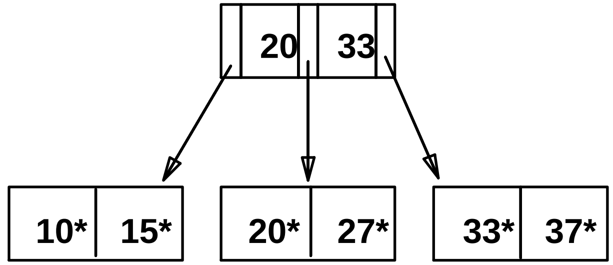
- **We will study two Tree-based structures:**
  - ***ISAM*:** A **static** structure (does not **grow** or **shrink)**.
    - Suitable for situations where the target relation does **not change frequently**;
    - Copes better with **Locking Protocols (explained later),** because the **index/data entries** are statically allocated, thus are not required to be locked during **concurrent access.**
  - ***B+ tree*:** A **dynamic** data structure that adjusts efficiently under **inserts** and **deletes**.
    - Most widely used tree structure in DBMS systems because it copes **efficiently with updates**! and because the cost for range and equality searches is good.
    - **Will be covered subsequently in this lecture!**

Structure of Nodes in Trees
(Δομή Κόμβου σε Δένδρα)

- Same Structure for **ISAM** and **B+Trees** (we shall utilize Alt.1 with keyonly unless otherwise noted)
- **M Keys** and **M+1 Pointers** to **children** (either index entries or data entries)

