

# *Tree-Structured Indexes*

Davood Rafiei

# *Supported Search Operations*

- ❖ **Equality Search**: e.g. find the student with *sid*= "111222".
- ❖ **Range Search**: Find all students with *gpa*>3.
- ❖ If data was stored in a sorted file,
  - Can use binary search
  - Cost:  $\log_2 B$
- ❖ Can we reduce the cost?

# *Index File*

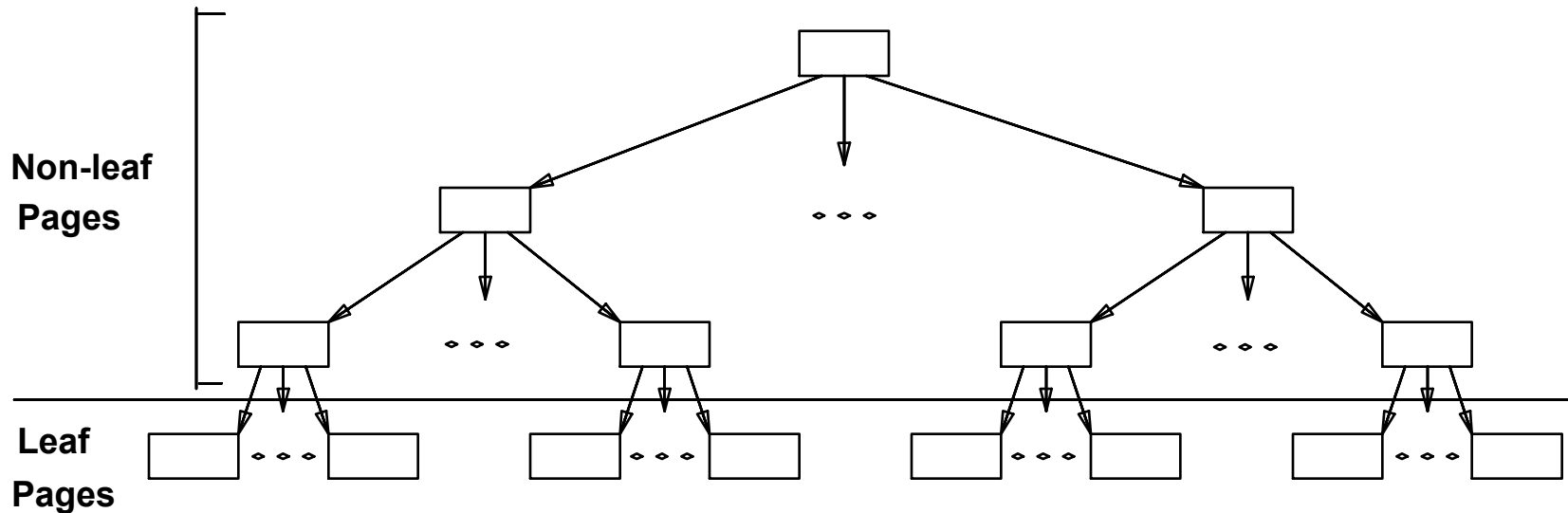
❖ Simple idea:



- ☞ Can do binary search on (smaller) index file.
- ☞ The index file can still be large!

# *Index File (cont.)*

❖ Can apply the idea repeatedly!

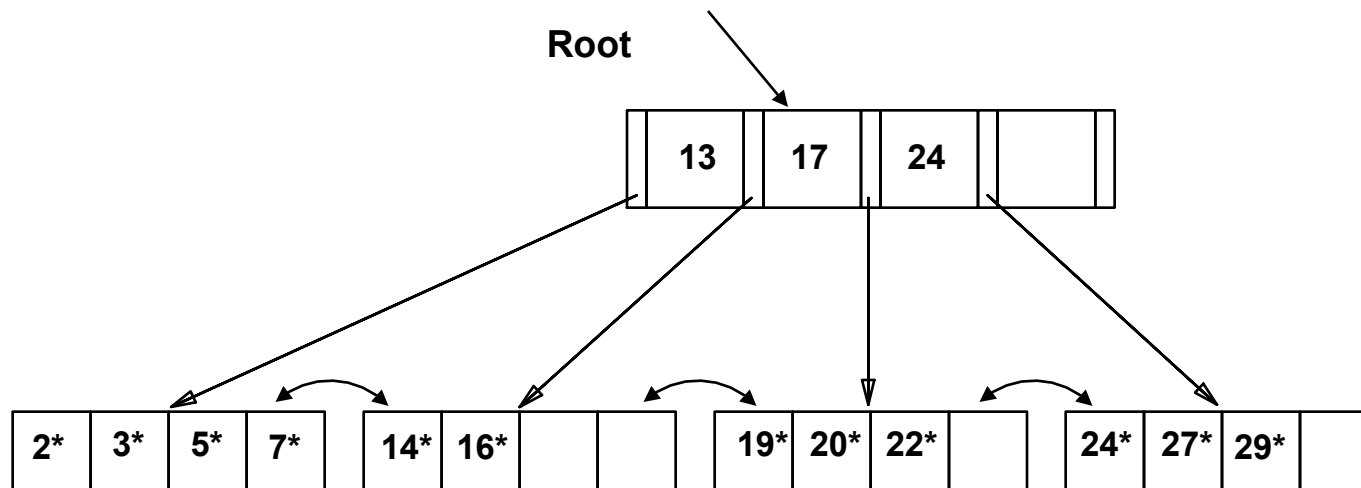


➡ Non-leaf pages contain *separators*.

➡ Leaf pages contain *index entries*.

# Tree Index Example

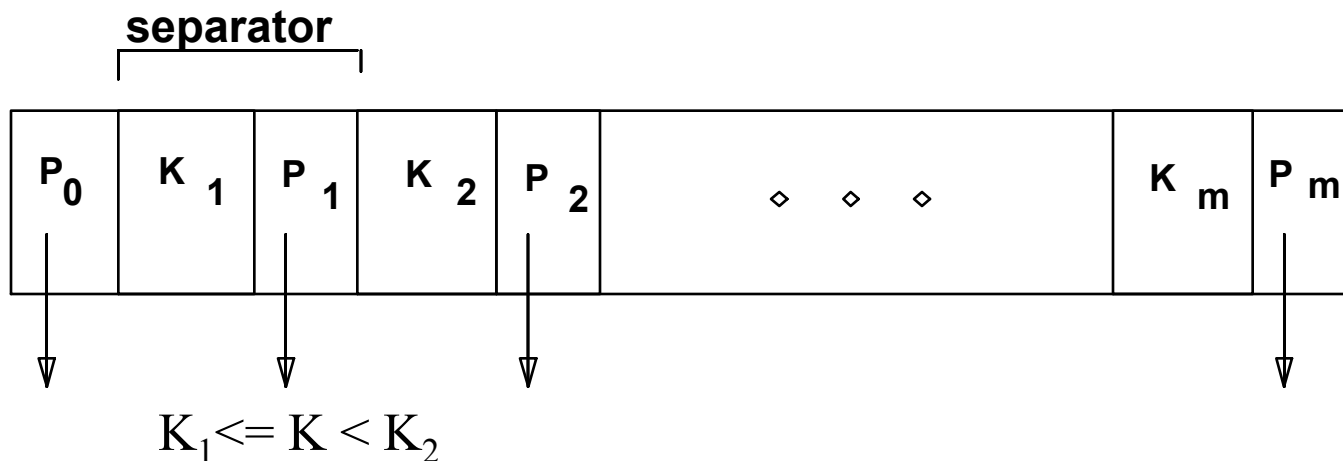
❖ Search for 5\*, 15\*, all data entries  $\geq 20^*$  ...



