KOÇ UNIVERSITY

Database Management Systems

Tree-Based Indexing

M. Emre Gürsoy

Assistant Professor
Department of Computer Engineering
www.memregursoy.com

Motivation

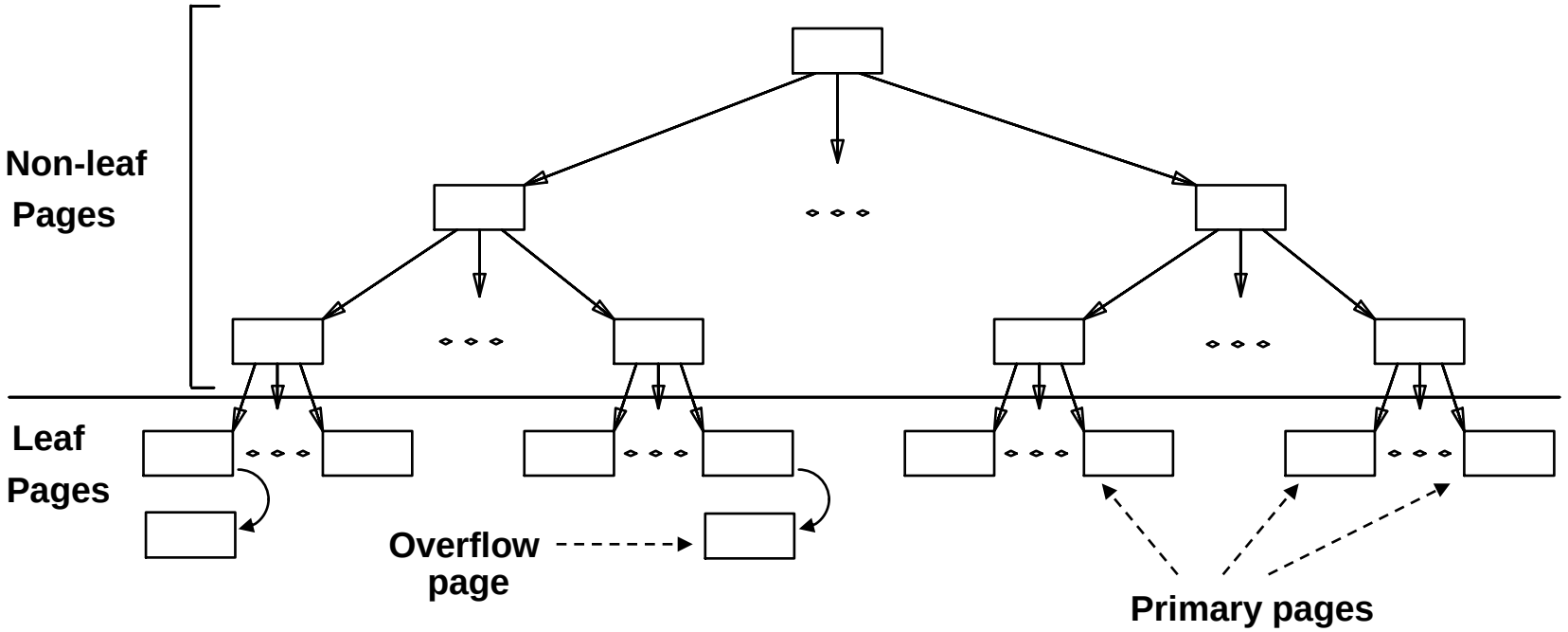
- Range selection (range search):
  - Different from equality selection
  - Common in practice
  - Not supported effectively by hash-based indexing

```
SELECT  *
FROM    Students
WHERE   Gpa > 3.0
```

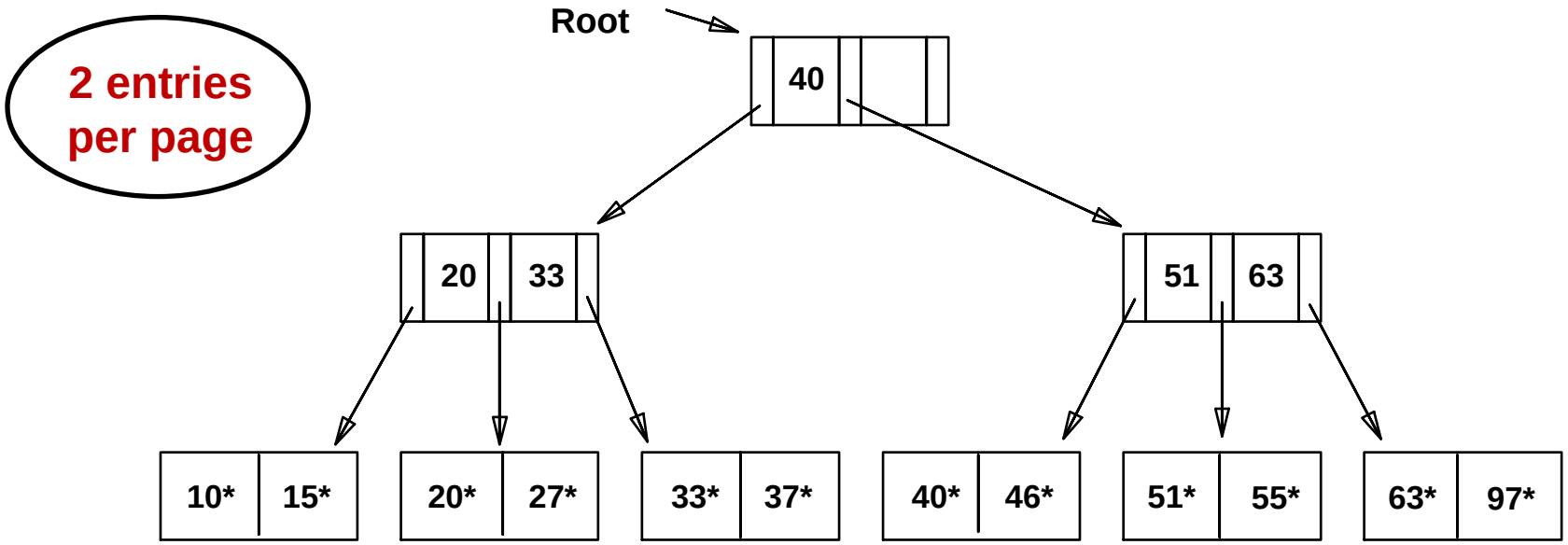
- Tree-based indexing techniques support both range selection and equality selection effectively.
  - Often used for table indexing in DBMS
- We'll cover two types of tree-based index structures:
  - ISAM Trees
    - Static, not preferred
  - B+ Trees
    - Also known as: "the coolest data structure ever"