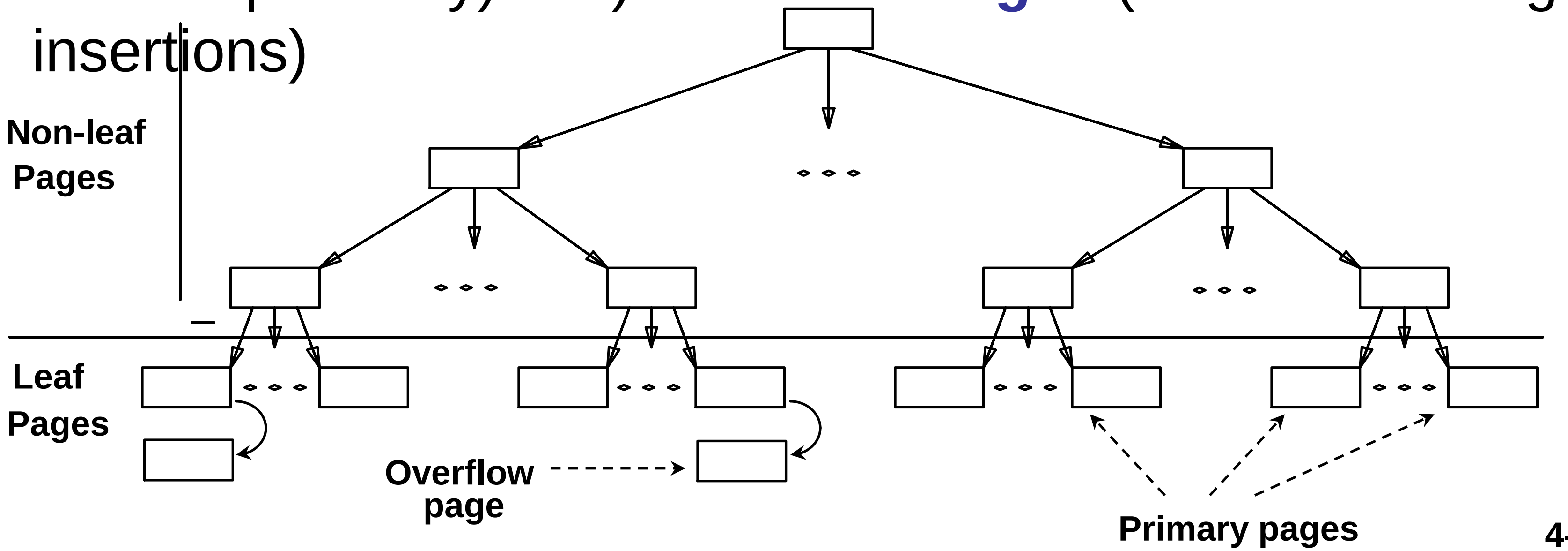


- Example with M=2



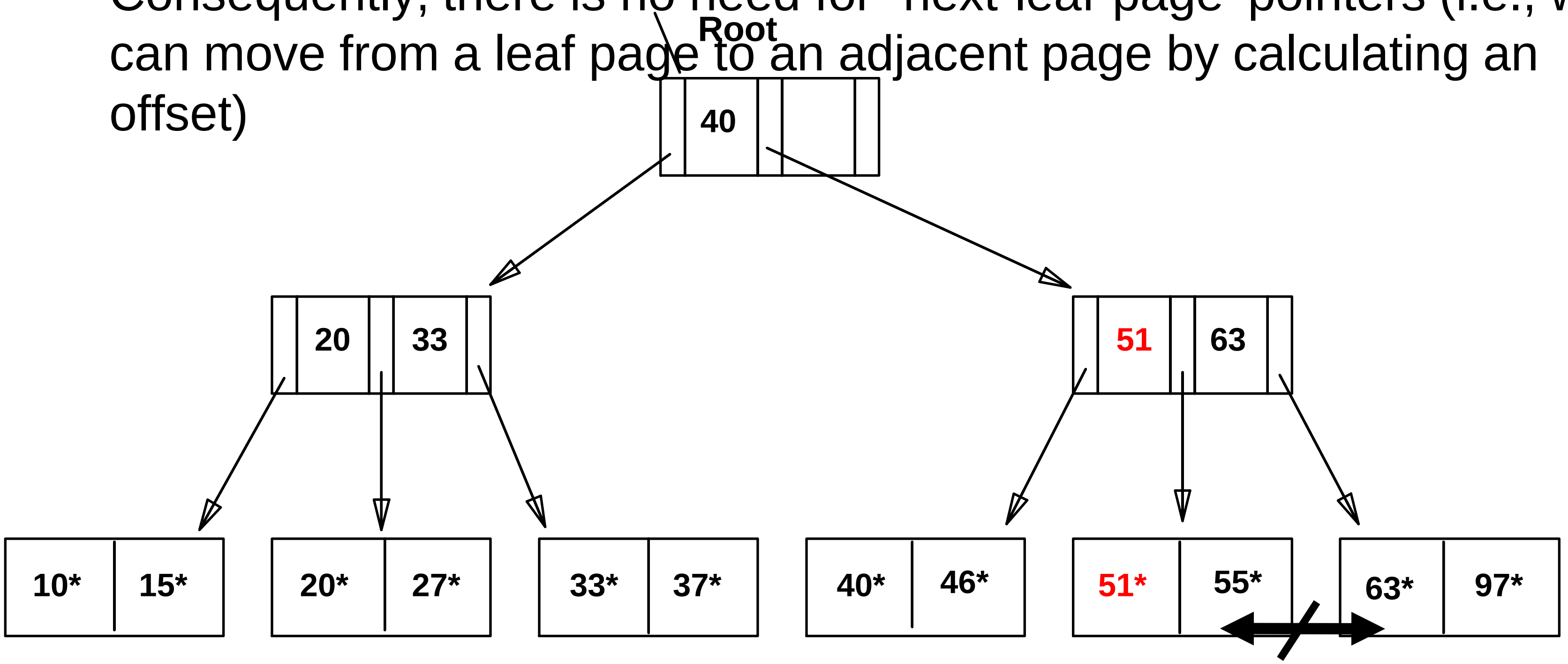
ISAM: Indexed Sequential Access Method

- A simple tree structure utilized by DBMS systems
- Constructed **Statically** at index creation time.
- Consists of **Non-leaf** (**index entries**, allocated at creation time) and **Leaf pages** (**data entries**) – **Alternative 1.**
- **Data Entries** : i) **Primary Pages** (allocated at creation time sequentially) or ii) **Overflow Pages** (allocated during insertions)



Outline of Operation (Ανασκόπηση Λειτουργίας)

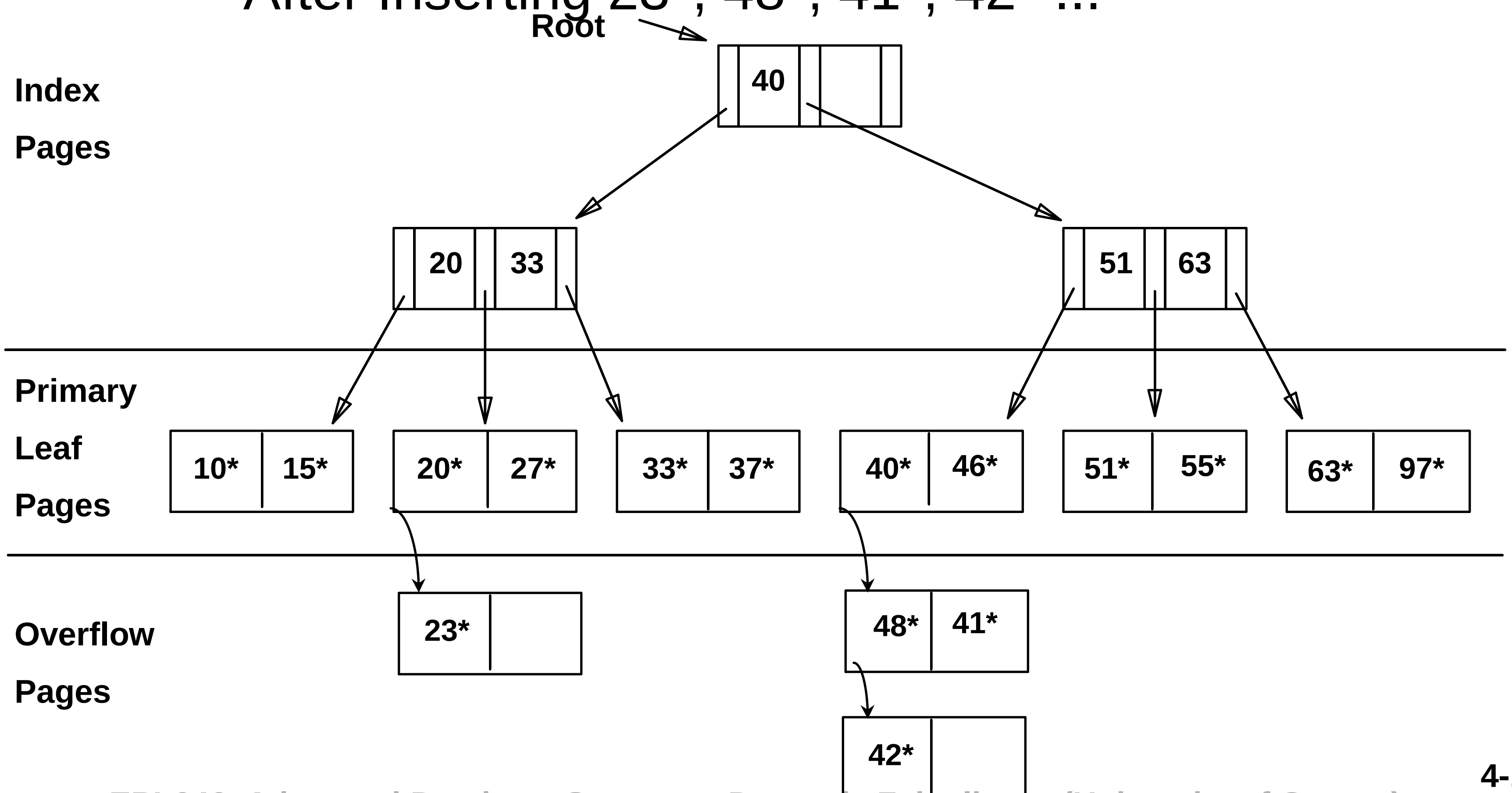
- **Search:** Start at root; use key comparisons to go to leaf. Cost: $\lfloor \mathbf{log}_F \mathbf{N} \rfloor$; F=#entries_per_indexPage+1, N=#leafpgs
- Recall that data Entries are allocated sequentially when the tree is created.
  - Consequently, there is no need for `next-leaf-page' pointers (i.e., we can move from a leaf page to an adjacent page by calculating an offset)



Inserting to an ISAM Index
(Εισαγωγές στο Ευρετήριο ISAM)

