COURSE 9

Tree Structured Files

Binary Tree Organization

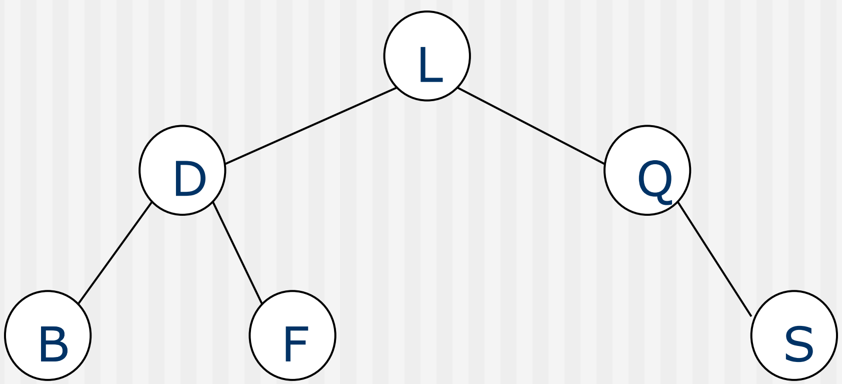
- Heap and sorted files - useful for static files
- Binary tree organization:
  - efficient inserting / deleting records
  - uses binary search algorithm
- Memory structure for a binary tree node:

K	Data	$\text{Pointer}_{\text{Left}}$	$\text{Pointer}_{\text{Right}}$

- Memory structure for a binary tree file:
  - collection of nodes; root pointer
  - list of empty nodes (linked by $\text{Pointer}_{\text{Left}}$)

Binary Tree Organization (cont.)

- Root – pointer to root
- Free – pointer to empty nodes list head
- Conceptual tree:

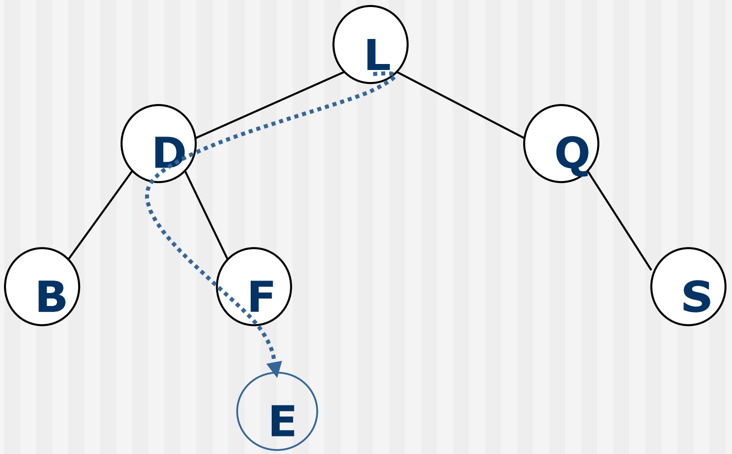


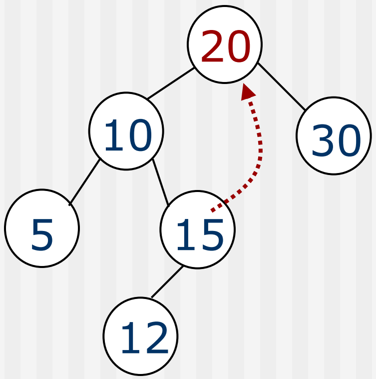
Root = 1
Free = 3

1	L	$Data_L$	2	4
2	D	$Data_D$	8	7
3			-6	NULL
4	Q	$Data_Q$	NULL	5
5	S	$Data_S$	NULL	NULL
6			-9	NULL
7	F	$Data_F$	NULL	NULL
8	B	$Data_B$	NULL	NULL
9			NULL	NULL

Inserting / Removing Records

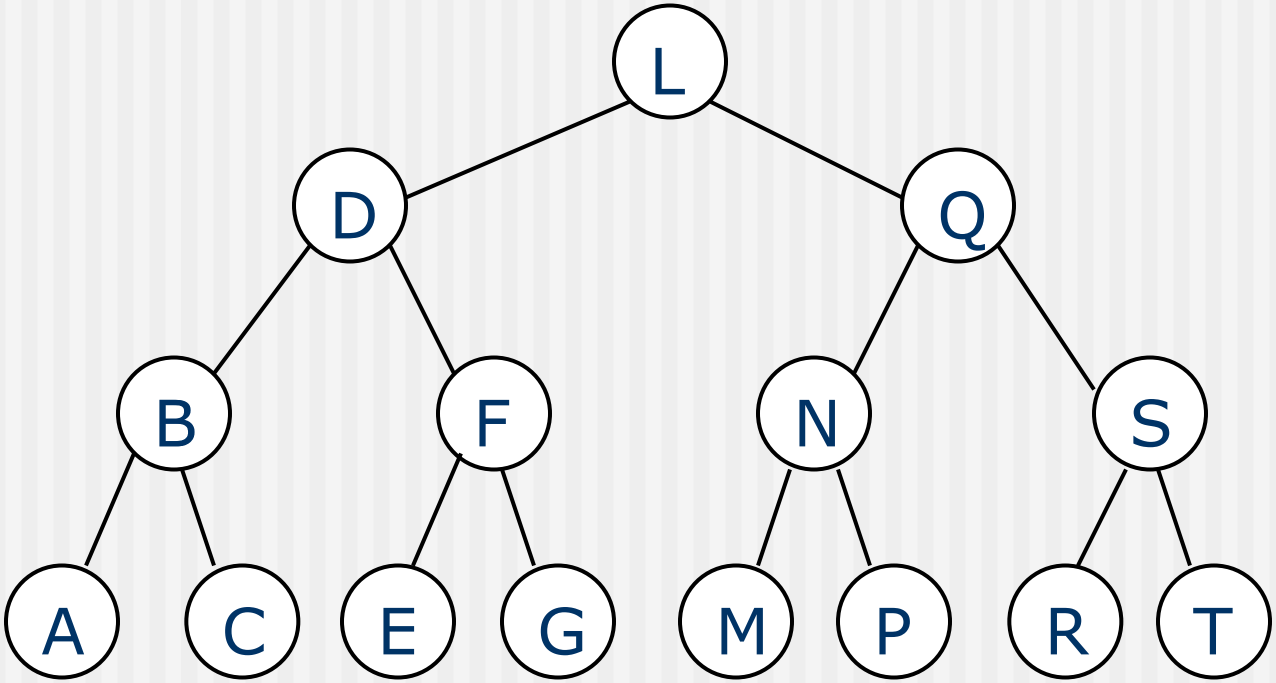
- Insert a record:
  - detect the position of the record
  - store new record in a free node
  - link the node its parent
- Remove a record:
  - search for the record
  - 3 cases:
    - no children: parent's pointer= NULL
    - 1 child: attach child to parent
    - 2 children: replace with the closest neighbor value
  - add node to empty nodes list





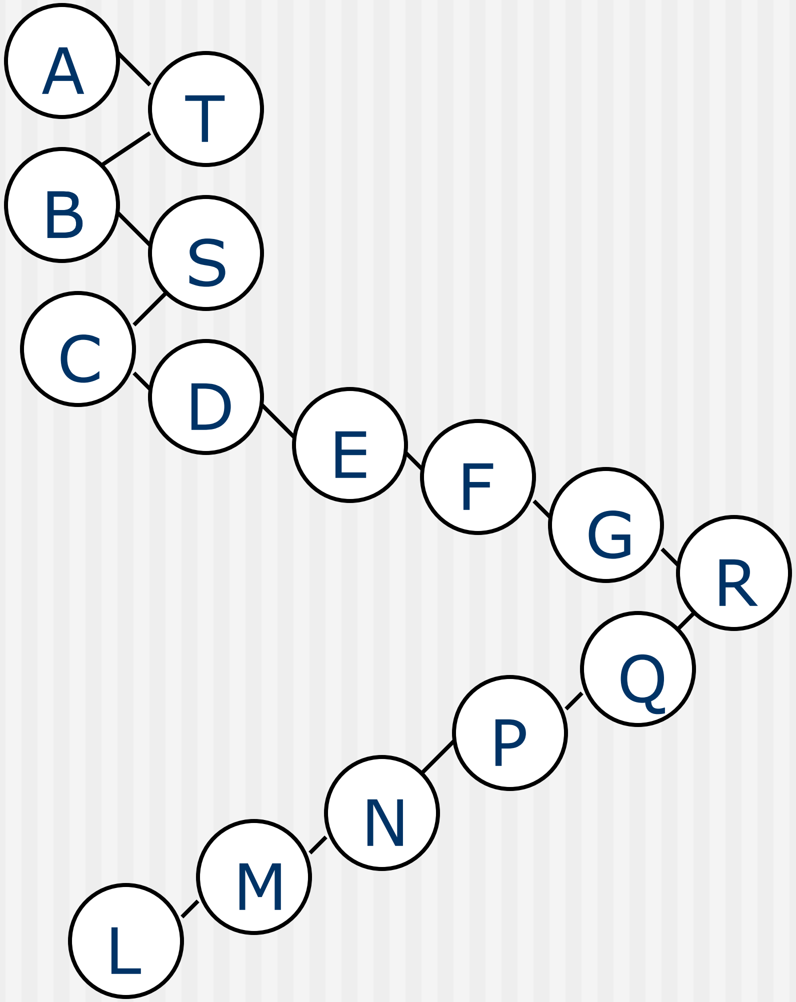
Insertion Anomaly in Binary Tree

- L, D, B, Q, N, F, S, R, T, M, E, G, P, A, C



Insertion Anomaly in Binary Tree (cont.)