👉 *Based on the search for 15\*, we know it is not in the tree!*

# Searching the Index

- ❖ Separators direct searches to index entries.
- ❖ Search: Start at root; use key comparisons to go to leaf.
  - Cost:  $\log_F N$   
F = Fan-out, N = # leaf pages.



# Updating the index

## ❖ Static index structure: ISAM

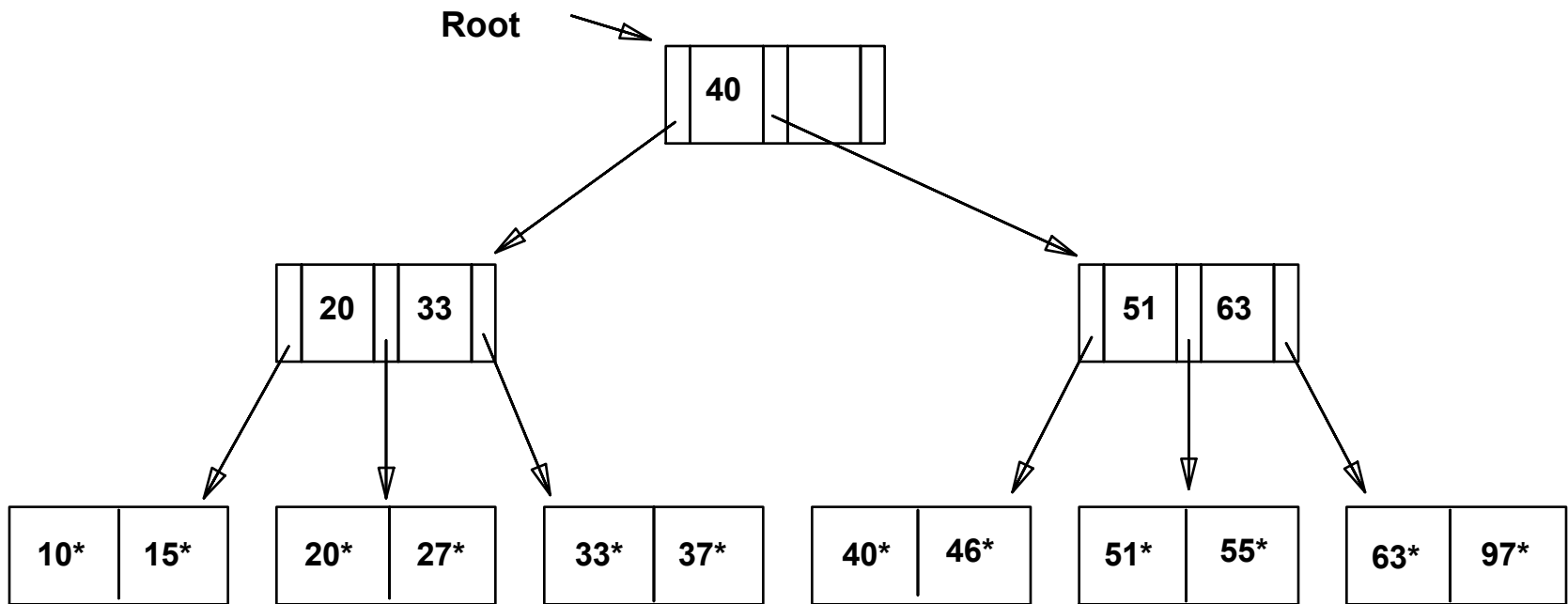
- Inserts and deletes only affect leaf pages.
- Insert: Find the leaf page data entry belongs to, and put it there. If there is no space, allocate an overflow page.
- Delete: Find and remove from leaf; if empty overflow page, de-allocate.

## ❖ Dynamic Index structure: B+ tree

- Adjust the tree as data entries are inserted/deleted.

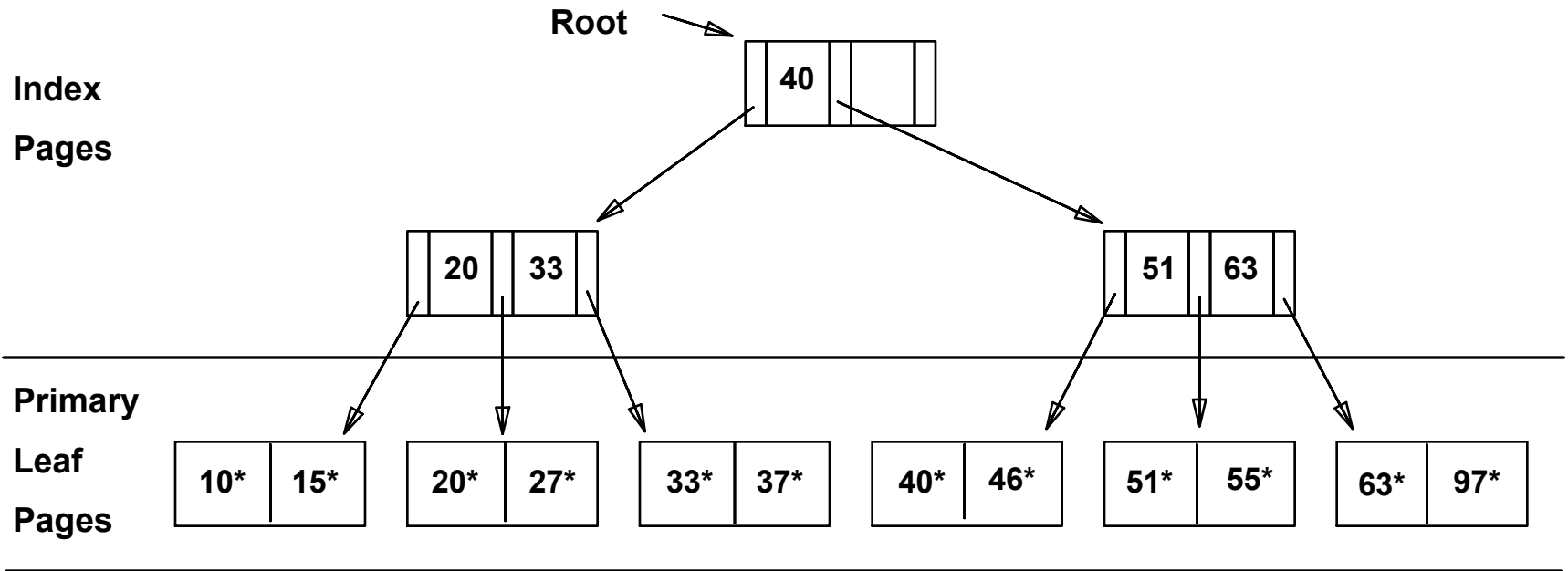
# Example ISAM Tree

❖ Each node can hold 2 entries.