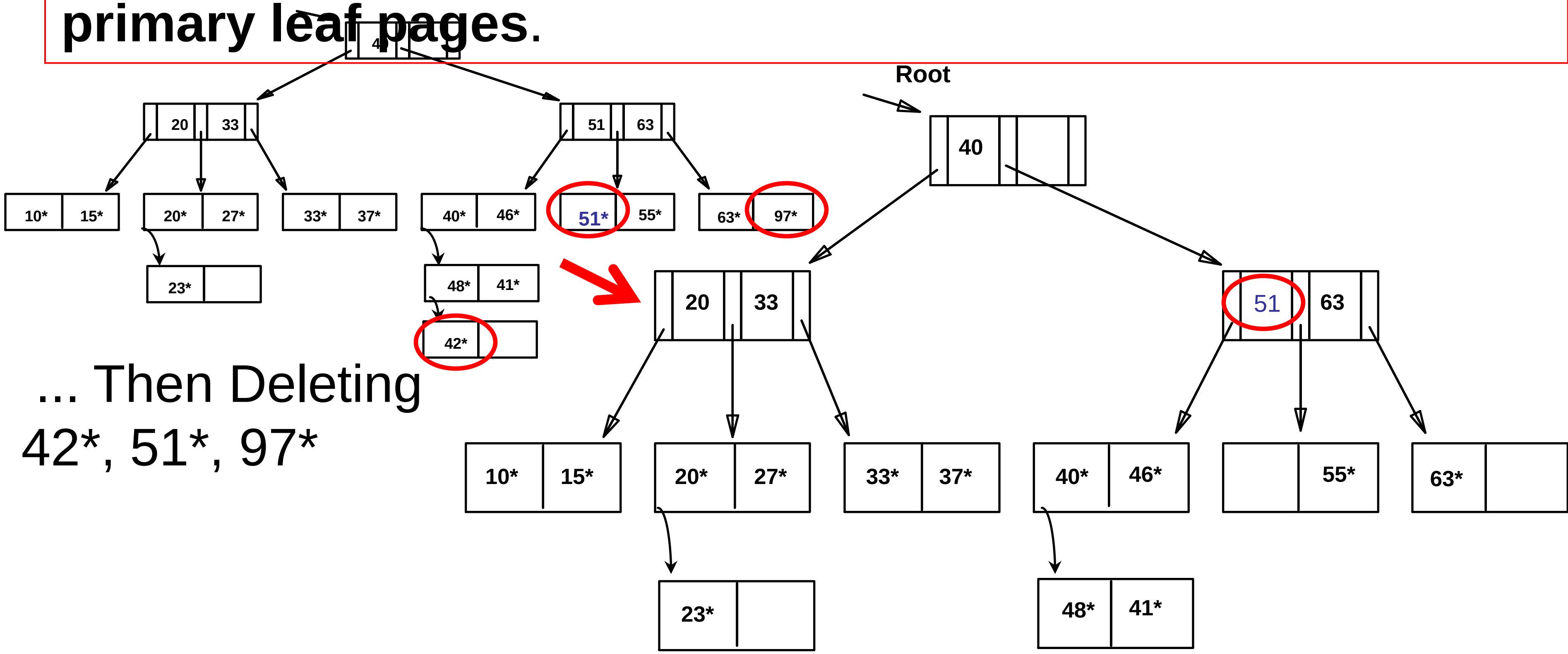
Insert: Find the appropriate **leaf data entry** and assign it to there. If full, allocate an overflow page and put it there

After Inserting 23*, 48*, 41*, 42* ...



Deletions from an ISAM Index
(Διαγραφές από το Ευρετήριο ISAM)

Delete: Find and remove from leaf; if **overflow page gets empty** then de-allocate then given page. Never deallocate **primary leaf pages**.



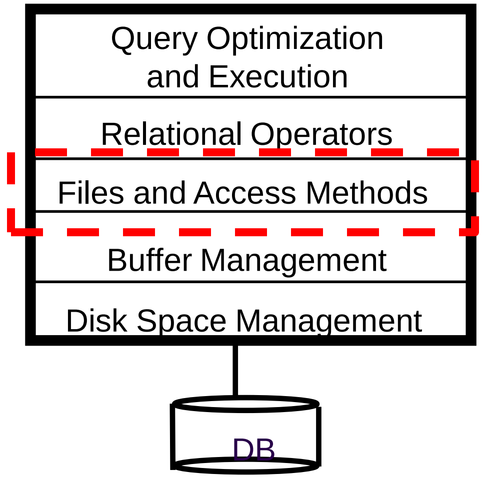
... Then Deleting 42*, 51*, 97*

☒ Note that 51* appears in index levels, but not in leaf! Static tree structure: inserts/deletes affect only leaf pages! …Will be useful for concurrency control (locking protocol)

Lecture Outline

B+ Trees: Structure and Functions

- 10.3) Introduction to B+ Trees
- 10.4-10.6) B+Tree Functions: Search / Insert / Delete with Examples
- 10.7) B+ Trees in Practice.
  - Prefix-Key Compression (Προθεματική Συμπίεση Κλειδιών)
  - Bulk Loading B+Trees (Μαζική Εισαγωγή Δεδομένων)

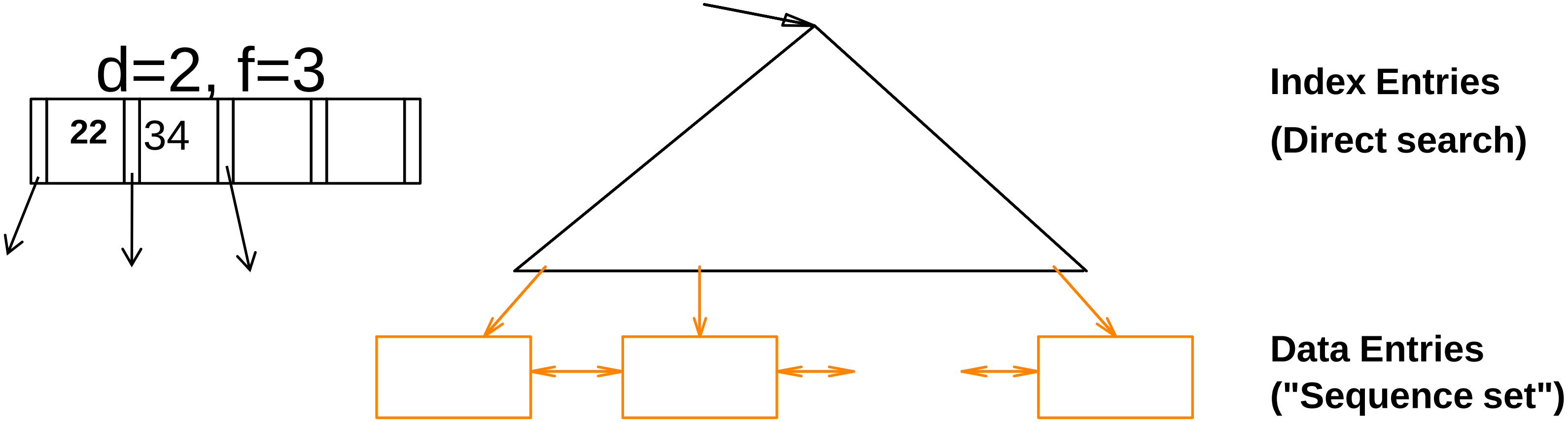


Introduction to Tree Structures
(Εισαγωγή σε Δενδρικές Δομές)