- A, T, B, S, C, D, E, F, G, R, Q, P, N, M, L



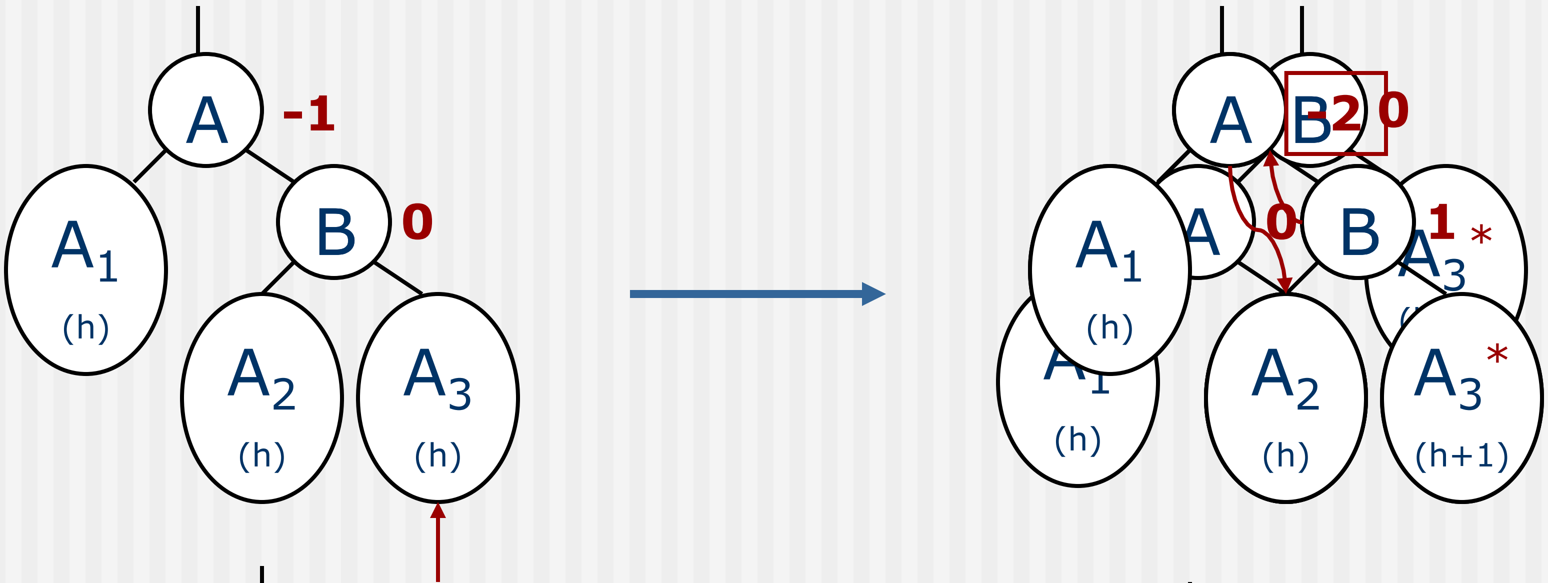
Optimal vs. Balanced Binary Trees

Drawback: searching time depends on inserting order

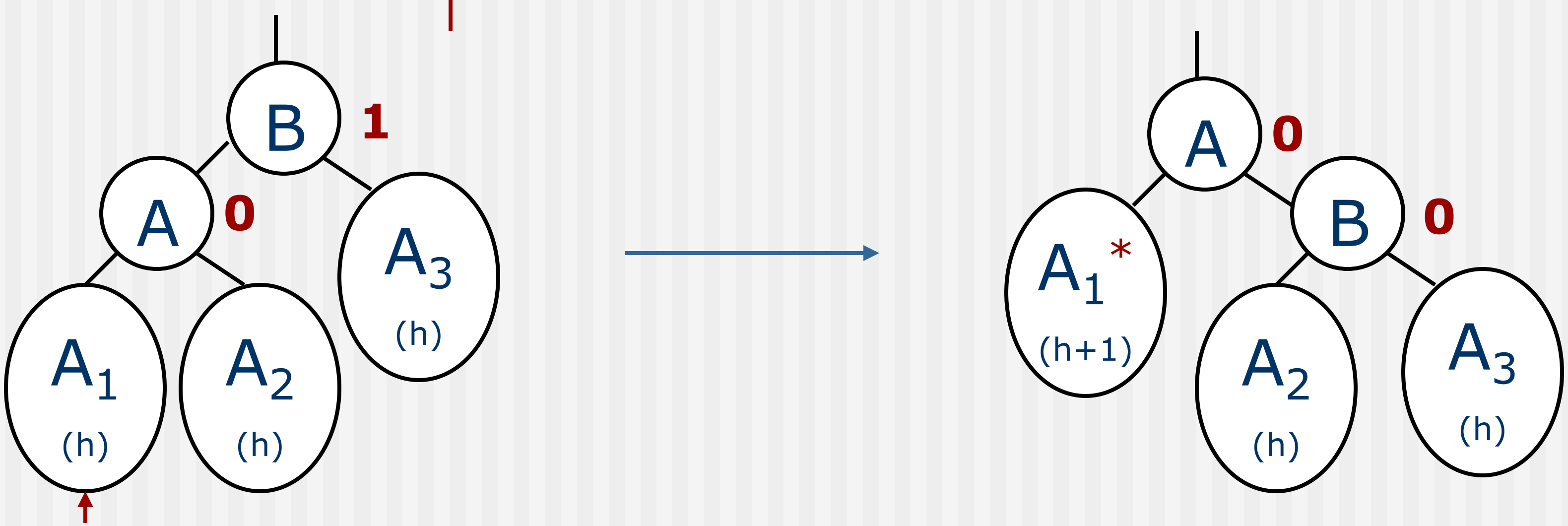
- Optimal tree
  - leaves are positioned on at most 2 levels
  - maintenance is difficult / time consuming
- Balanced tree
  - for each node the difference between its sub-trees *heights* is 0, 1 or –1 (*tree height*: length of the longest path from root to leaves)
  - lower number of operations for maintenance
  - 6 distinct cases when a tree become unbalanced after insertion

Balanced Trees Maintenance

(1)