ISAM Trees

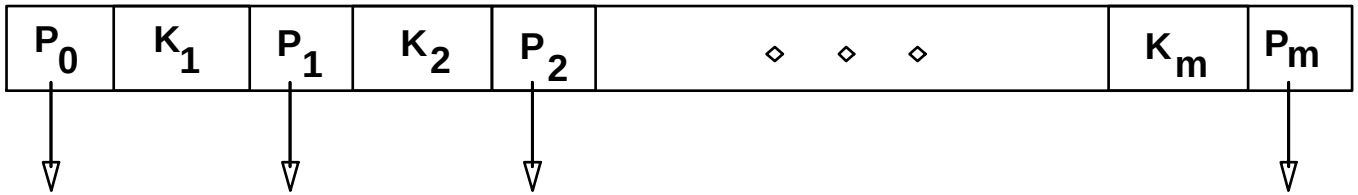
- Tree-based data structure (non-leaf pages are static)
  - Leaf pages store actual data entries
  - Non-leaf pages act as the “guide”



Example ISAM Tree

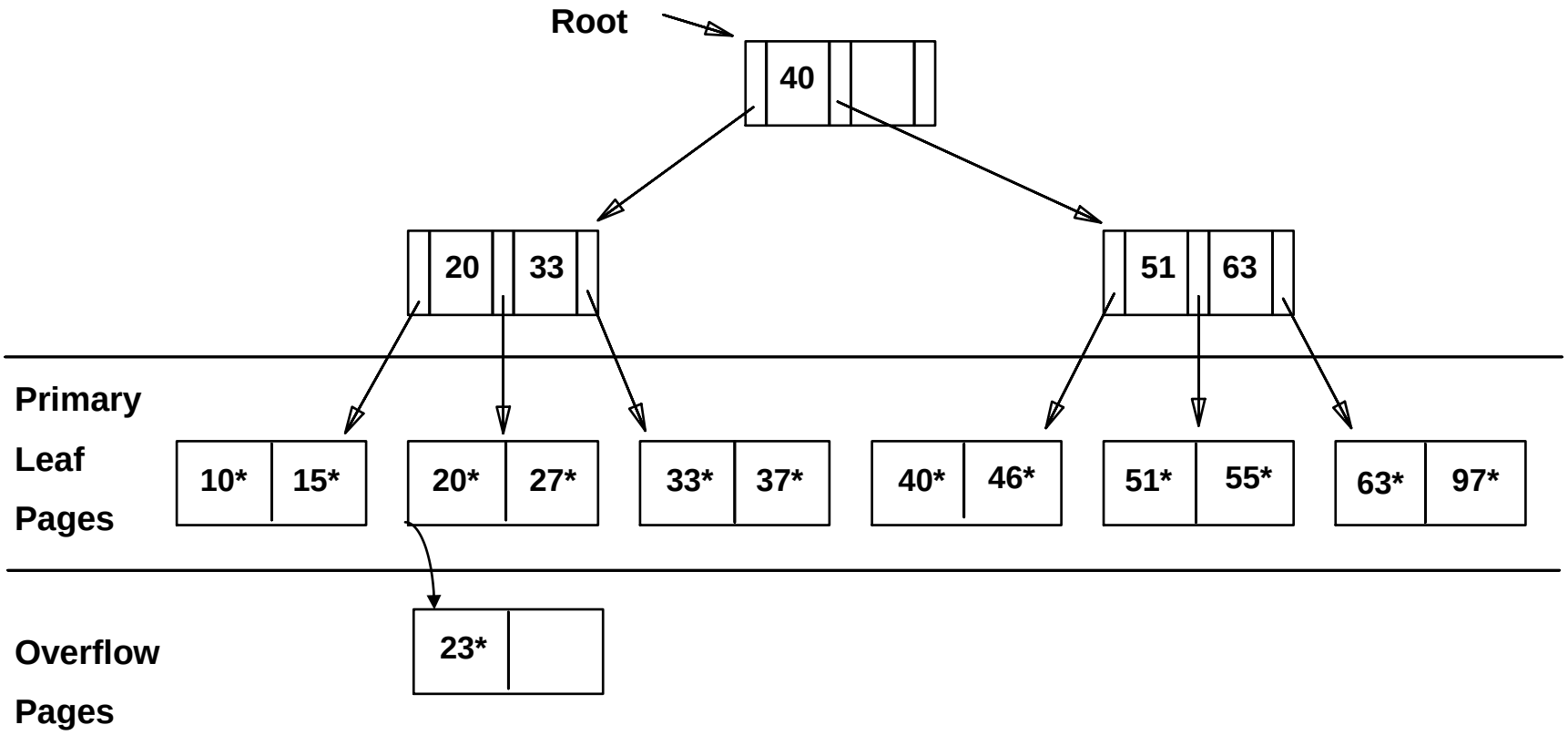


- Structure of leaf pages: only store data entries
- Structure of non-leaf pages: pointers and keys