- **We will study two Tree-based structures:**
  - ***ISAM*:** A **static** structure (does not **grow** or **shrink)**.
    - **Suitable when changes are infrequently;**
    - Copes better with **Locking Protocols**
  - ***B+ tree*:** A **dynamic** data structure which adjusts efficiently under **inserts** and **deletes**.
    - Most widely used tree structure in DBMS systems!
    - Has similarly to ISAM, nodes with a high **fan-out (f)** (~133 children per node).
    - Similar to a **Btree** but different…
      - In a B+Tree, **data entries** are stored at the leaf level.
      - **A Btree allows search-key values to appear only once**; eliminates redundant storage of search keys (not suitable for DB apps where more index entries yield better search performance)

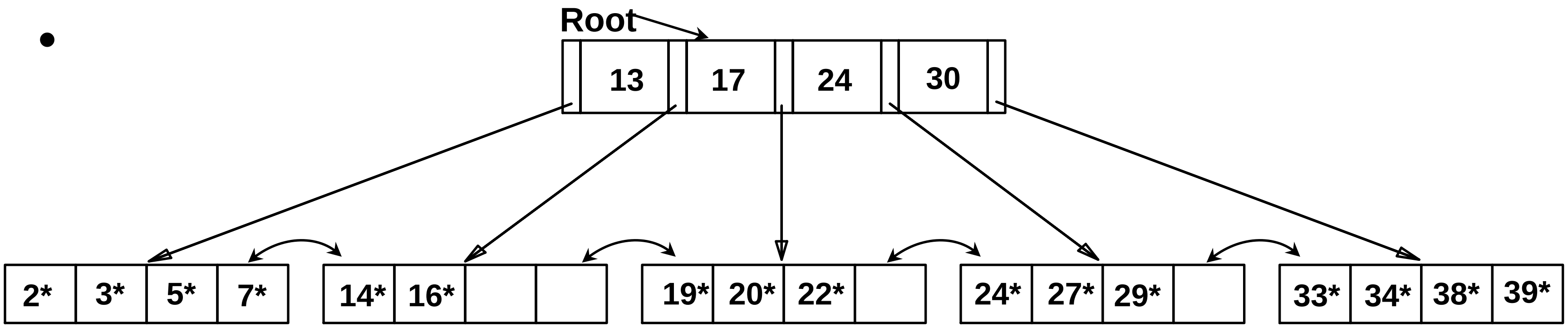
B+ Tree: Introductory Notes
(B+Tree: Εισαγωγικές Επισημάνσεις)

- Insert/delete at **log $_F$ N** cost; keep tree ***balanced (ισοζυγισμένο).*** (**F** = fanout, **N** = # leaf pages)
- **Minimum 50% occupancy** (except for root). Each node contains **d** <= *m* <= 2**d** entries. The parameter **d** is called the ***order* of the tree (βαθμός του δένδρου)**.
- Supports **equality** and **range-searches** (αναζητήσεις ισότητας και διαστήματος) efficiently.



Example B+ Tree
(Παράδειγμα B+Tree)

- Search begins at root, and key comparisons direct it to a leaf (as in ISAM).
- Search for 5*, 15*, all data entries >= 24* ...
-

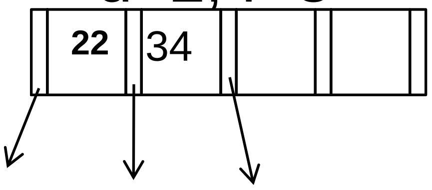


- Based on the search for 15*, we know its not in the tree!
- Note that **leaf pages** (τερματικοί κόμβοι) **are** linked together in a **doubly-linked list** (as opposed to **ISAM).**
- That happens because ISAM nodes are allocated **sequentially** during **Index construction time**
  - consequently, no need to maintain the next prev-next-pointer.

B+ Trees in Practice
(B+Trees στην Πράξη)

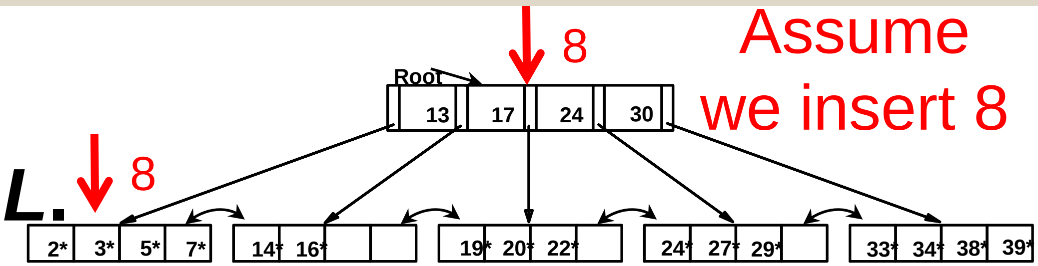
- **Typical order (d):** 100 (ie100<=#children<=**200**)
- **Typical fanout (f)** = 133
  - **Typical fill-factor: 67% (133/200)**