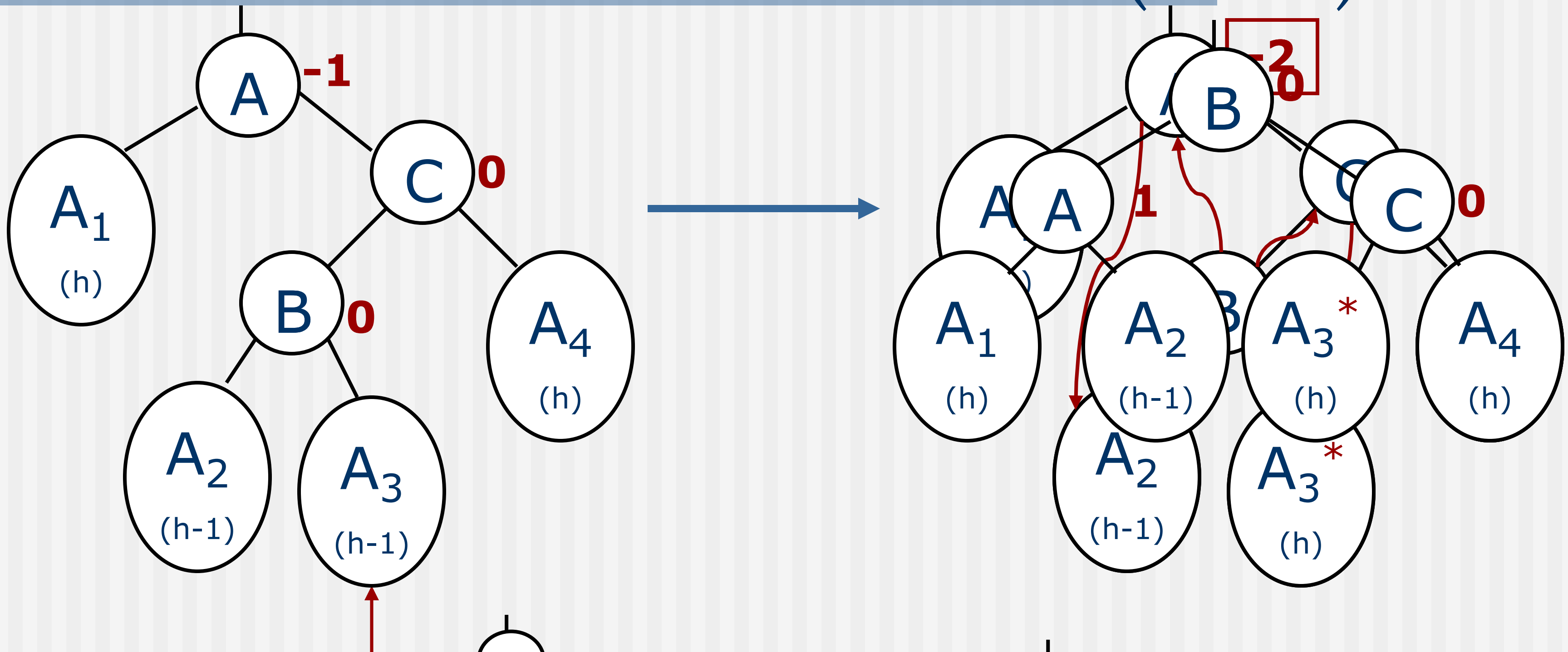


(2)

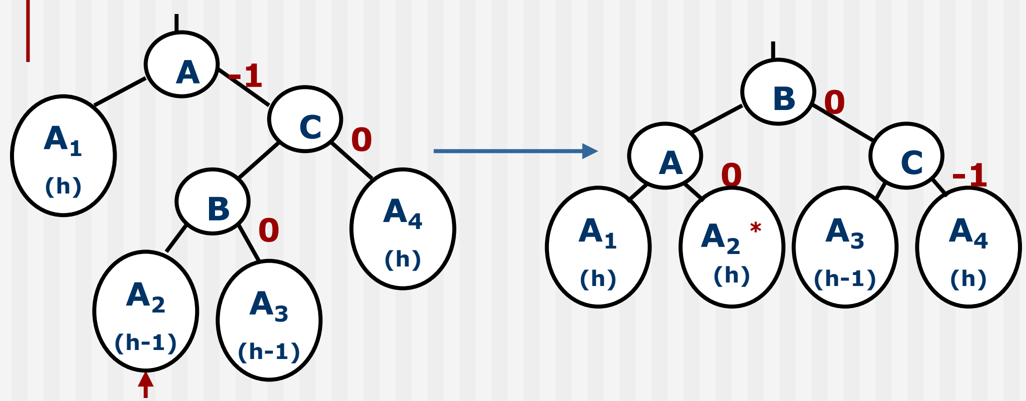


Balanced Trees Maintenance (cont.)

(3)

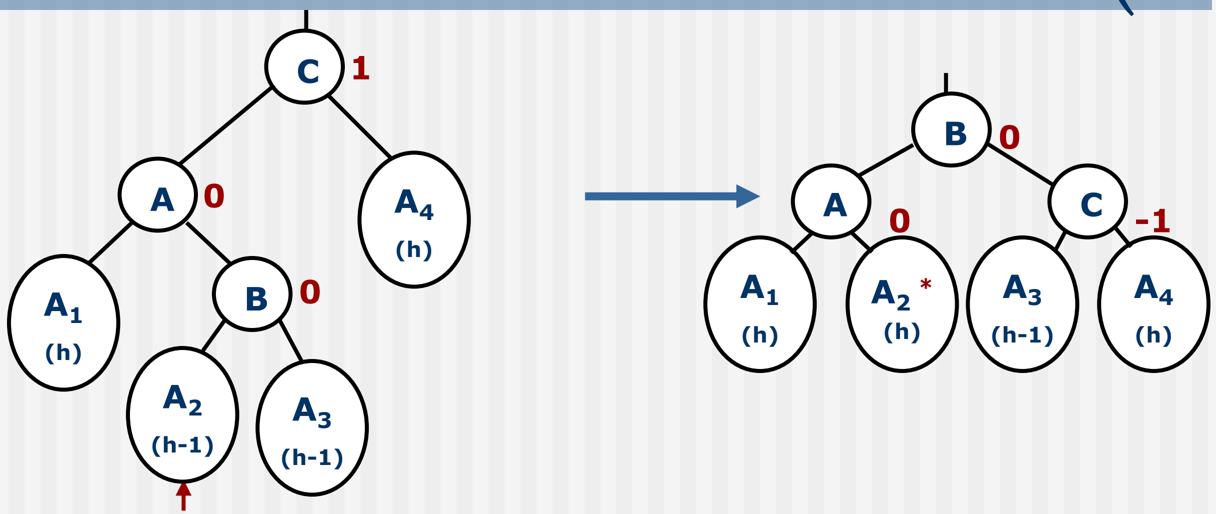


(4)

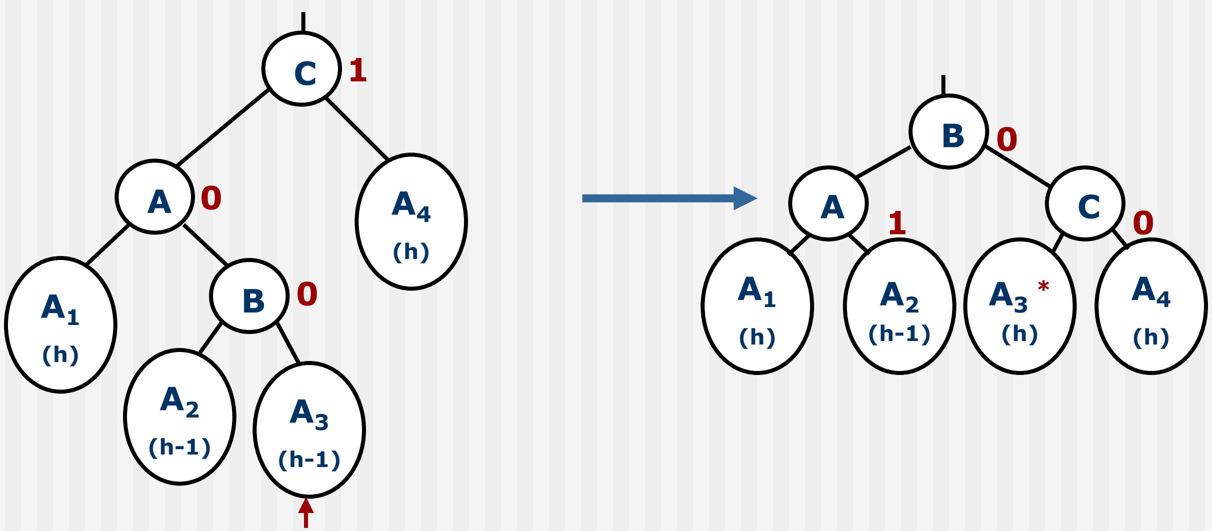


Balanced Trees Maintenance (cont.)