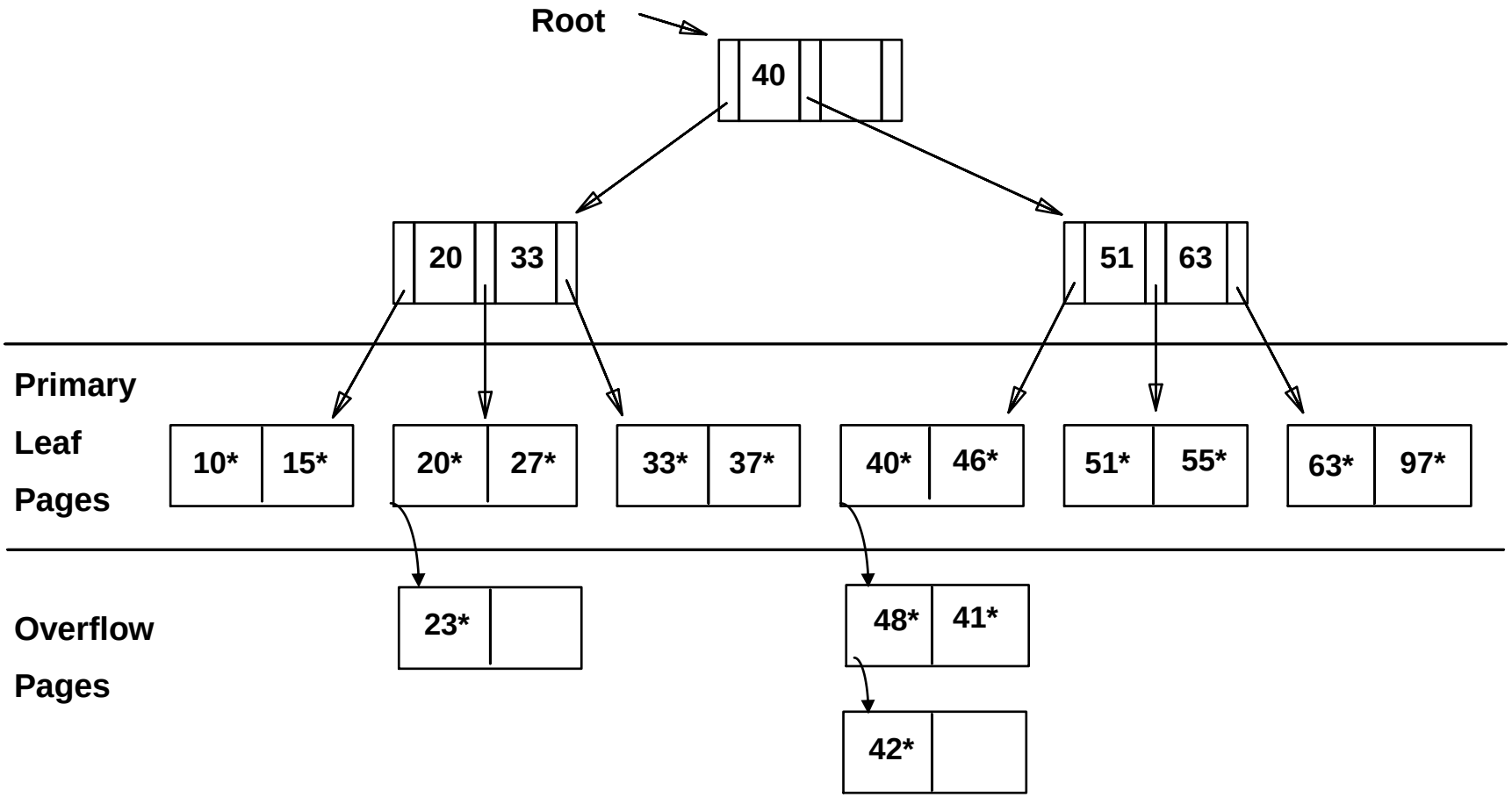
Insertion

- After Inserting 23*:



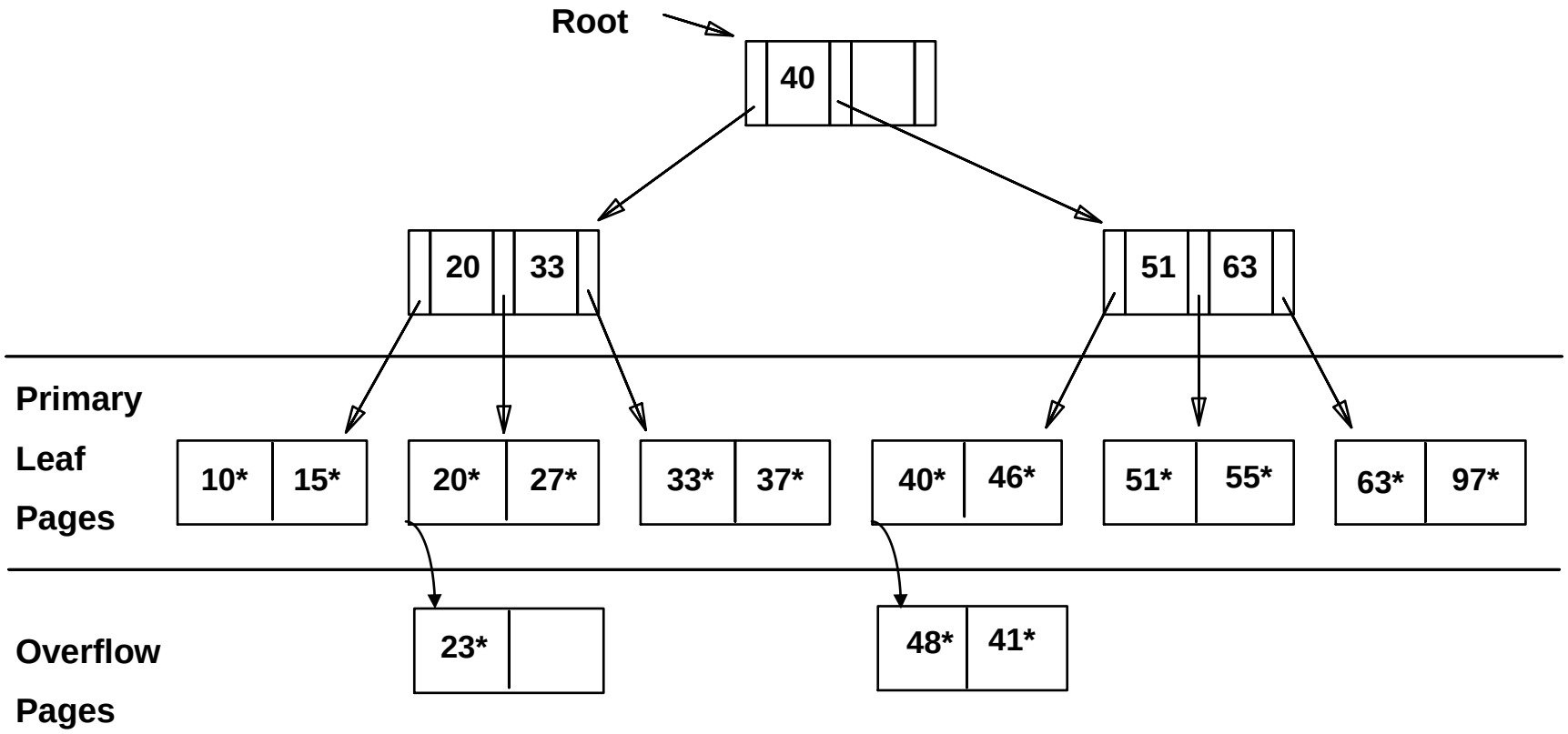
Insertion

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



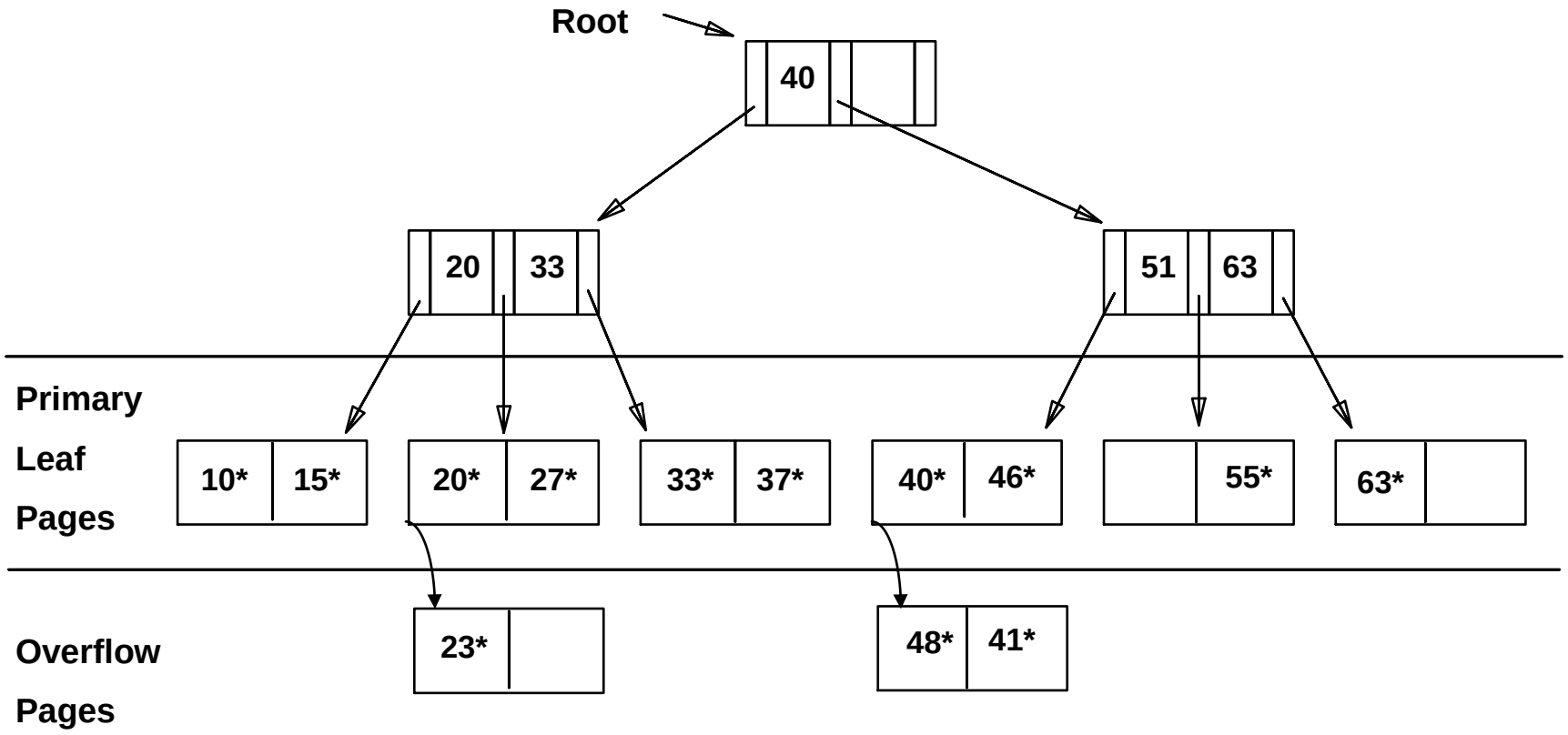
Deletion

- After Deleting 42*:



Deletion

- After Deleting 51*, 97*:



Note that 51 can appear in the non-leaf pages but not in the leaf pages!

ISAM Shortcomings

- ISAM tree is a static data structure
  - Leaf pages and overflow pages may change since they contain actual data
  - Non-leaf pages (constituting the index) **do not change**

- ISAM tree cannot balance itself after inserts/deletes
  - Long overflow pages are possible
  - What if data distribution changes over time?

- **What are the best-case and worst-case complexities of searching for a record in an ISAM tree?**

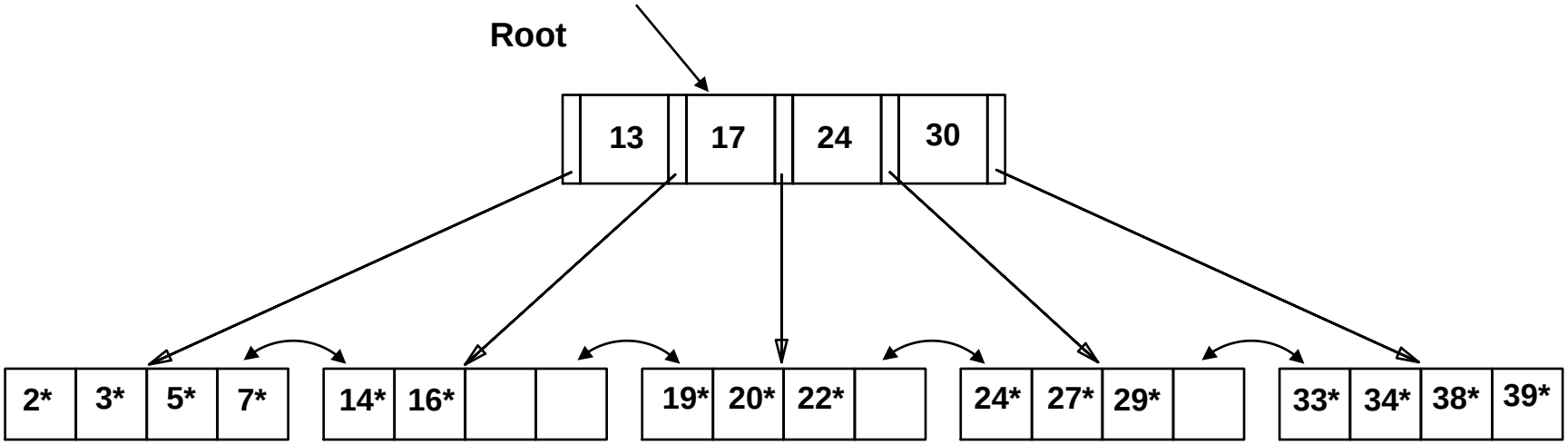
B+ Trees

- A self-balancing tree data structure that keeps data sorted and allows search, sequential access, insertion, and deletion in **O(logN)** time.
  - Belonging to the “family” of B-Trees
    - B-tree, B+ tree, B* tree, B^{link} tree, …
  - **Very commonly** used in DBMS

- Properties:
  - Perfectly height-balanced at all times
  - Every node other than the root must be at least half full
  - Order = m means nodes have at most m children
    - Thus, each node holds at most m-1 data entries

Example B+ Tree

- Search for 5*
- Search for 15*
- Search for all entries ≥ 24 and ≤ 35
  - Use pointers among the leaves