d=2, f=3

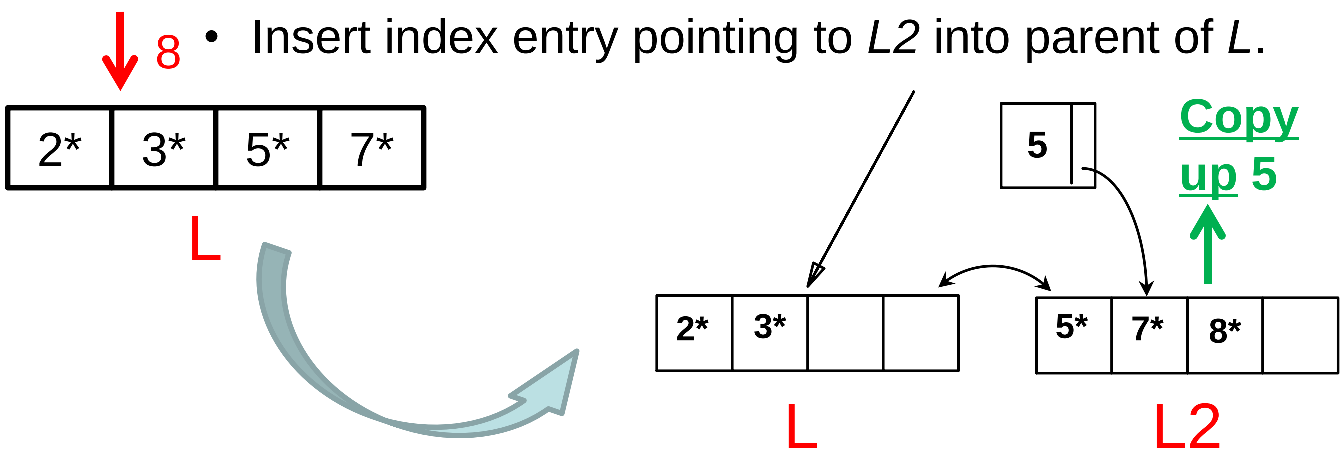


- **Typical capacities:**
  - Height 4: 133^4 = 312,900,700 records
  - Height 3: 133^3 = 2,352,637 records
- **Can often hold top levels in buffer pool:**
  - Level 1 = 133^0 = **1 page = 8 Kbytes**
  - Level 2 = 133^1 = 133 pages = ~1 MB (1064 KB)
  - Level 3 = 133^2 = 17,689 pages = ~133 MB (141,512KB)

B+ Tree Insertion Algorithm (Αλγόριθμος Εισαγωγής στο B+Tree)



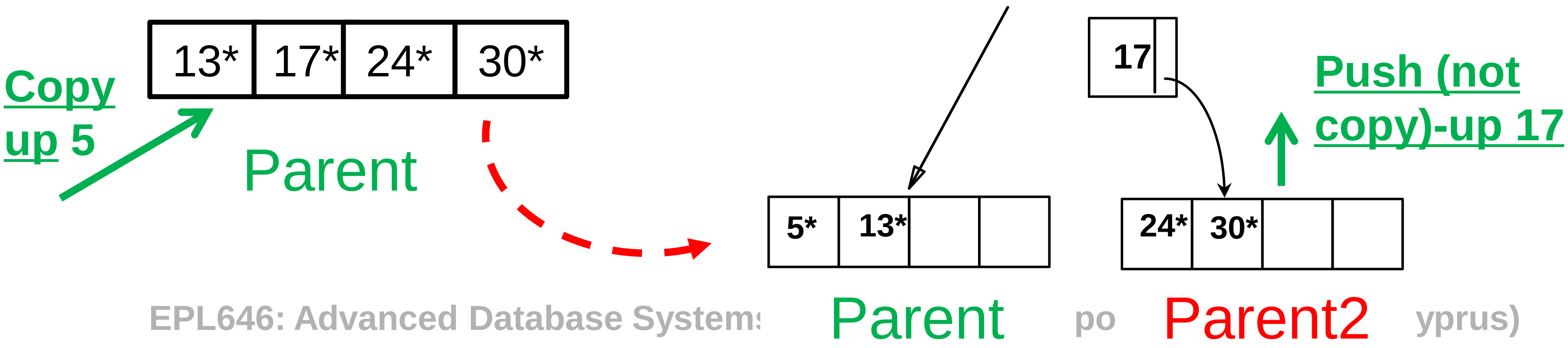
1. **Find** correct **leaf *L.***
2. **Put** data entry **onto *L*.**
   - If *L* has enough space, *done*!
   - Else <u>*split (διαμοίραση)*</u> ***L** (into **L** and a new node **L2**)*
     - Redistribute (Ανακατένειμε) entries evenly between L and L2, <u>**copy up (Αντιγραφή-Πρός-Τα-Πάνω)**</u> middle key.
     - Insert index entry pointing to *L2* into parent of *L*.



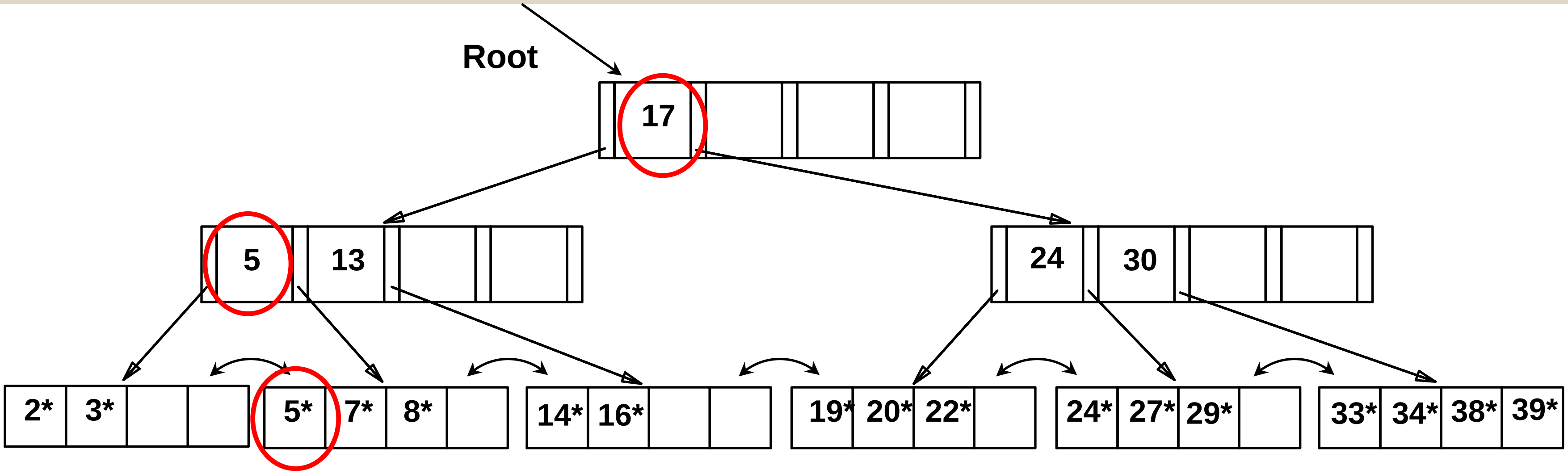
- <u>**Copy up**</u> **5:** cannot just <u>push-up</u> 5 as every data entry needs to appear in a leaf node
- **Problem:** 5 won't fit in parent of L2. (see next slide)

B+ Tree Insertion Algorithm (Αλγόριθμος Εισαγωγής στο B+Tree)

3. A parent needs to recursively **Push-Up (Προώθηση-Προς-Πάνω)** the **middle key** until the insertion is successful i.e.,
   - No need to **copy-up** as the latter will generate redundant index entries.
   - If ***Parent*** has enough space, *done*!
   - Else *split (διαμοίραση)* *Parent*
     - Redistribute (Ανακατένειμε) entries evenly, **push up** middle key.
4. Splits "grow" tree; root split increases **height** (ύψος)
   - Tree growth: gets *wider* or *one level taller at top.*