(5)

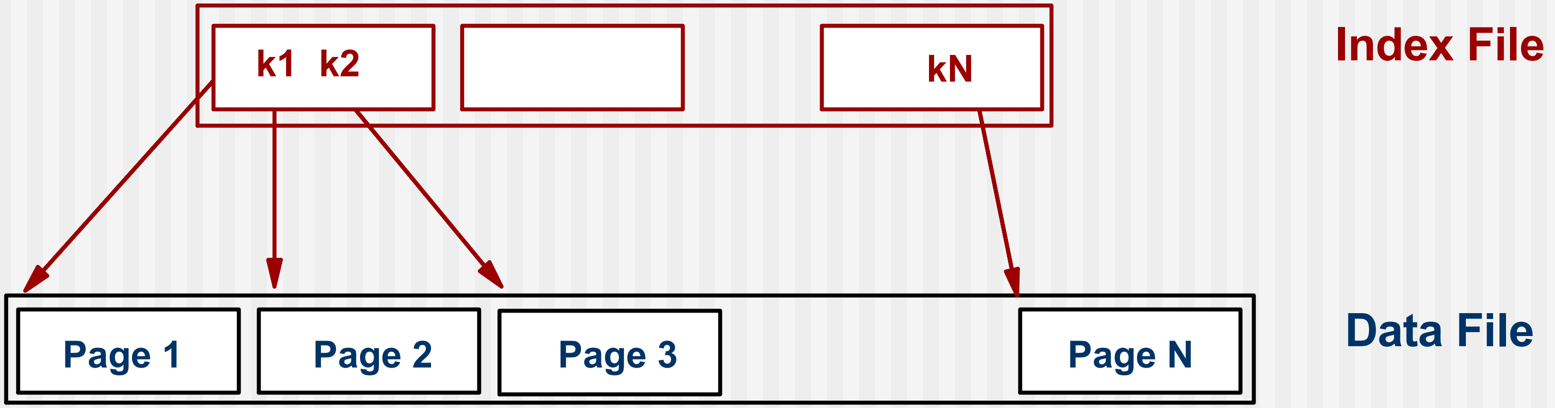


(6)



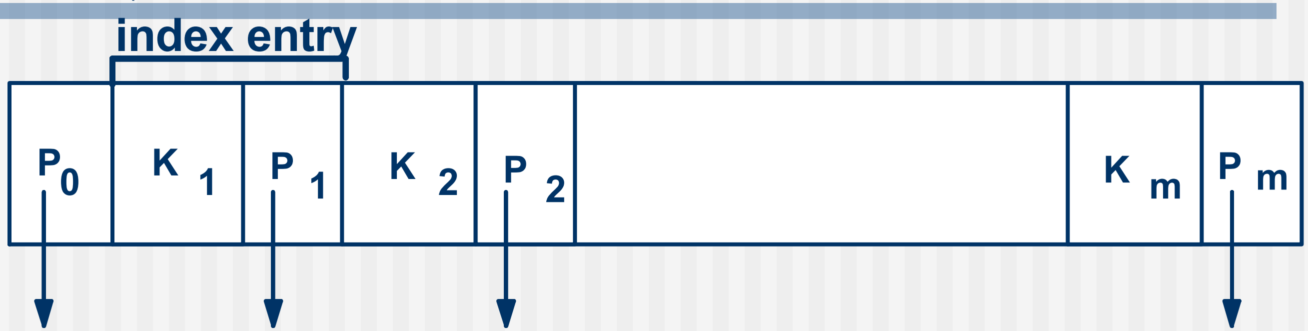
Range Searches

- *``Find all students with grade > 8.0''*
  - If data is in sorted file, do binary search to find first such student, then scan to find others.
  - Cost of binary search can be quite high.
- Simple idea: Create an `index' file.

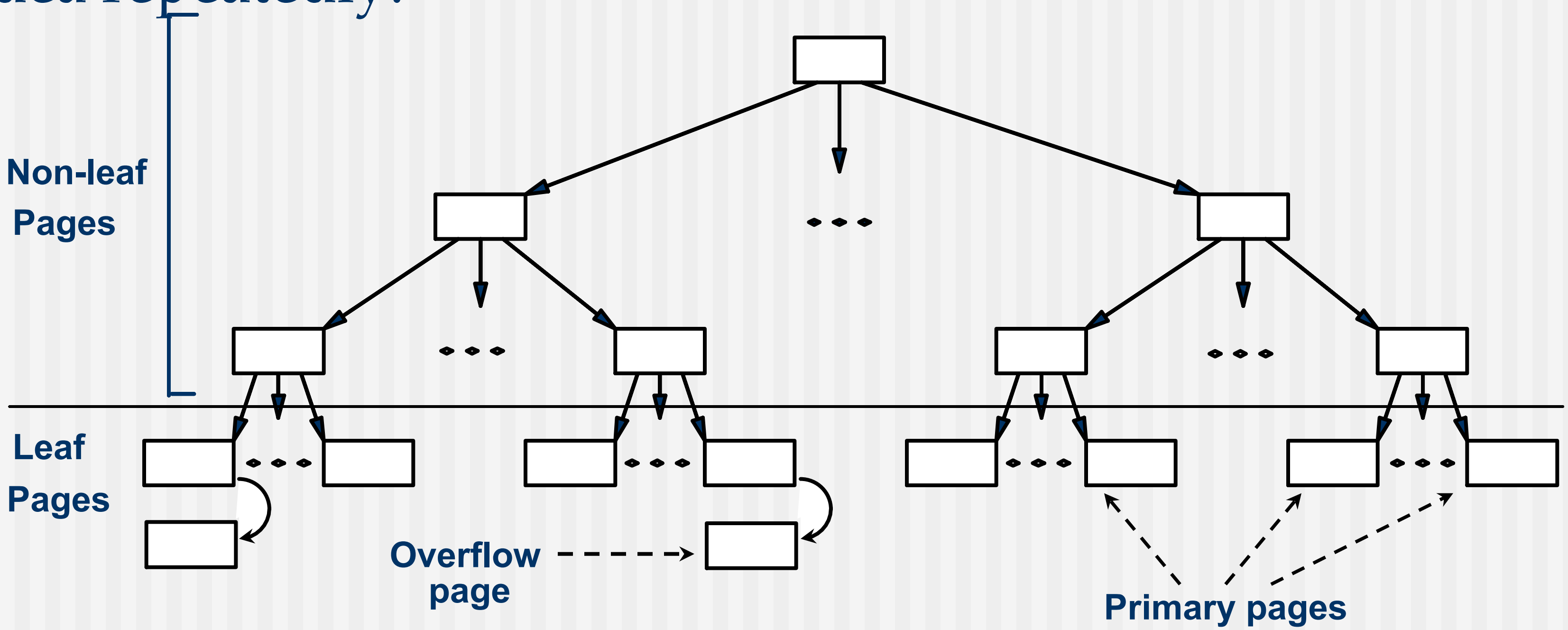


☛ *Can do binary search on (smaller) index file!*

ISAM

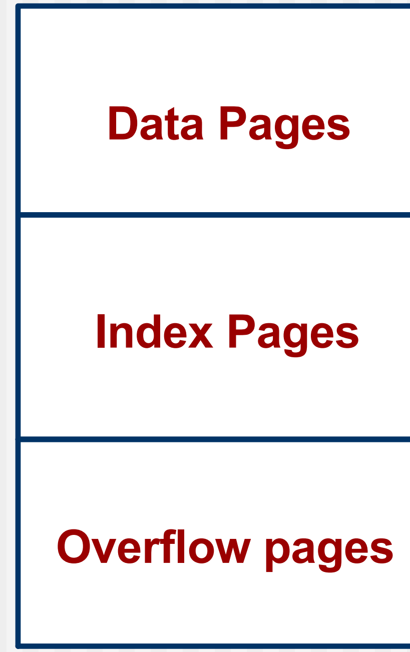


- Index file may still be quite large. But we can apply the idea repeatedly!