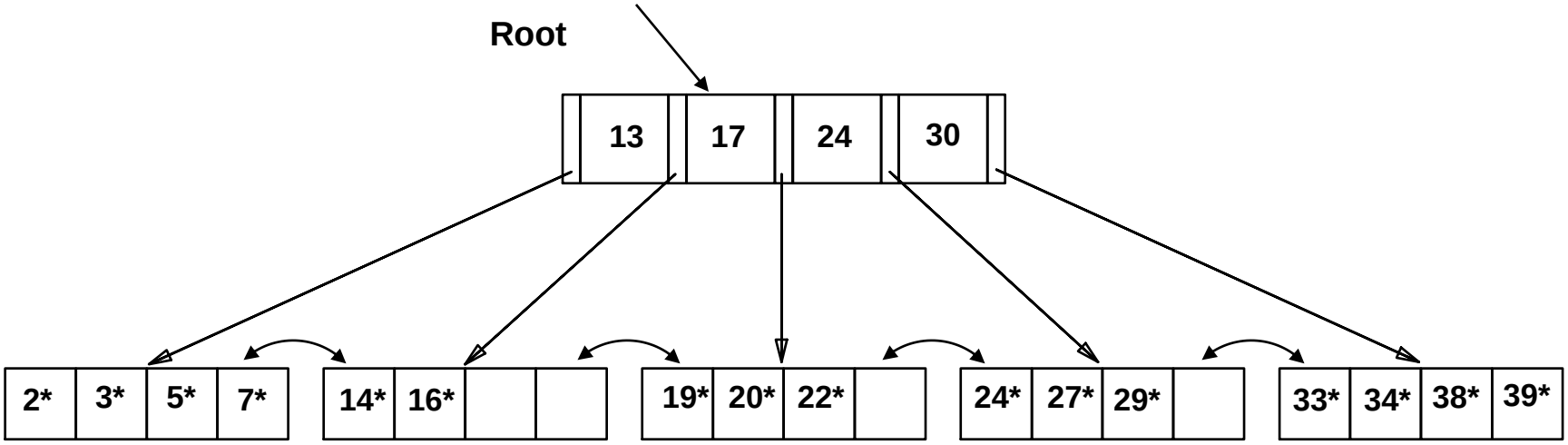
Insertion

- Start at root, find the leaf node *L* to insert to.
- If *L* has enough space:
  - Insert into *L*, ensure the leaf is sorted, done!
- Otherwise:
  - Must split *L* into two nodes
  - Redistribute entries evenly, copy up the middle key
  - This split may cause recursive splits in upper levels

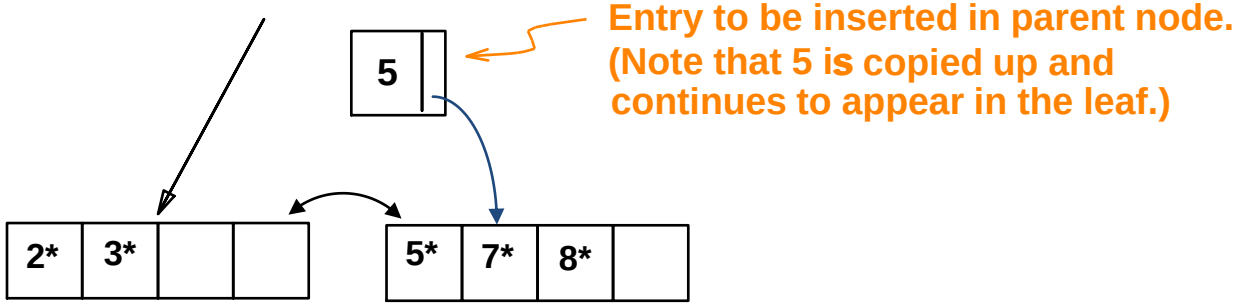
- To handle splits in upper levels:
  - Redistribute entries evenly, push up the middle key
  - Always check if the push up causes further splits

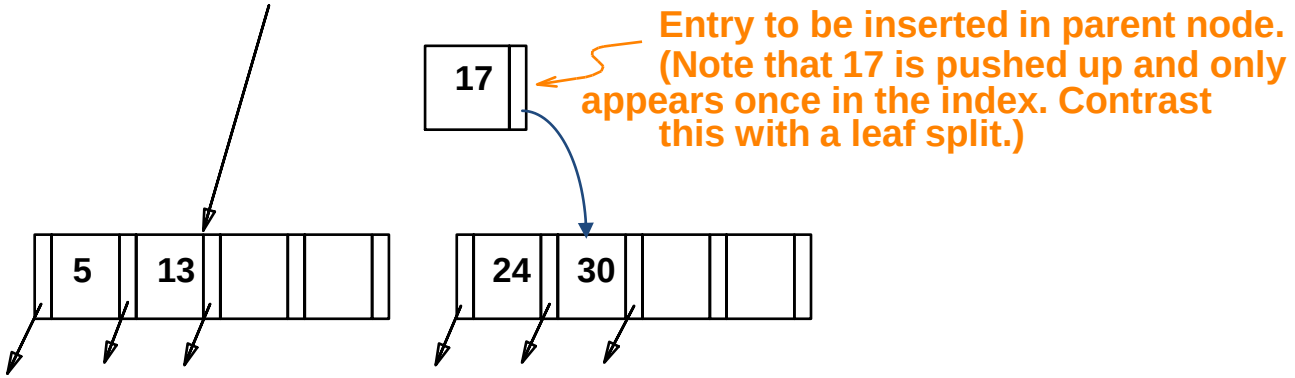
Insertion

- Inserting 15* is straightforward
- How about inserting 8*?



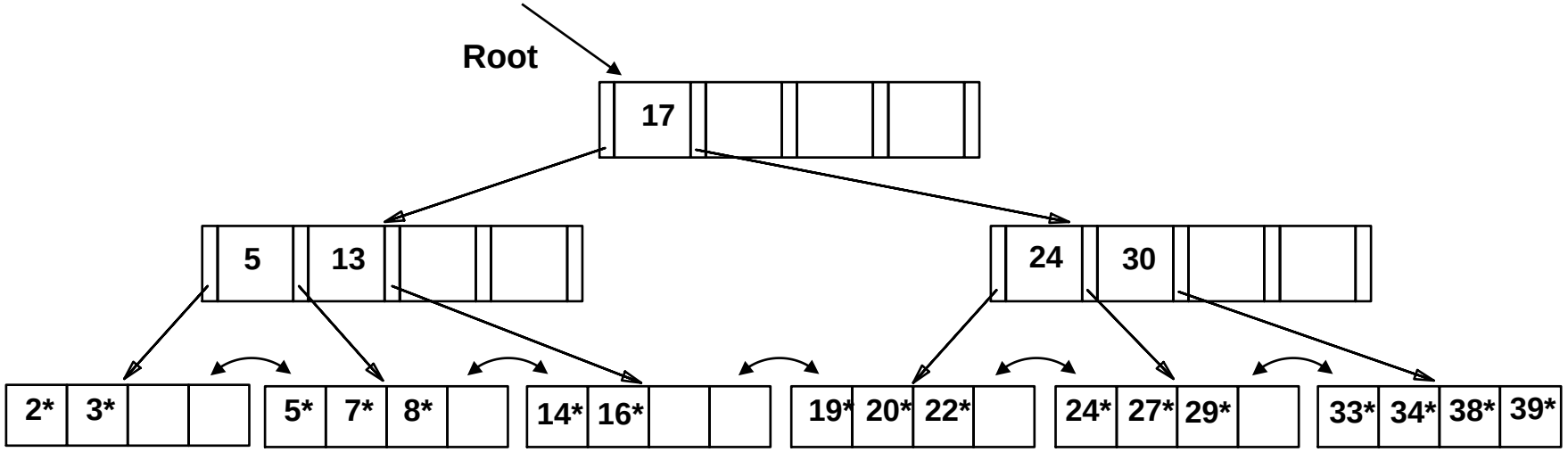
Insertion


5
2*
3*
5*
7*
8*
Entry to be inserted in parent node.
(Note that 5 is copied up and
continues to appear in the leaf.)


17
5
13
24
30
Entry to be inserted in parent node.
(Note that 17 is pushed up and only
appears once in the index. Contrast
this with a leaf split.)

Insertion

- After inserting 8*
  - Notice how the root was split, causing increased height
  - The end tree is balanced
  - “At least half full” constraint remains satisfied (how?)