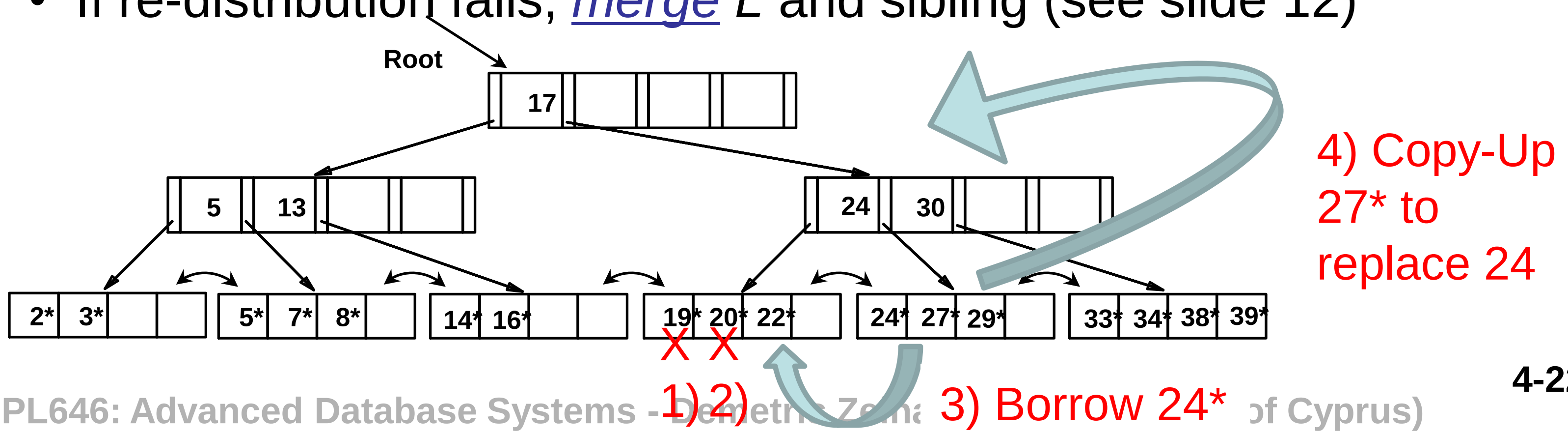
Example B+ Tree After Inserting 8* Αποτέλεσμα Εισαγωγής 8*



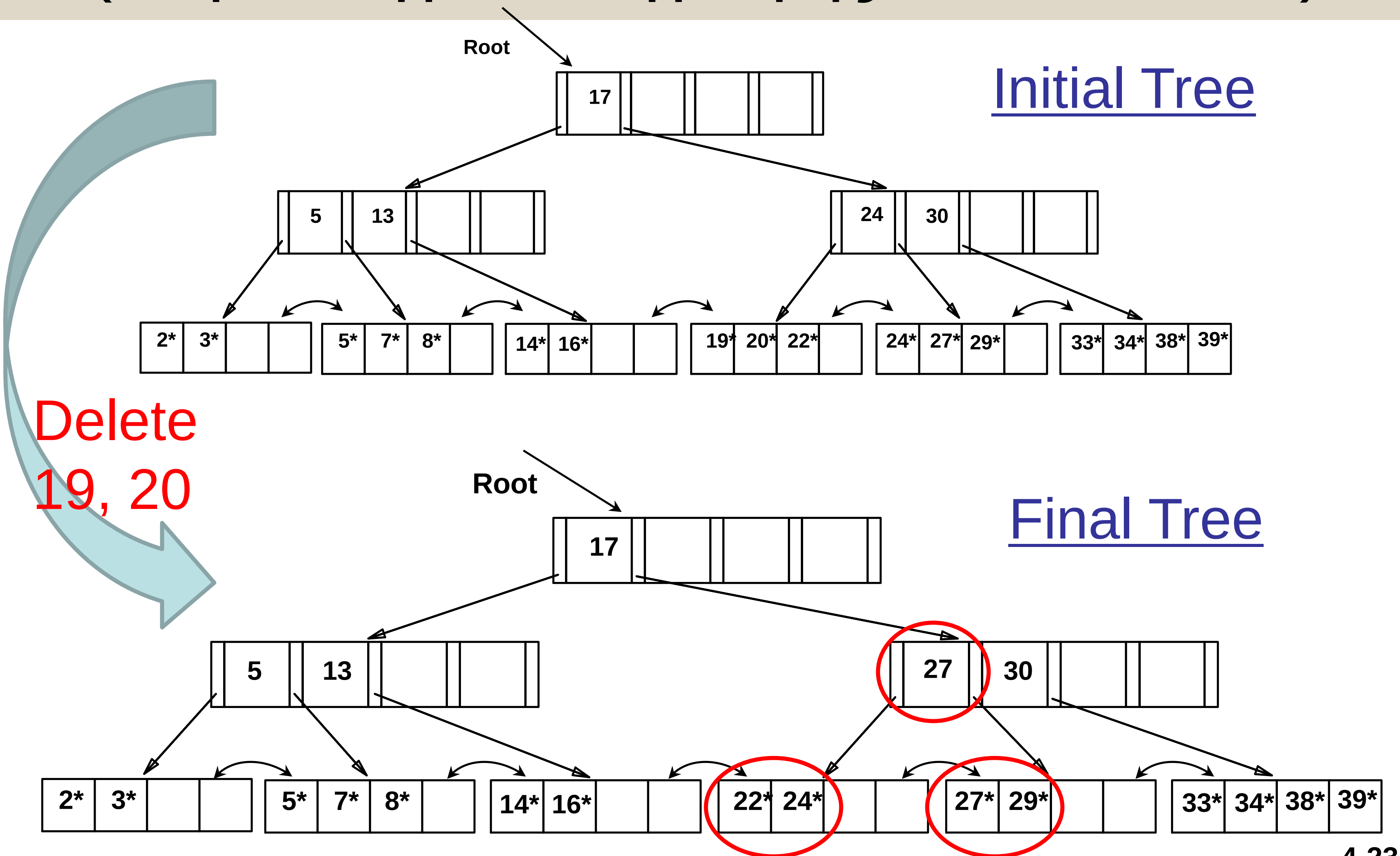
- **Root was split =>** That lead to increase in height from 1 to 2.
- **Minimum occupancy (d, i.e., 50%)** is guaranteed in both leaf and index pages splits (for root page this constraint is relaxed)
  - **Split occurs** when adding 1 key to a node that is full (has **2d entries**). Thus we will end up with two nodes, one with **d** and one with **d+1** entries.
- Can avoid split by **re-distributing entries between siblings** – (αδελφικοί κόμβοι); however, this is usually not done in practice. The borrowing practice is adopted only during deletions (see next).