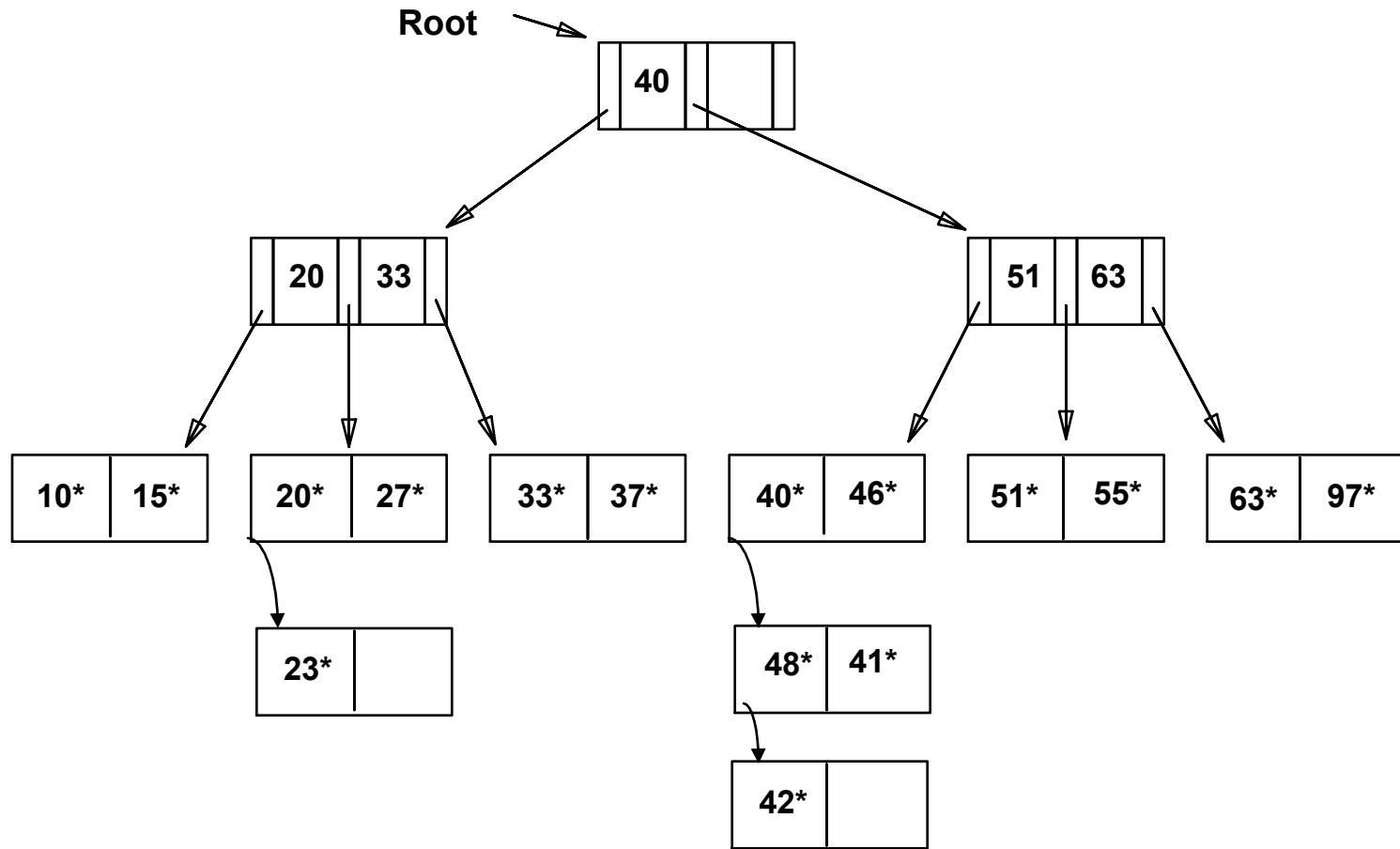


*Insert 23\*, 48\*, 41\*, 42\* ...*

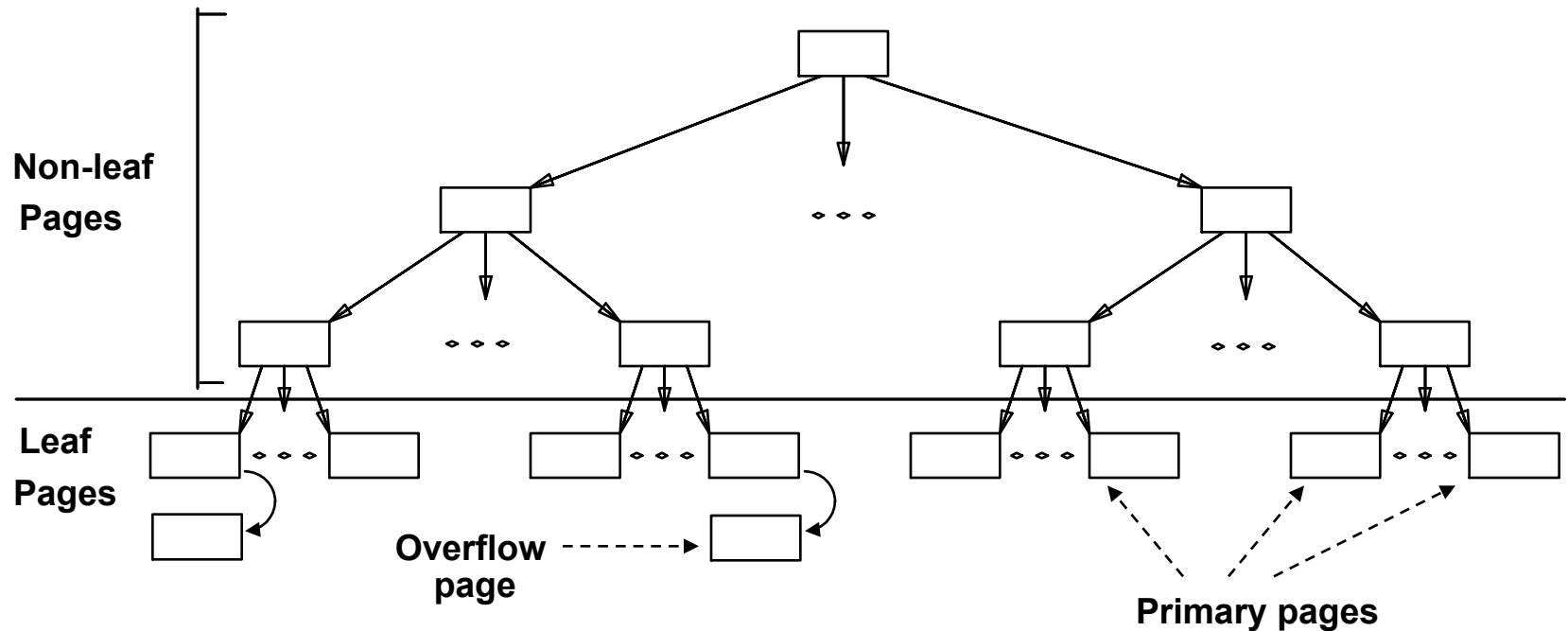


**Overflow  
Pages**

*... Then Delete 42\*, 51\*, 97\**



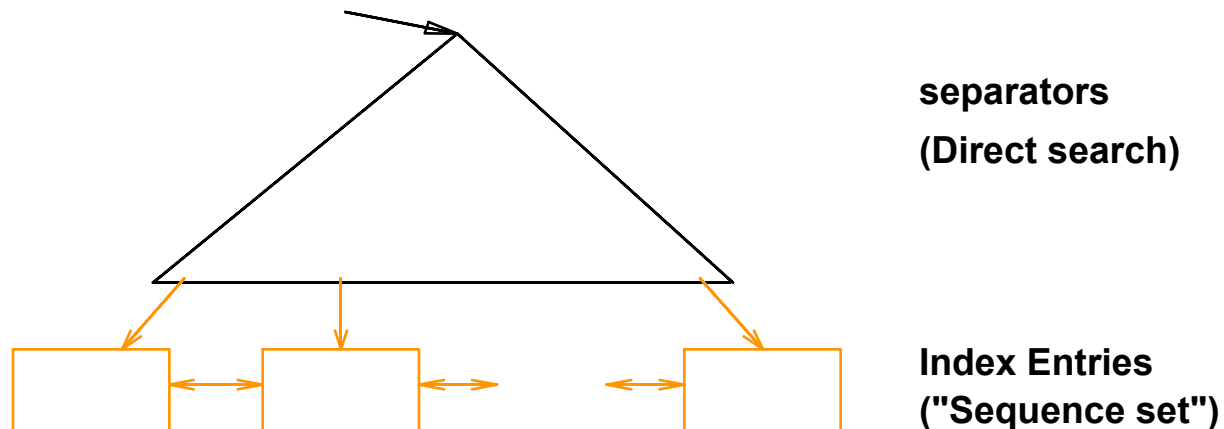
# ISAM



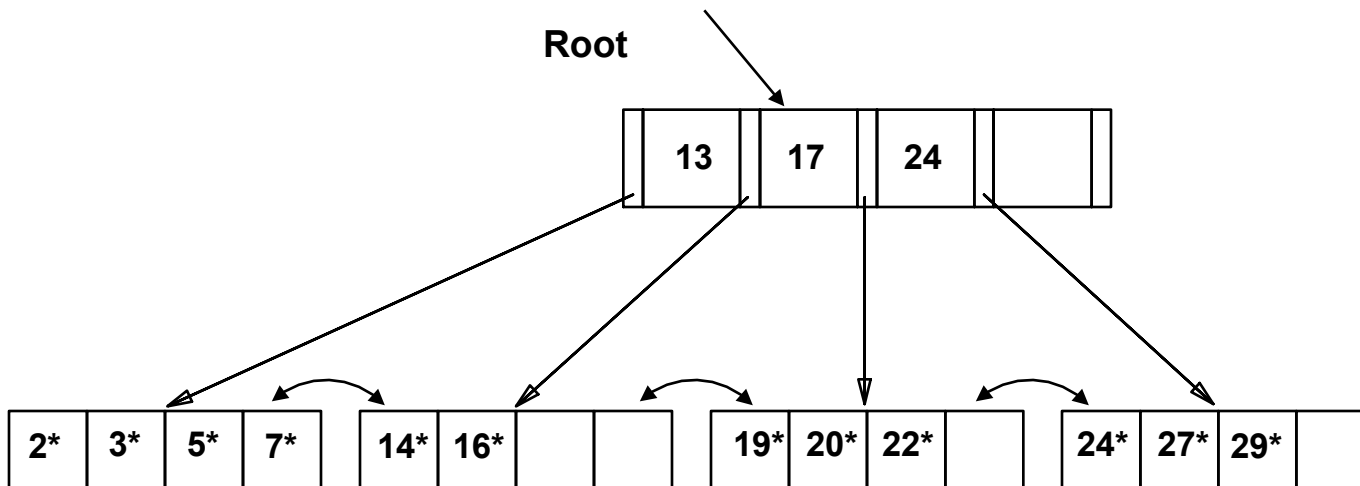
- ❖ The tree after some updates
- ❖ Cost of a search can be more than  $\log_F N$   
(depending on the number of overflow pages)

# B+ Tree

- ❖ Main features:
  - Search/insert/delete guaranteed at  $\log_F N$  cost.
  - Minimum 50% occupancy (except for root).
  - Leaf pages form a sequence set.
- ❖ Everything else is much like ISAM.