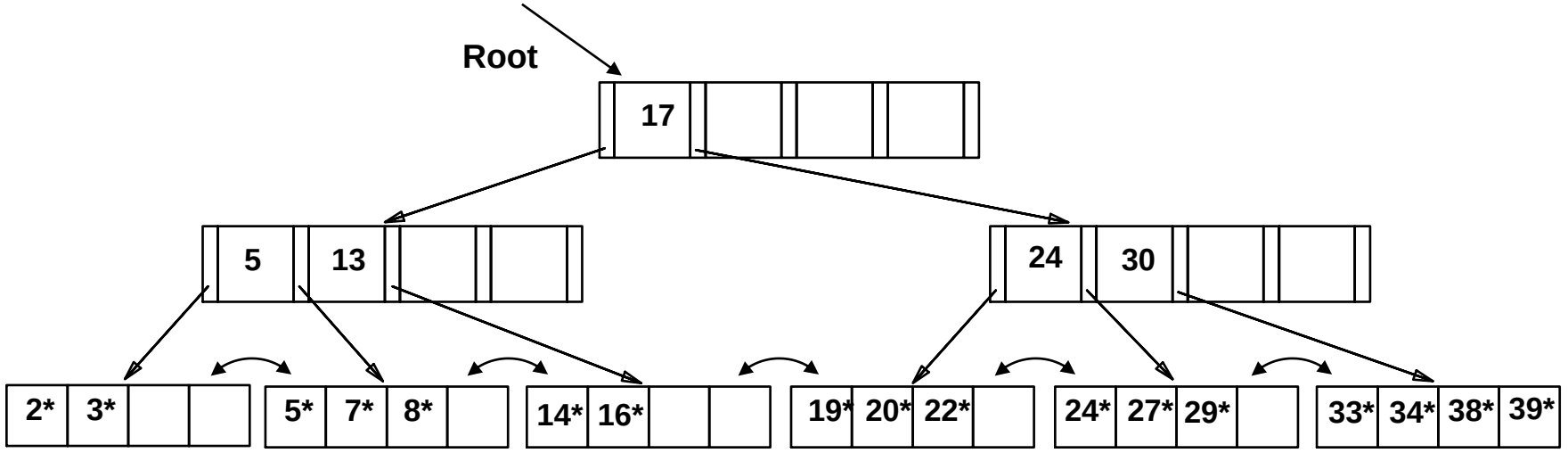
Deletion

- Start at root, find leaf *L* where entry resides.
- Remove the entry from *L*.
  - After removal, if *L* is at least half full, done!
  - Otherwise, try to re-distribute by borrowing from *L*'s sibling (adjacent node with same parent as *L*).
  - If re-distribution fails, merge *L* and its sibling.

- If a merge occurs, we must delete the entry (pointing to *L* or its sibling) from the parent of *L*.
- Merge could propagate all the way up to the root and cause decreased tree height.

Deletion

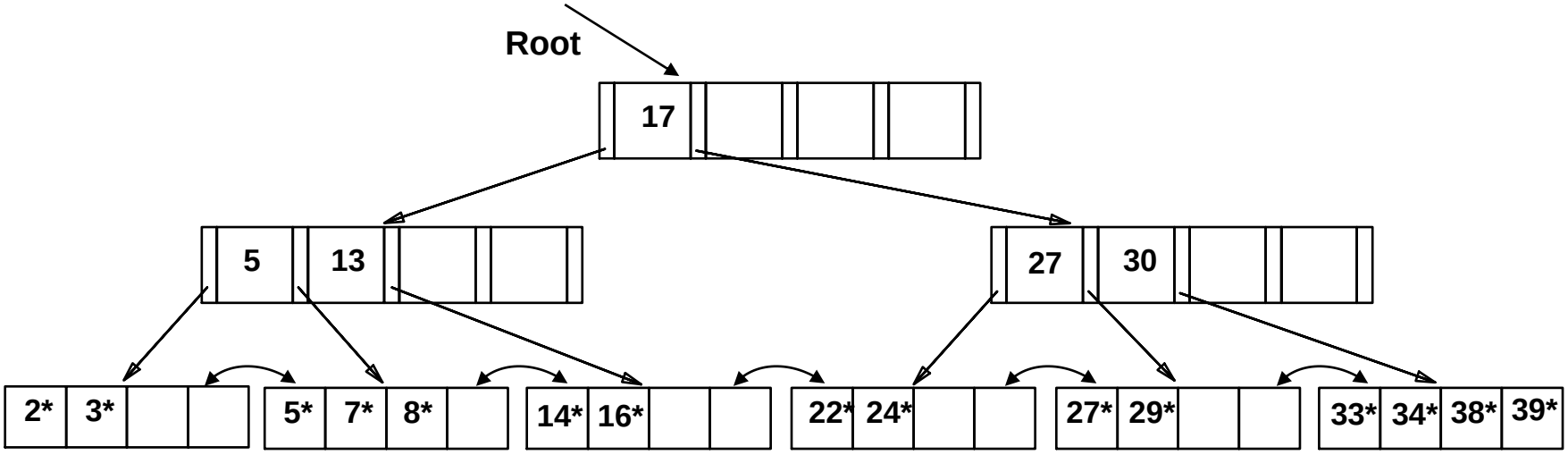
- Delete 19*, then delete 20*
  - Deleting 19* is easy
  - Deleting 20* requires re-distribution



Deletion

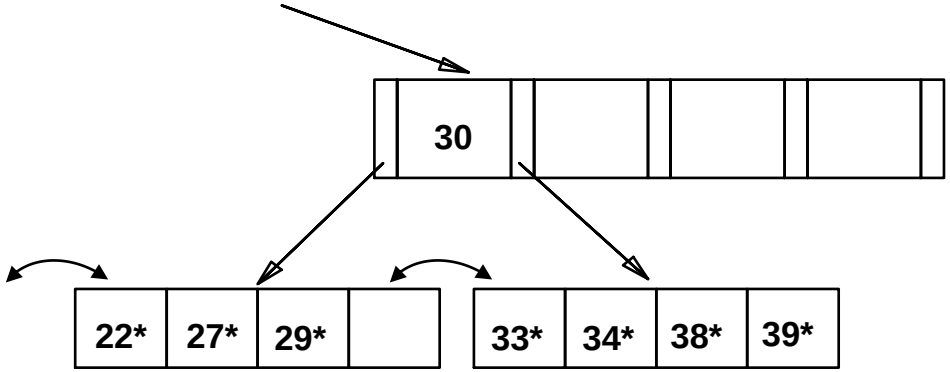
- Result after 19* and 20* are deleted
  - Notice how 24 in *L*'s parent changes to 27