Comments on ISAM

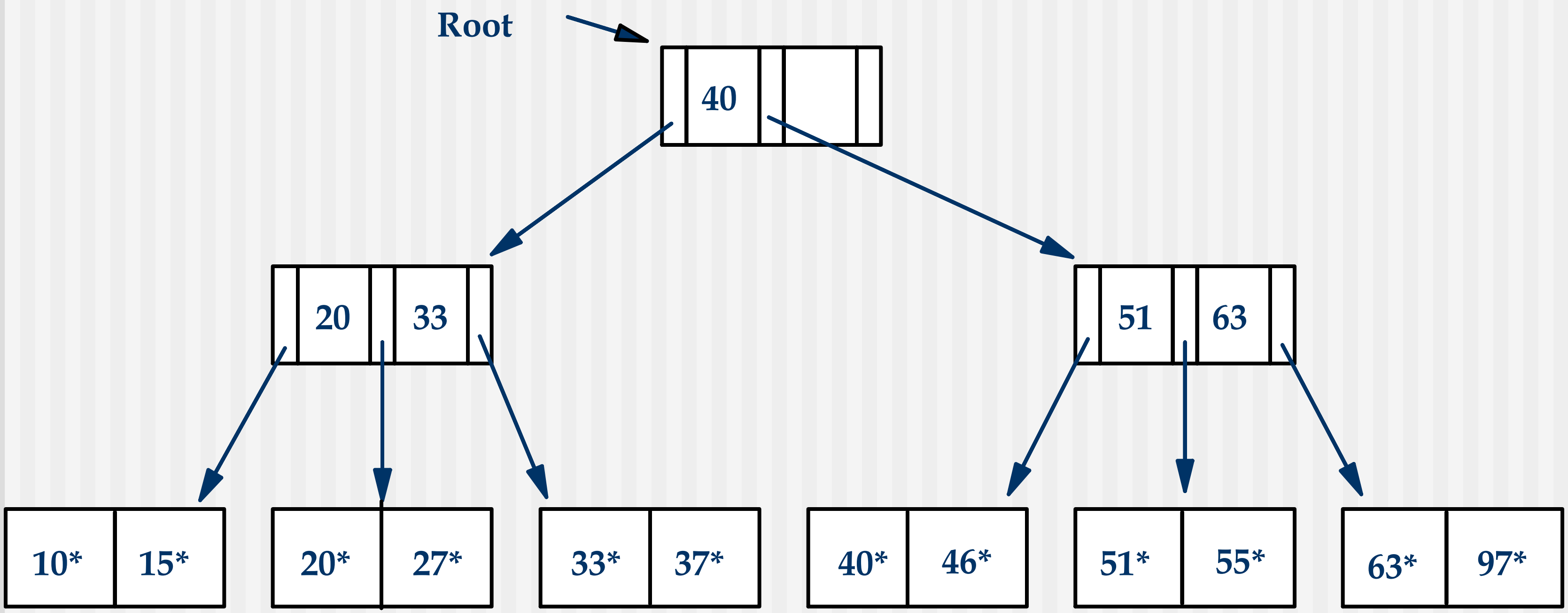
- *File creation*: Leaf (data) pages allocated sequentially, sorted by search key; then index pages allocated, then space for overflow pages.
- *Index entries*: <search key value, page id>; they `direct' search for *data entries*, which are in leaf pages.
- *Search*: Start at root; use key comparisons to go to leaf. Cost $\propto \log_F N$; F = number of entries/index pg, N = number of leaf pgs
- *Insert*: Find leaf data entry belongs to, and put it there.
- *Delete*: Find and remove from leaf; if empty overflow page, de-allocate.



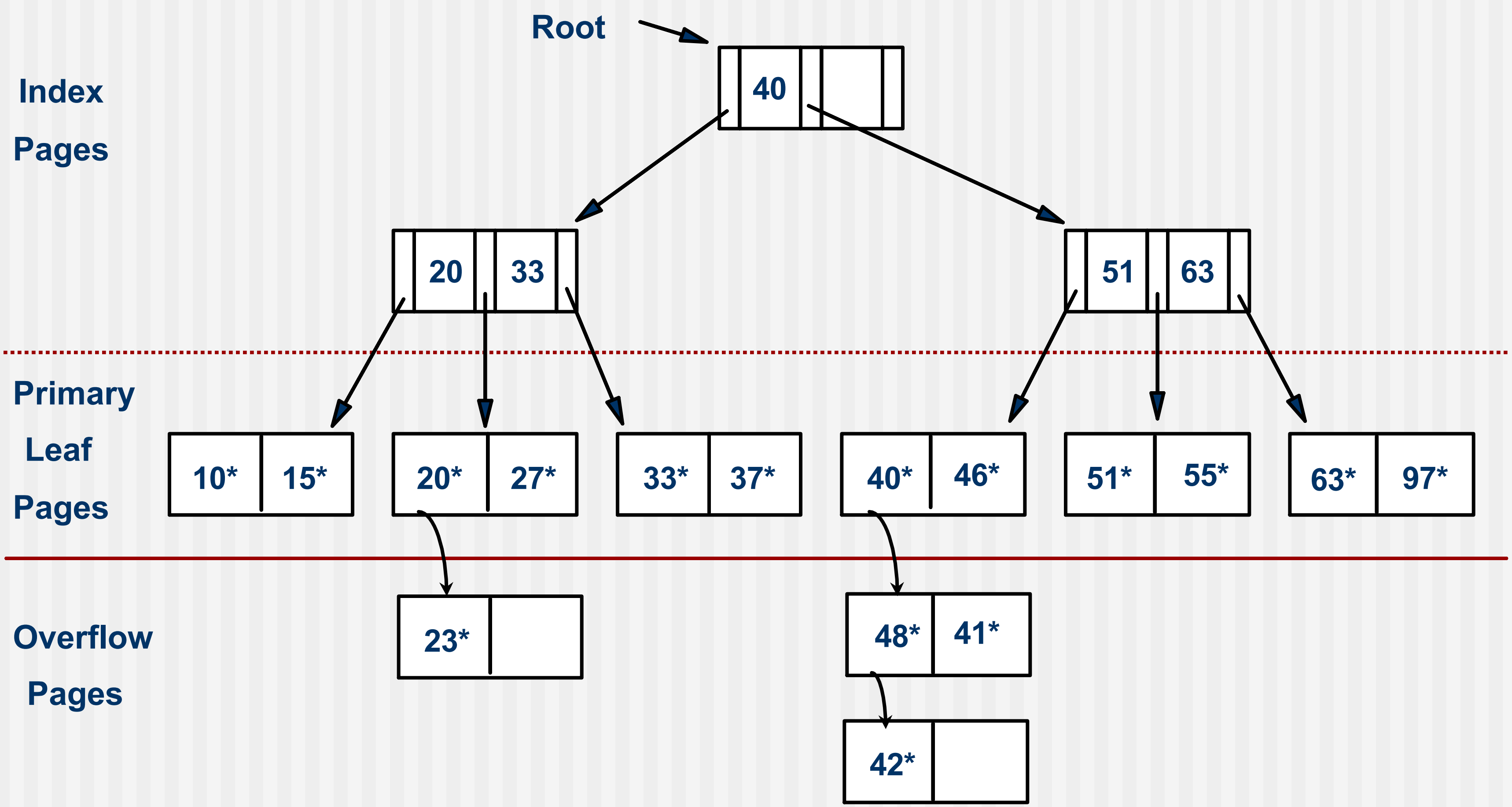
Static tree structure: *inserts/deletes affect only leaf pages*!

Example ISAM Tree

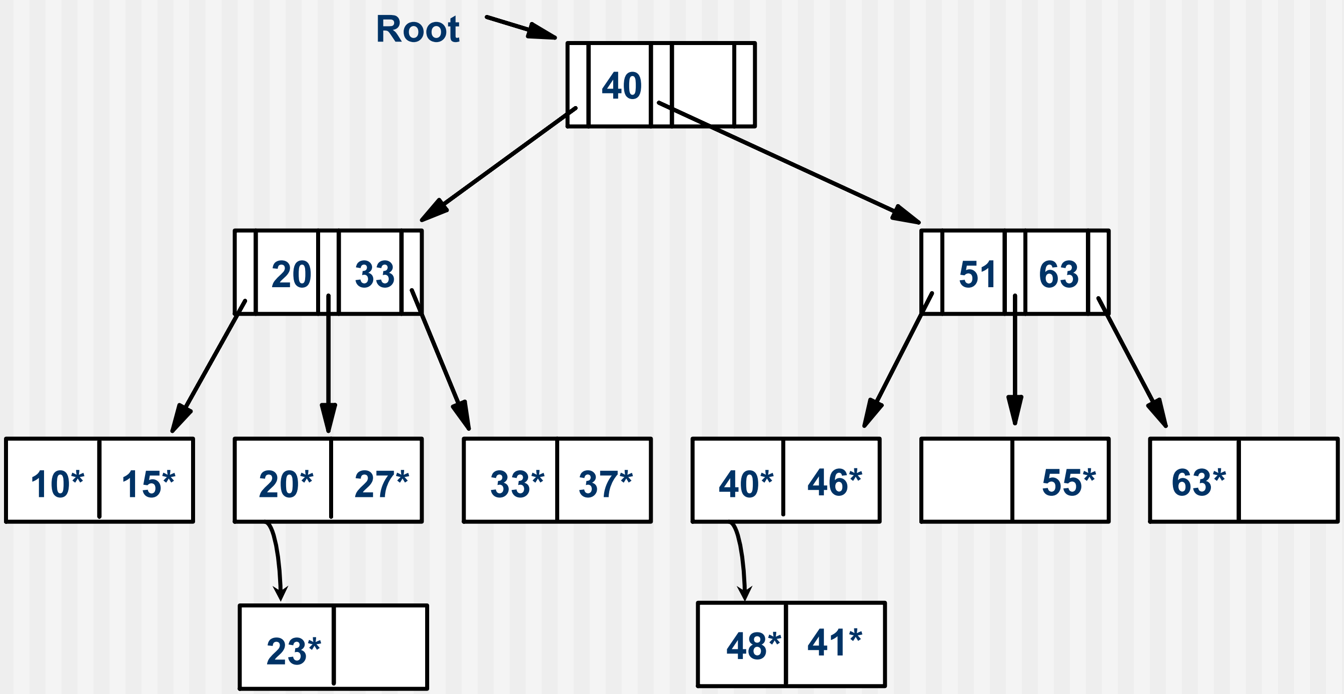
- Each node can hold 2 entries;