B+ Tree Deletion Algorithm (Αλγόριθμος Διαγραφής απο B+Tree)

- Start at root, **find leaf *L*** where entry belongs.
  - E.g., deleting 19 then 20
- **Remove the entry K* (not respective index entries)**.
  - If L is **at least half-full**, *done! (e.g., after deleting 19*)*
  - If L has only **d-1** entries, (e.g., after deleting 20*)
    - Try to re-distribute, borrowing from <u>*sibling*</u> *(adjacent node with same parent as L)*. (e.g., borrow 24* and update)
    - If re-distribution fails, <u>*merge*</u> *L* and sibling (see slide 12)

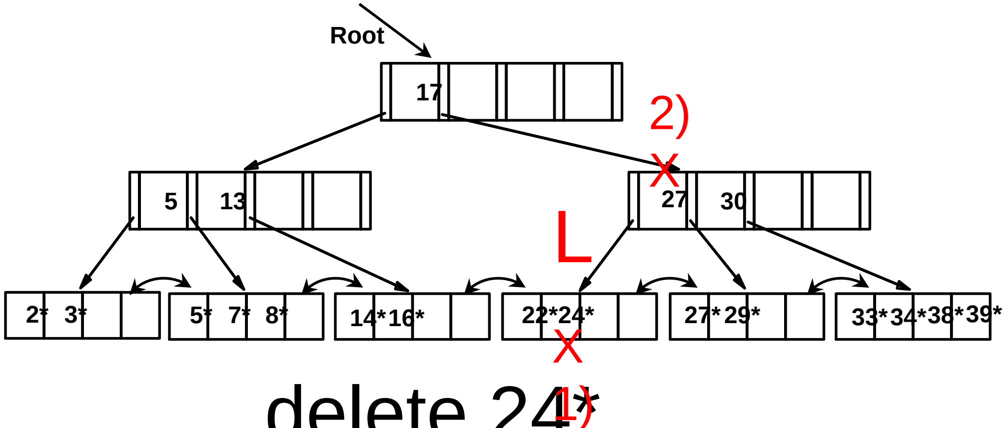


B+ Tree Deletion Example (Παράδειγμα Διαγραφής από B+Tree)

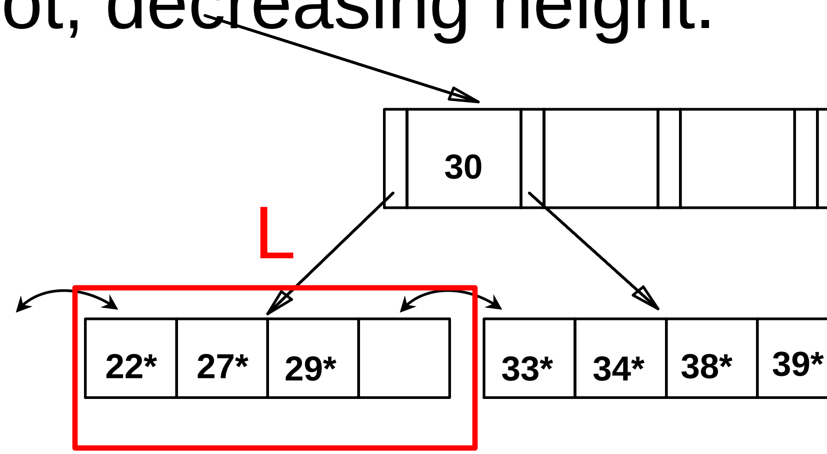


B+ Tree Deletion Algorithm (Αλγόριθμος Διαγραφή απο B+Tree)

- If re-distribution after delete fails then *merge* *L* **and sibling** (e.g., **delete 24 =>** can't borrow => merge)
- Now we also need to adjust **parent of *L*** (pointing to *L* or sibling). **(i.e., delete 27)**
- Merge could propagate to root, decreasing height.



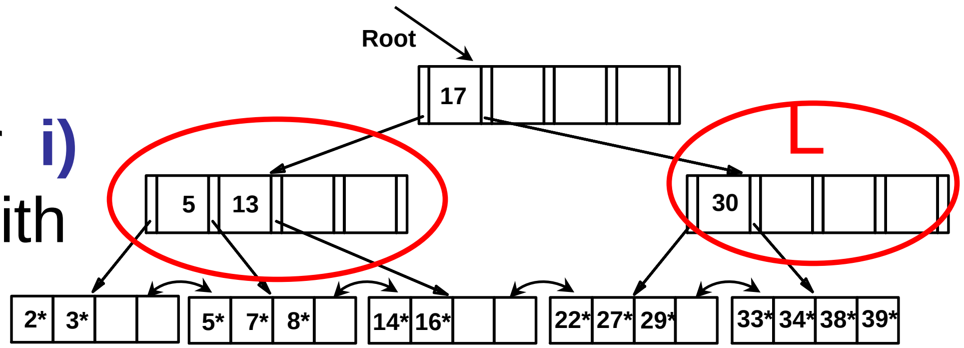
delete 24*

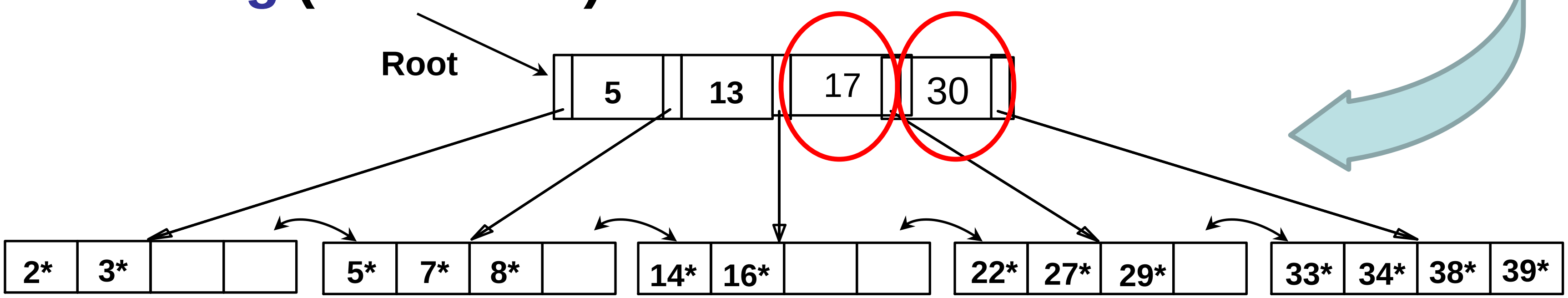


Merged {22} with {27,29}

Merging propagates to sink
(Η Συγχώνευση διαδίδεται μέχρι τη ρίζα)