- Now, let's delete 24*

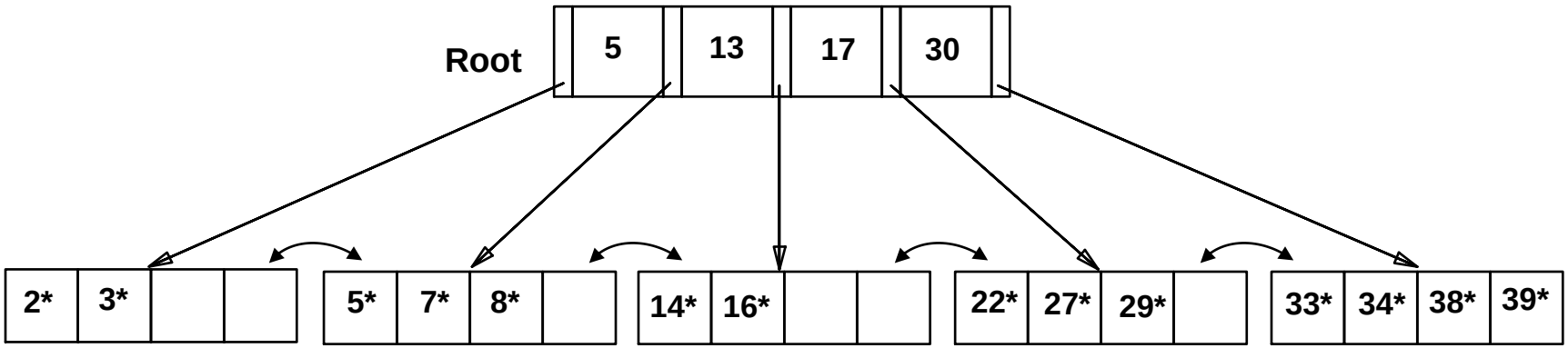


Deletion

- Must merge leaves and throw away 27 in the parent:



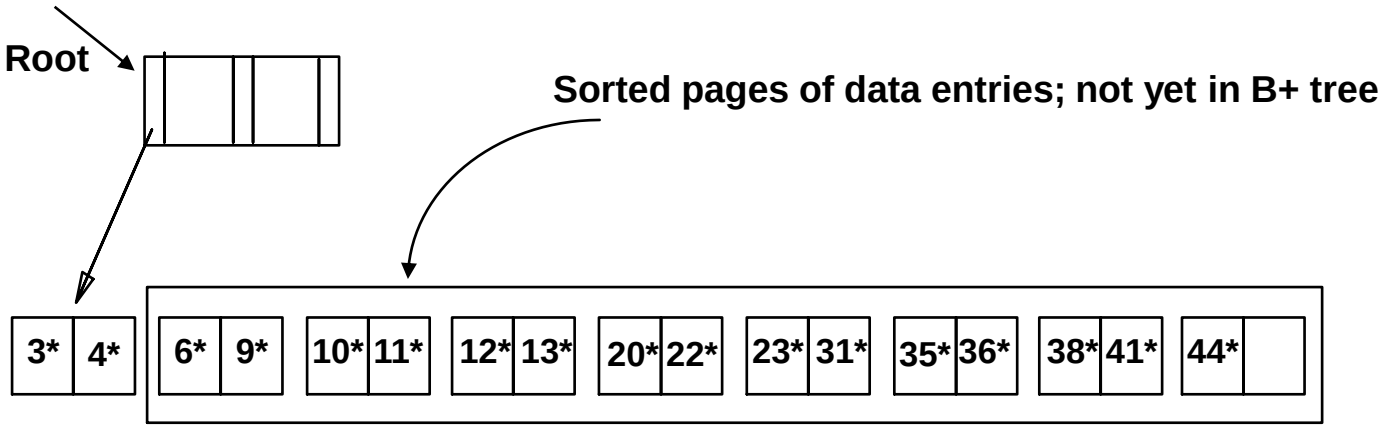
- But this time, the parent is no longer half full!
  - Merge propagates up one more level