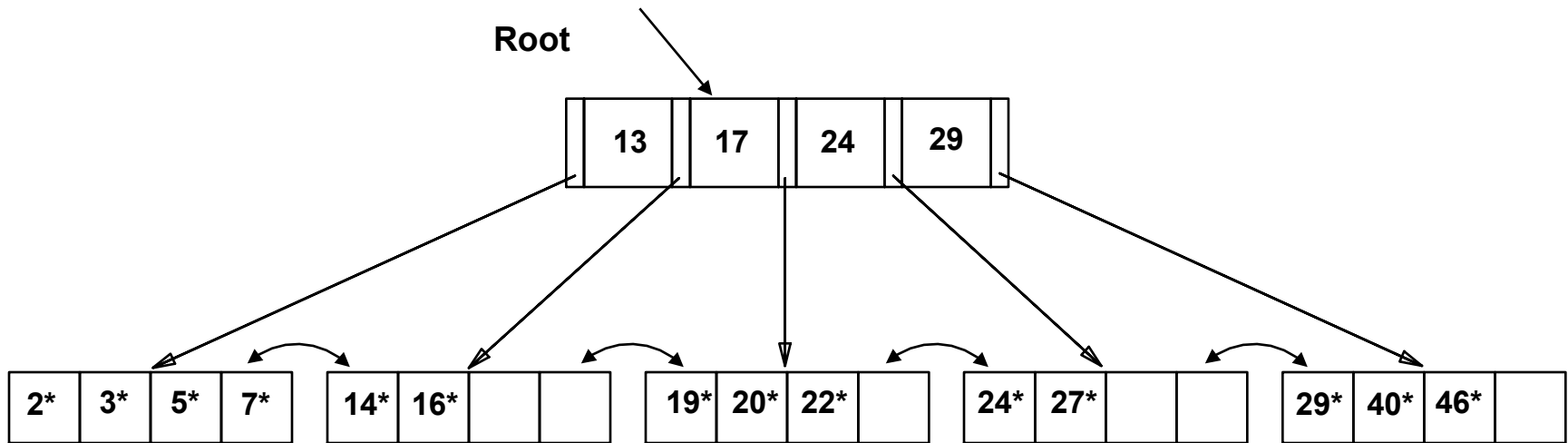


*Insert 40\*, 46\**



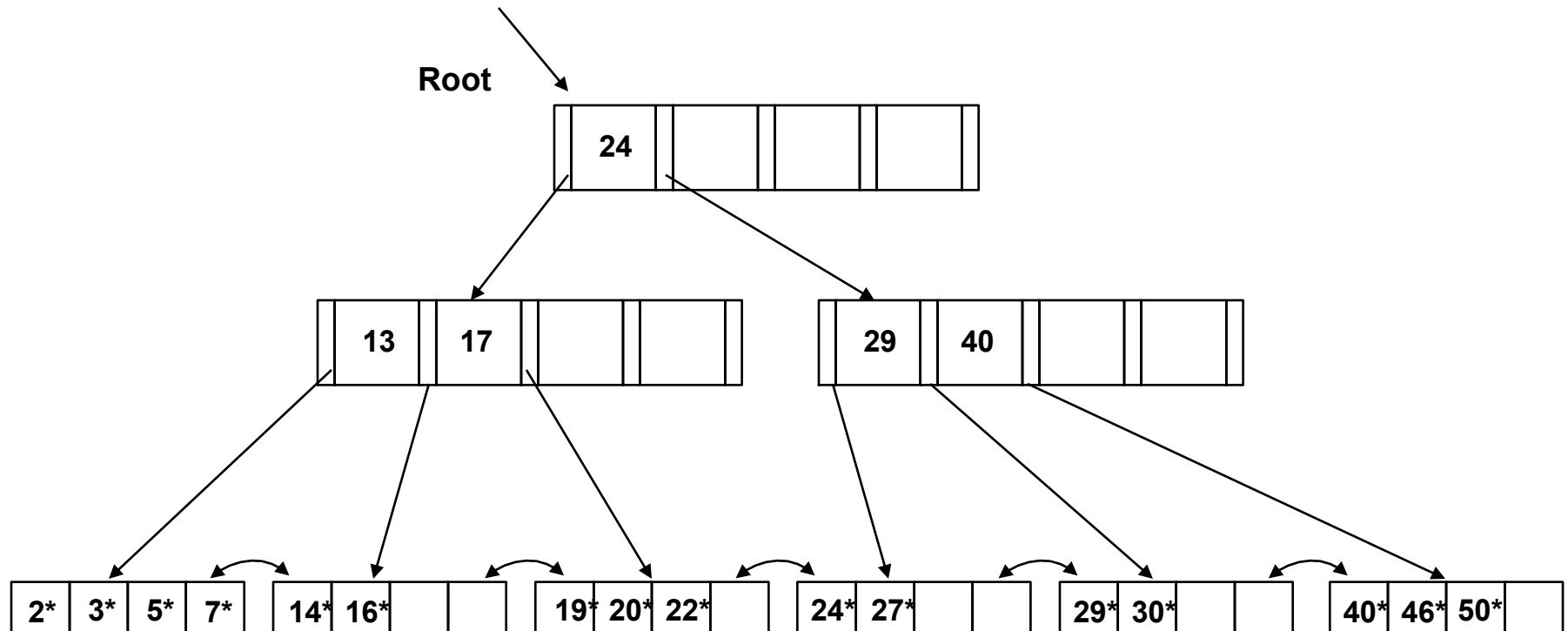
➡ Inserting 46\* causes a leaf split: *copy-up*

*Insert 50\*, 30\**

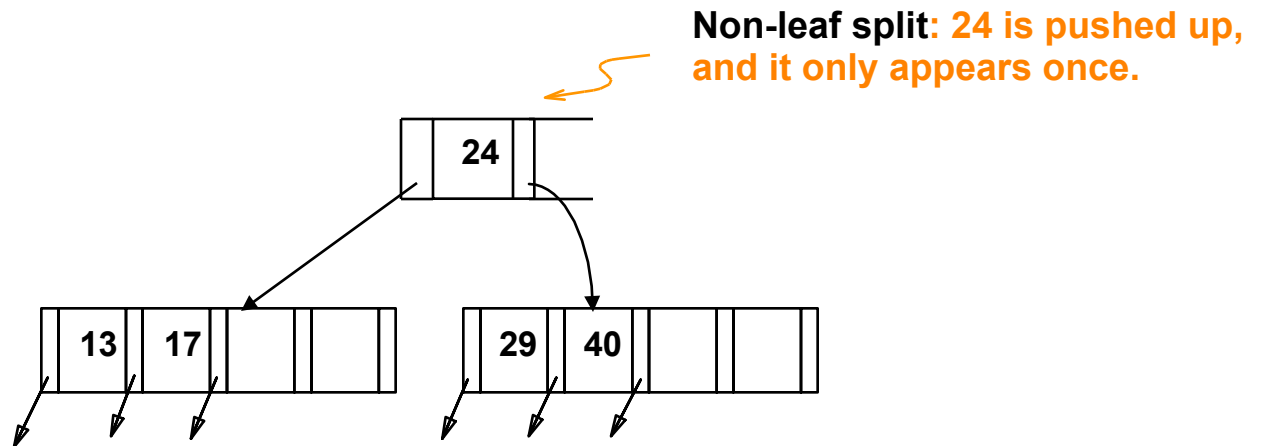
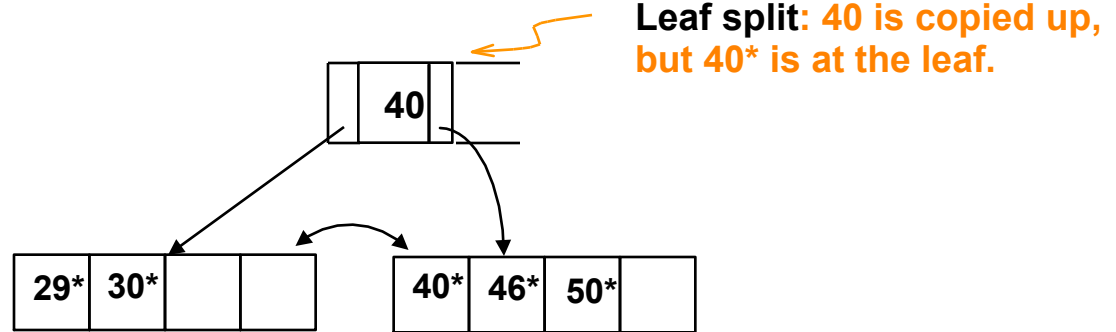


- ☞ Split propagates to the root.
- ☞ Non-leaf split: *push up*.