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



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



☛ *Note that 51* appears in index levels, but not in leaf!*

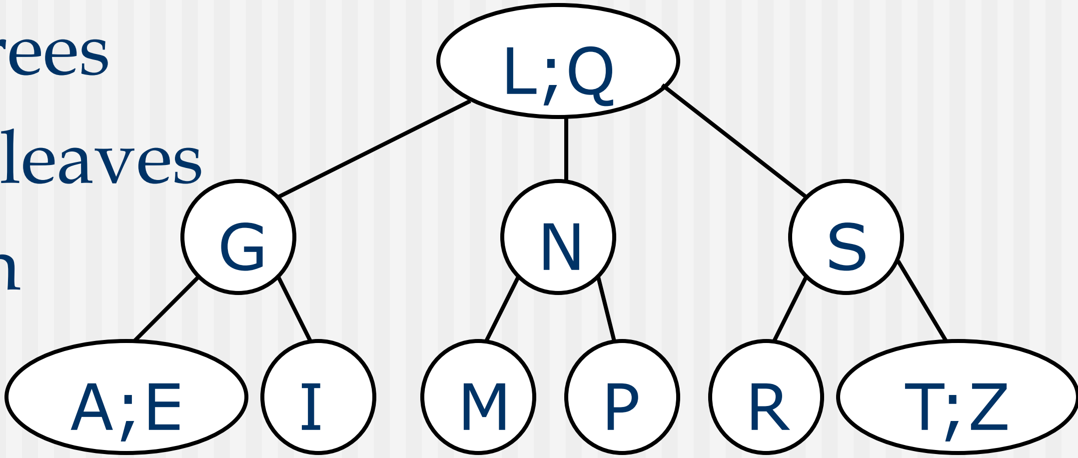
Advantages & Disadvantages of ISAM

- Can get out of balance - with lots of inserts & deletes (resulting in non-uniform search time)
- Overflow records may not be sorted (they could be - they're usually not)
- Faster insert & delete (no tree balancing…no I/Os for nodes in the tree)
- Better concurrent access (tree nodes are never locked)

Suitable only for files that aren't expected to change much.

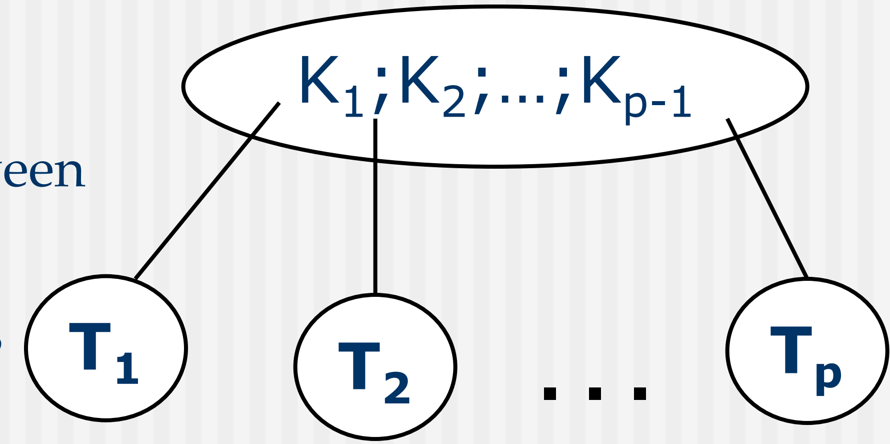
B – Trees Organization Files

- Most popular organization of indexes in DB
- "B" – stand for "balanced" or "broad"
- B–Tree – ordered tree; each node has several sub-trees
- A node contains keys and pointers to its sub-trees
- Paths from root to leaves have the same length



B-Tree Properties

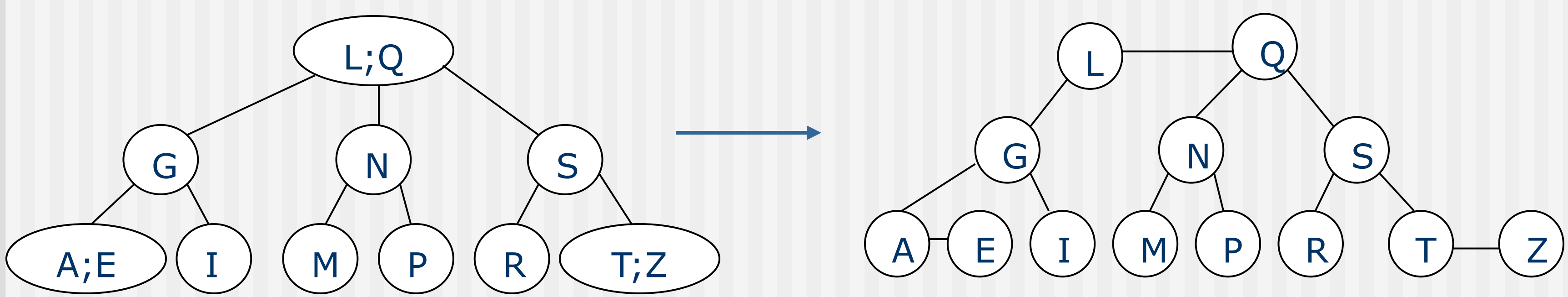
- B-Tree of order *m*:
  - If is not a leaf, the root has at least 2 sub-trees
  - Each internal node has at least [*m*/2] sub-trees (except root)
  - Each internal node has at most *m* sub-trees
  - All leaves are on the same level
  - A node with *p* sub-trees contains *p-1* ordered key values (K_1, K_2, … K_p)
    - T_1 contains key values <K_1
    - T_i contains key values between K_{i-1} and K_i
    - T_p contains key values > K_p



Memory Structure of B-Trees

K	Data	$\text{Pointer}_{\text{Left}}$	$\text{Pointer}_{\text{Right/H}}$

- As binary trees
  - *$Pointer_{Left}$*
    - refers first key of its left sub-tree in B-Tree
  - *$Pointer_{Right/H}$*
    - refers right neighbor key in B-Tree node
    - refers first key of its right sub-tree in B-Tree (if is last key value in a B-Tree's node)
  - Additional flag / signed right pointer



Memory Structure of B-Trees (cont.)