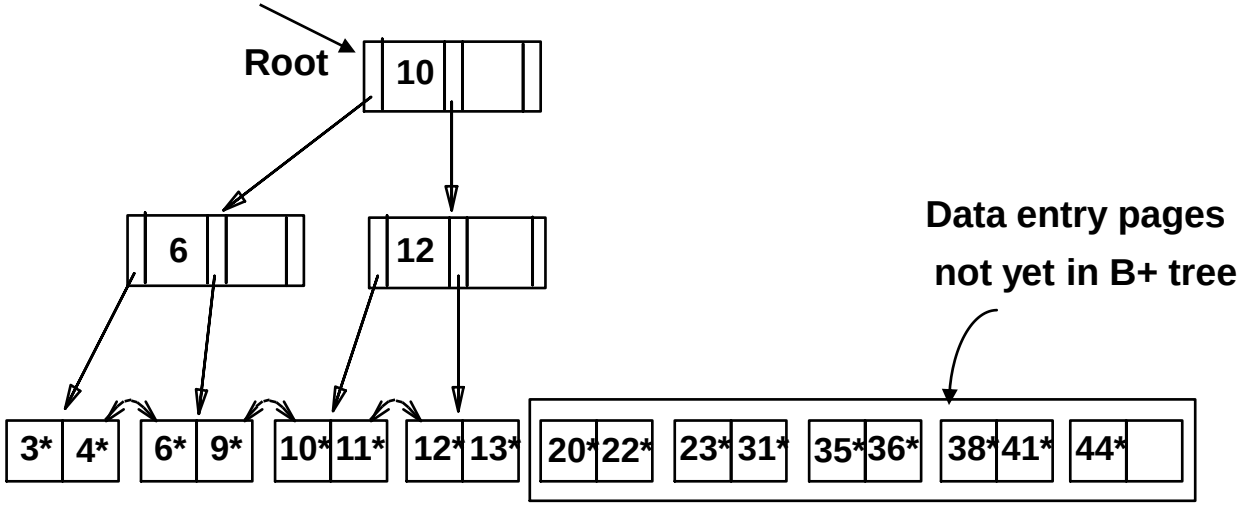
Bulk Loading

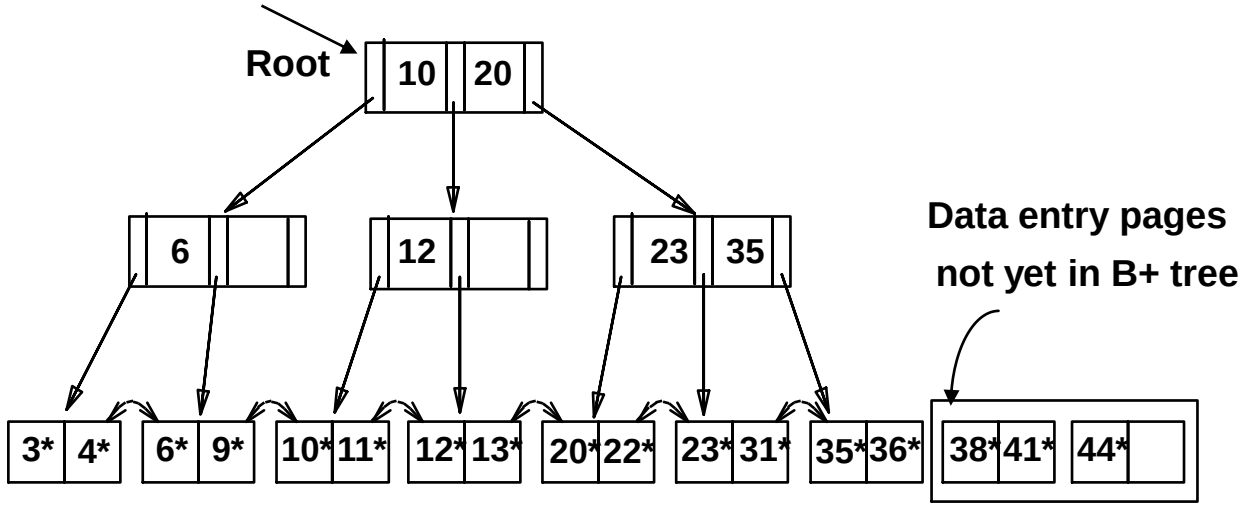
- If we have a large collection of records and we want to create a B+ tree with them, doing so by repeatedly inserting records in random order can be slow.
  - Can be solved by **bulk loading** of the B+ tree.

- Sort all records, initialize empty root and start adding.

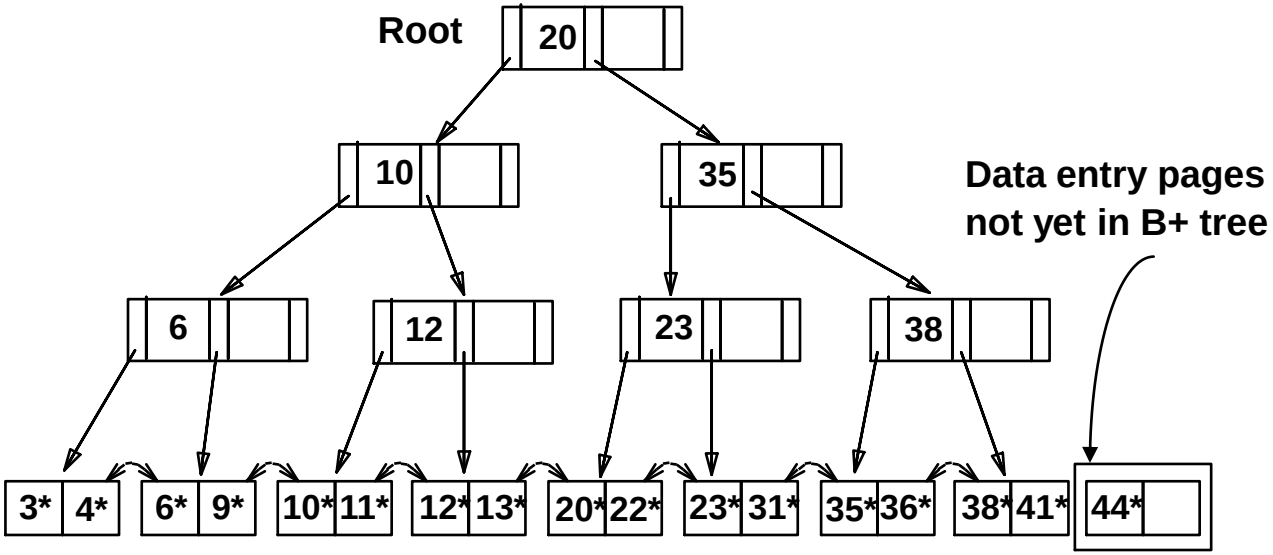


Bulk Loading



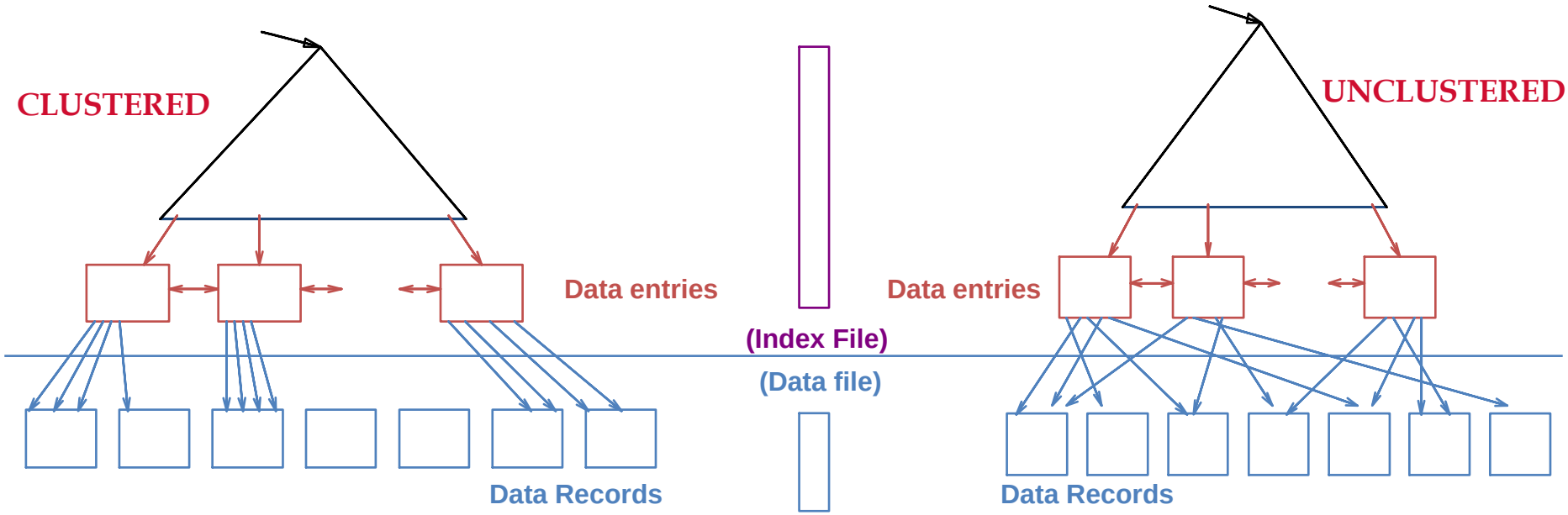


Bulk Loading



Clustered vs Unclustered Index

- In a clustered index, the physical storage layout of the tuples in pages will match the order in which they appear in the index.
  - How does this become useful in physical disk I/O?
  - In what ways could this be difficult? (Insert/delete)
- **Note:** Clustered indexes are not limited to B+ trees.



Summary

- Tree-structured indexes are good fit for indexing tables
  - Efficiently support range selection and equality selection
- ISAM tree is a static structure
  - Only leaf pages are modified
  - Insertions and deletions are straightforward
  - Long overflows can degrade performance
- B+ tree is a dynamic structure
  - Very popular and commonly used
  - Insertions and deletions are more involved than ISAM, but search remains O(logN)
  - Use bulk loading for the initial construction of B+ tree
- Clustered indexes determine physical storage layout