# *Example B+ tree After Insertions*



# Split Policy

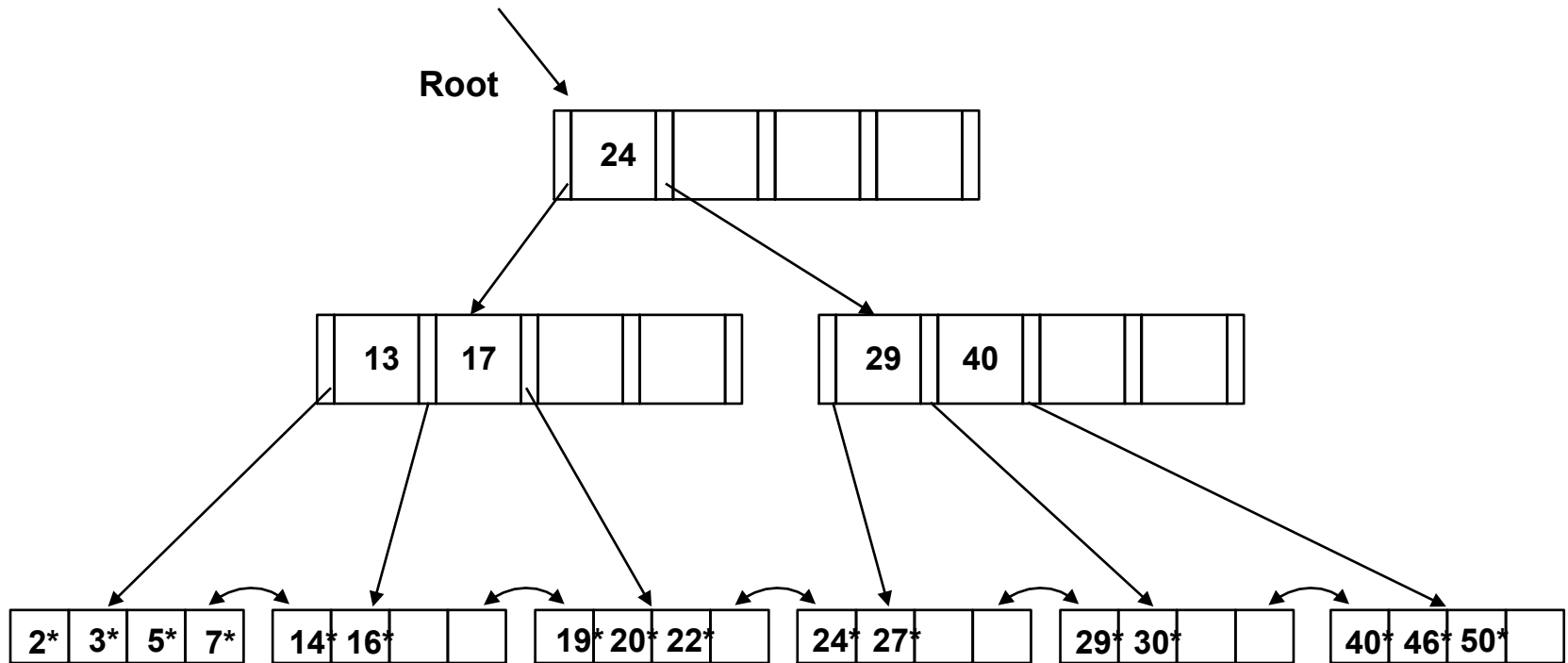




# *Inserting a Data Entry into a B+ Tree*

- 1) Find correct leaf node
- 2) Add index entry to the node
- 3) If enough space, *done!*
- 4) Else, split the node
  - ❖ Redistribute entries evenly between the current node and the new node
- 5) Insert *<middle key, ptr to new node>* to the parent
- 6) Go to Step 3