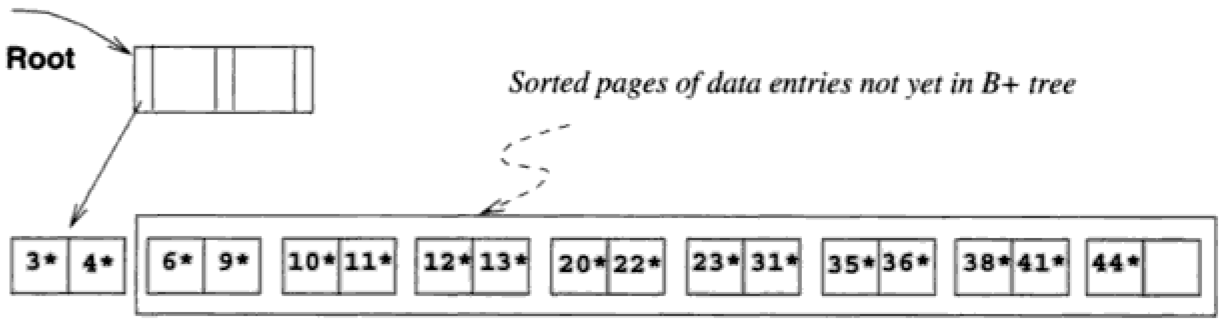
- But … occupancy Factor of **L** dropped below 50% (d=2) which is not acceptable.
- Thus, **L** needs to be either **i) merged (συγχωνευτεί)** with its sibling {5,13}
- or **ii) redistributed (ανακατανεμηθεί) with its sibling (next slide)**



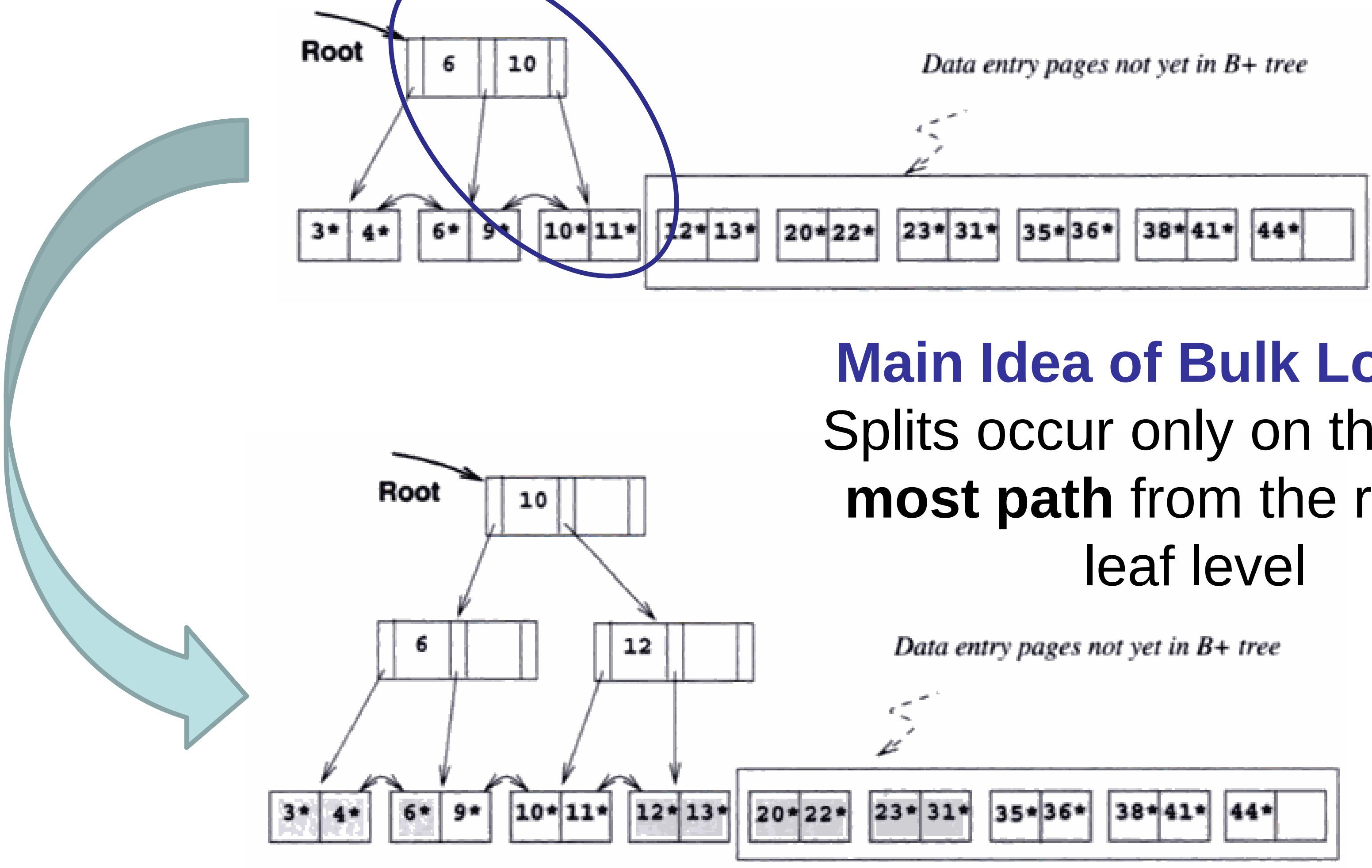


Summary of Bulk Loading (Μαζική Εισαγωγή Δεδομένων)

- **Scenario:** We want to **construct** a B+Tree on a **pre-existing collection (υφιστάμενη συλλογή)** of records
- **Option 1: multiple (individual) inserts.**
  - Slow and does not give sequential storage of leaves.
- **Option 2: *<u>Bulk Loading</u>* *(Μαζική Εισαγωγή)*.**
  - **Idea: Sort** all data entries, insert pointer to **first (leaf)** page **in a new (root)**.
  - **Effect:** Splits occur only on the **right-most path** from the root to leaves.
  - **Advantages: i**) **Fewer I/Os** during build and ii) **Leaves** will be **stored sequentially** (and linked, of course).



Bulk Loading with Example (Μαζική Εισαγωγή με Παράδειγμα)



Main Idea of Bulk Loading:
Splits occur only on the **right-most path** from the root the leaf level