- Allocate memory to store m-1 key values per node
  - N – number of stored keys
  - K_i – key value, AD_i – address of record
  - $Point_i$ – refers a sub-tree

N	K_1	K_2	. . .	K_{m-1}	$Point_1$	. . .	$Point_m$

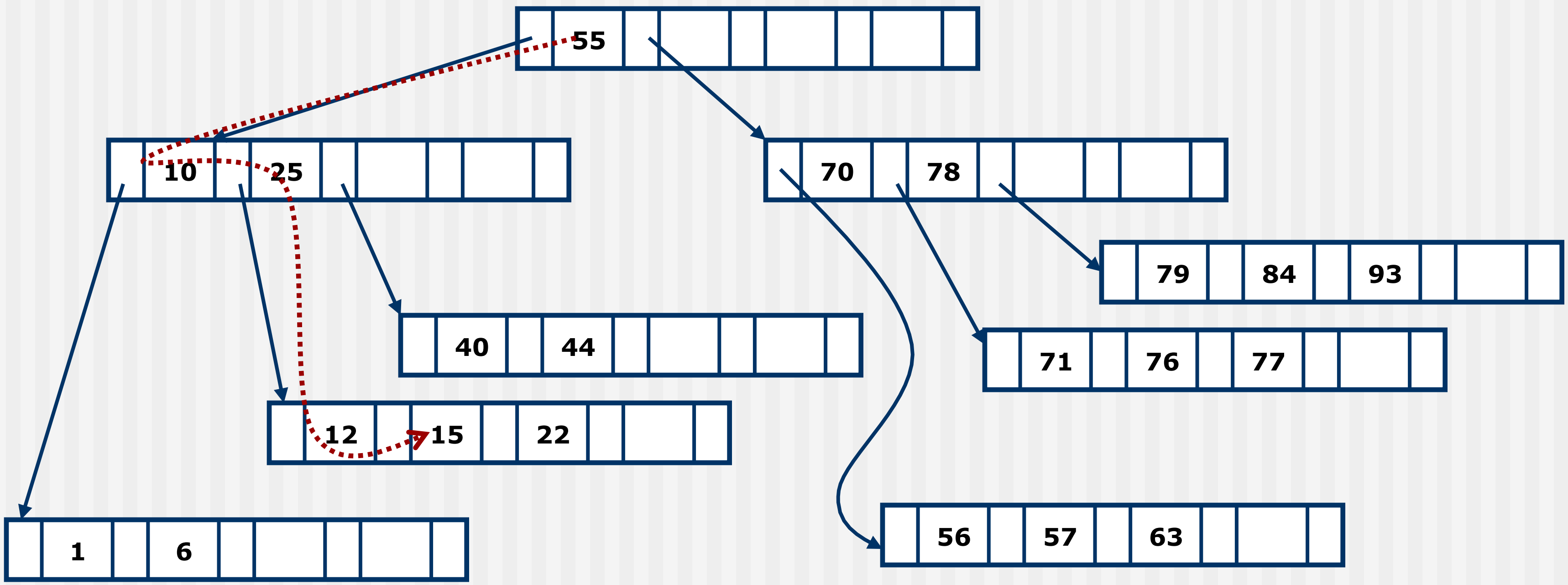
  - Use pointers memory space to store key values for leaves

K_m	K_{m+1}	. . .	K_t

    - $m/2 \le N \le m\text{-}1$ – for internal nodes
    - $m/2 \le N \le t$ – for terminal nodes
    - additional flag / signed N

Searching a B-Tree

- Option of following two paths at each node
  - Example: searching the '15' in a B-Tree



Inserting Records in B-Trees

- Insertion steps:
  - find proper location (node in which the key should be inserted)
  - insert new key
  - perform *balance* procedure if there is an *overflow*
- **Algorithm of the Insertion Procedure**

  1. Find the insertion point

  2. Insert the key

  3. If the node is full:

  A) Create a new node and put in the keys bigger than the median key

  B) Insert the median key into the parent node

  C) The right side of the key should reference the new node, the left side references the old one that has the smaller elements

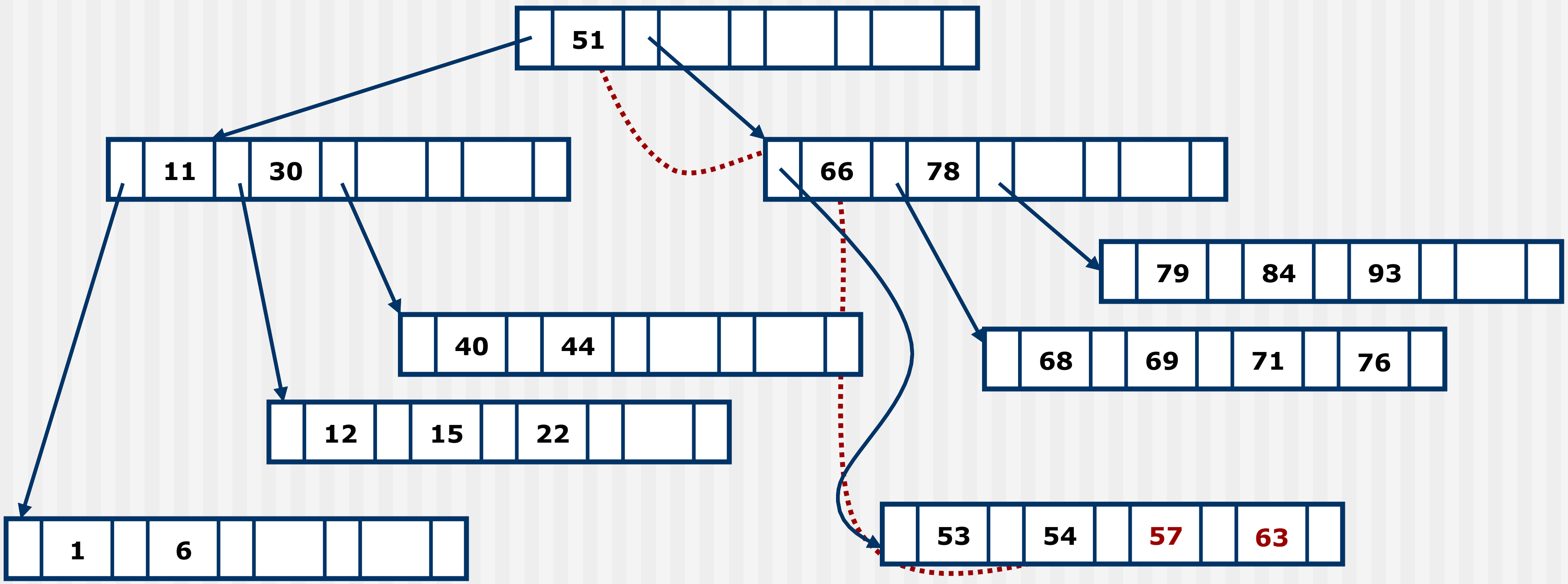
  4. If the parent node is full:

  A) If the parent node is a root then create a new root

  B) Repeat step 3 with the parent node

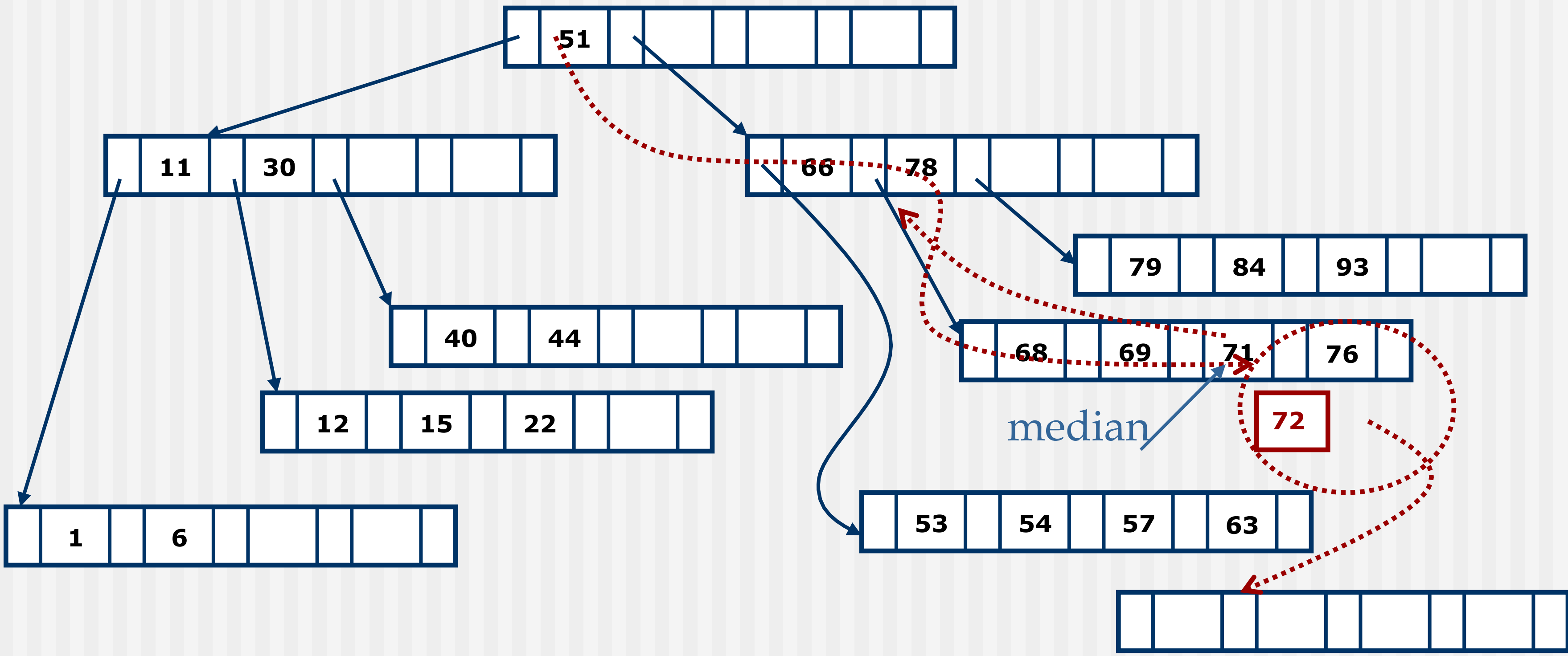
Inserting Records in B-Trees (cont.)