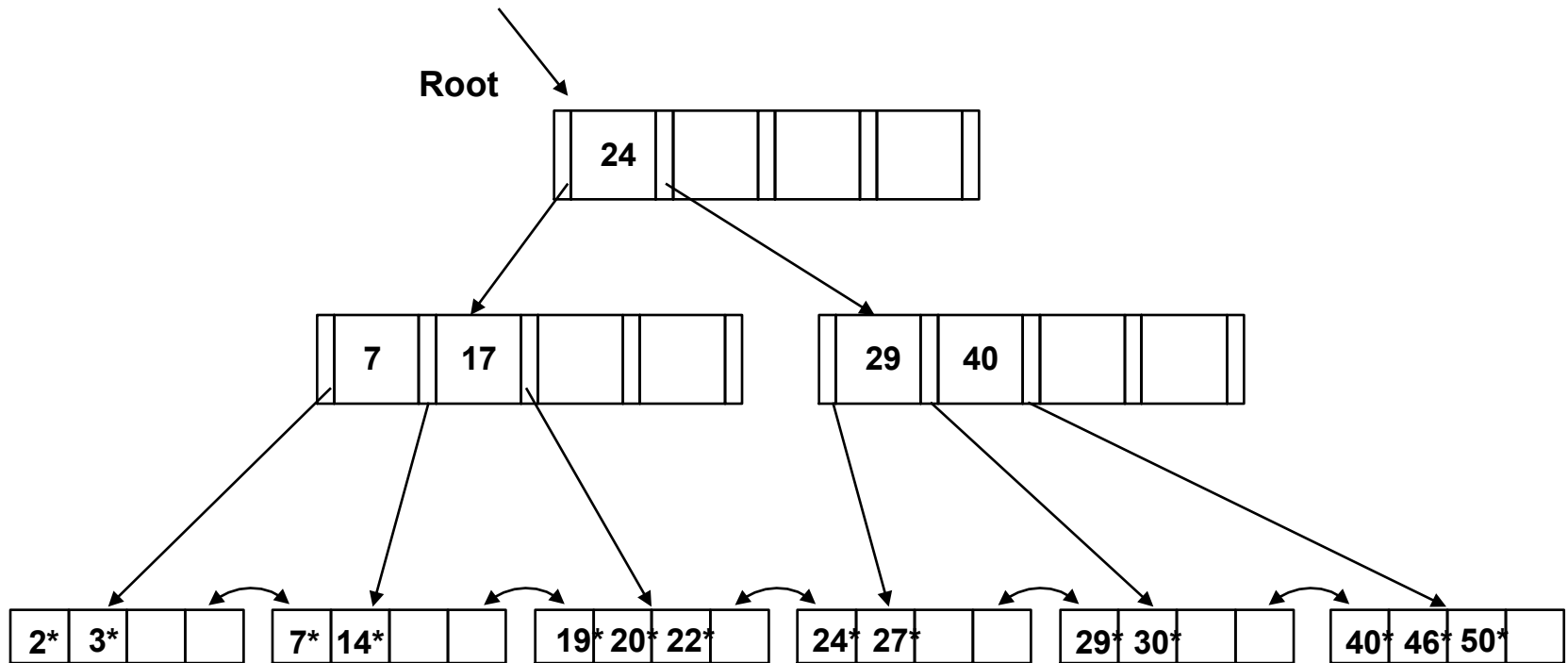
# Delete 5\* and 16\*



## ❖ Deleting 16\*

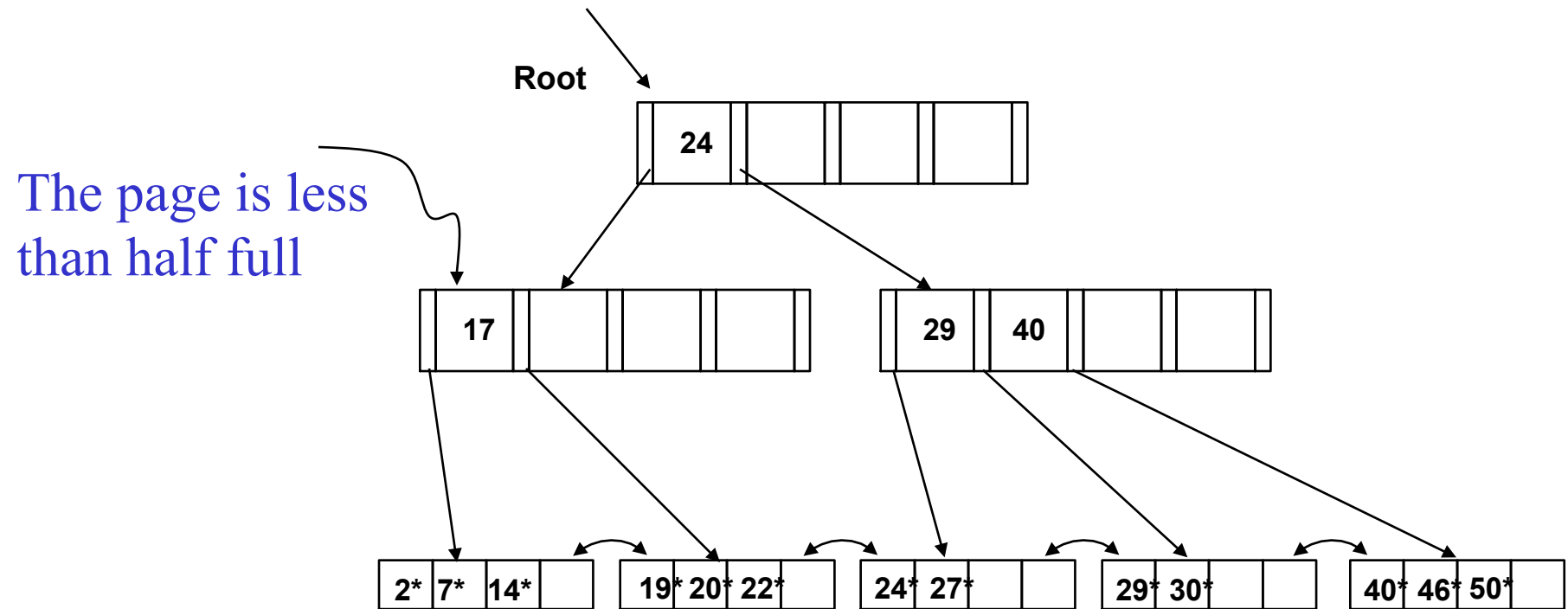
- The page becomes less than half full!
- Borrow some keys from a neighbour (redistribute the keys equally between them): *copy up*.

# Delete 3\*



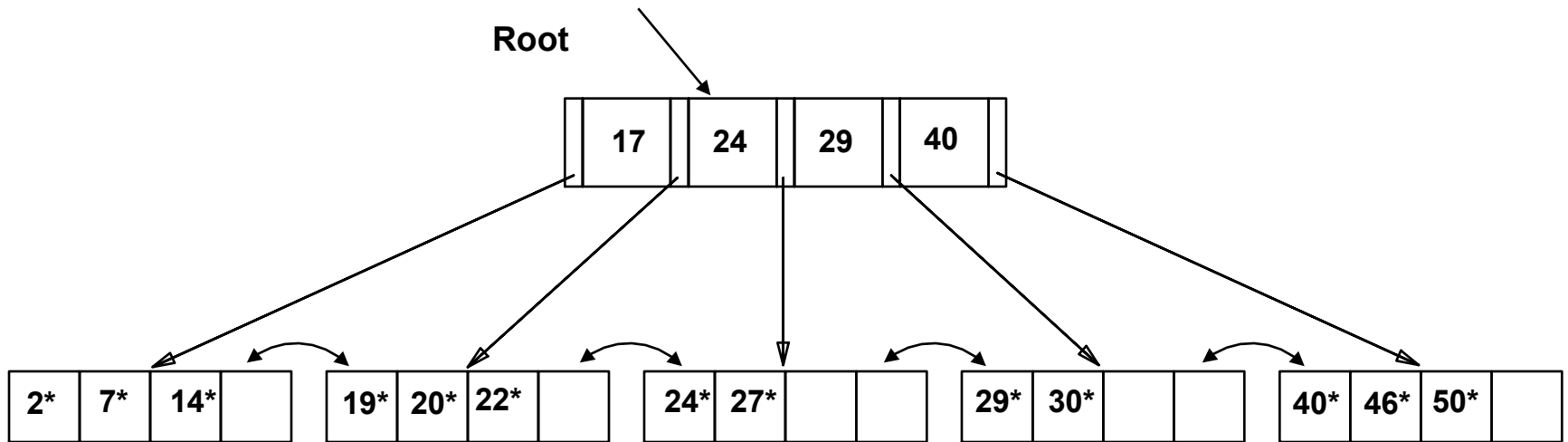
- ❖ Cannot borrow from a neighbour.
- ❖ Merge the page with its neighbour.

# The tree after merging the leaves



- ❖ Cannot borrow from a neighbour.
- ❖ Merge again: *pull down*

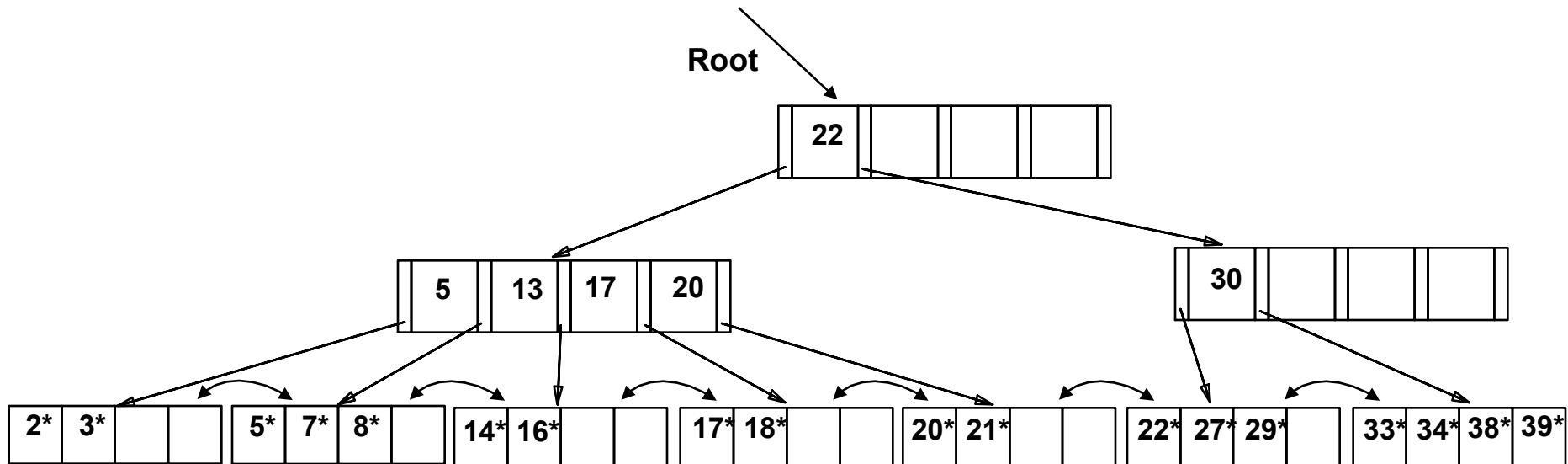
# *Example B+ tree after the deletion*



❖ New root at one level lower.

# *Another Example of delete*

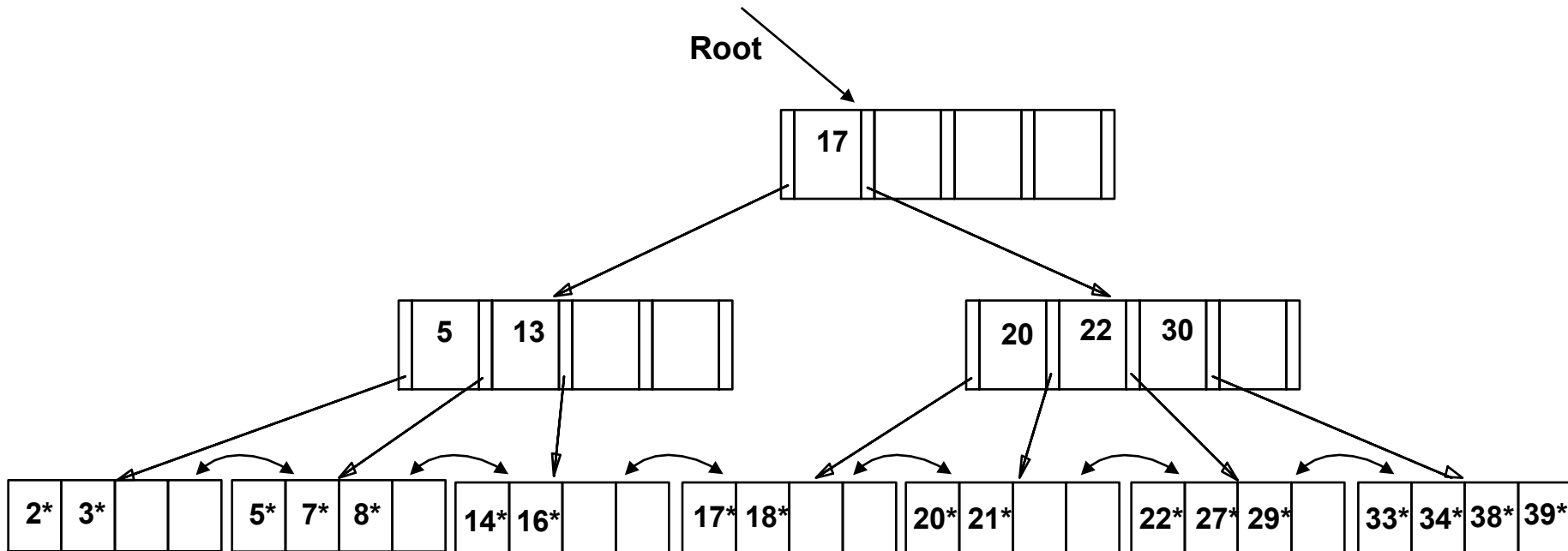
❖ The tree after a merge in the leaf layer:



- The node in the middle layer is less than half full.
- Redistribute the keys between the page and its neighbour.

# After Re-distribution

- ❖ Intuitively, entries are *re-distributed by 'pushing through'* the splitting entry in the parent node.
- ❖ It suffices to re-distribute index entry with key 20; we've re-distributed 17 as well for illustration.



# *Deleting a Data Entry from a B+ Tree*

- 1) Find correct leaf node
- 2) Remove the entry from the node
- 3) If the node is at least half full, *done!*
- 4) Else, possibly *borrow* some entries from a sibling
- 5) If not possible, *merge* the node with the sibling
- 6) Delete the separator between the node and the sibling from the parent node
- 7) Go to Step 3



# *B+ Trees in Practice*

## ❖ Typical trees

- maximum fanout: 200
- fill-factor: 67%.
- average fanout = 133

## ❖ Typical capacities:

- Height 4:  $133^4 = 312,900,700$  index entries
- Height 3:  $133^3 = 2,352,637$  index entries

## ❖ Can often hold top levels in buffer pool:

- Level 1 = 1 page = 8 Kbytes
- Level 2 = 133 pages = 1 Mbyte
- Level 3 = 17,689 pages = 133 MBytes

# *B+-tree Index Variations*

- ❖ Index entry
  - <full record>, <key, address(es)>, <key, address(es), some other columns>
- ❖ Character string keys
- ❖ Variable length keys
  - When is a node half full?
- ❖ Prefix B+-tree