- Insert a record with key '57'



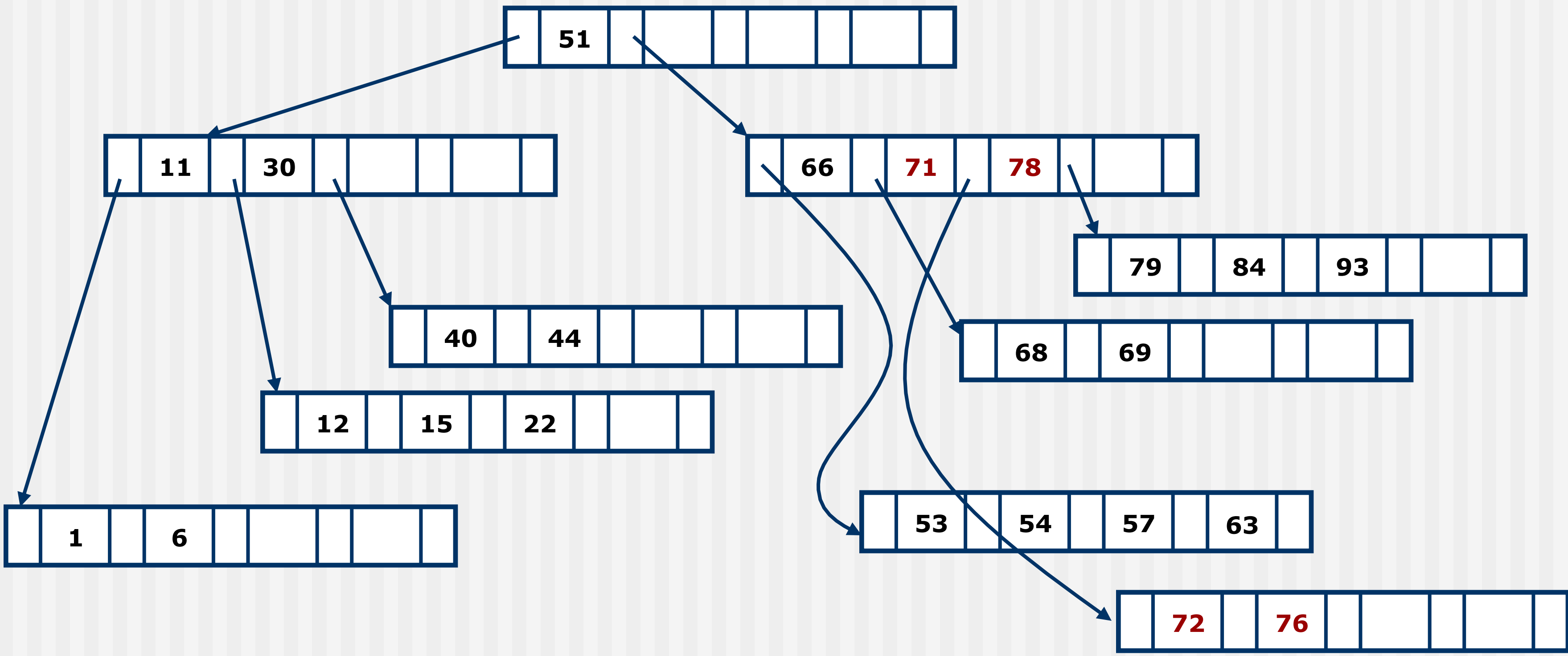
Inserting Records in B-Trees (cont.)

- … then insert a record with key '72'



Inserting Records in B-Trees (cont.)

- … then insert a record with key '72'



Deleting Records from B-Trees

- Deletion steps:
  - find node which contains the key should be deleted
  - if is internal node, *transfer* a key from leaves
  - if there is an *underflow*, perform *redistribution* or *concatenation*
- **Algorithm of the Deletion Procedure**

1. Search for the key to be removed

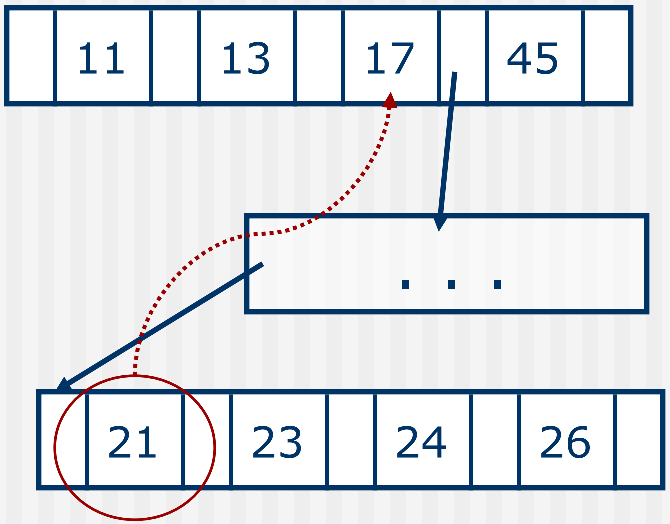
If the key is located into an internal node:

- replace it with its *bigger neighbor*

(i.e. with the *leftmost key*

of the *leftmost leaf*

of its *right tree*)



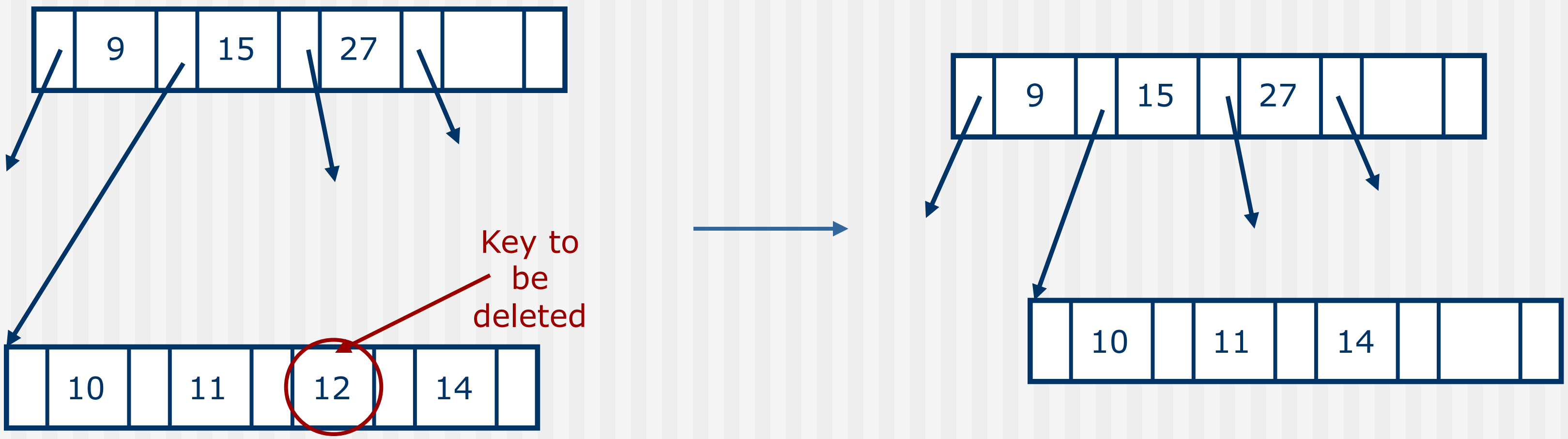
Deleting Records from B-Trees (cont.)

■ Algorithm of the Deletion Procedure (cont.)

2. Repeat this step until we fall in A) or B) cases

A) If the node which contains the key to be deleted is the root or the number of remaining keys is ≥ [m/2]:

- eliminate the key
- re-arrange the keys (and pointers) in the node
- finish the algorithm

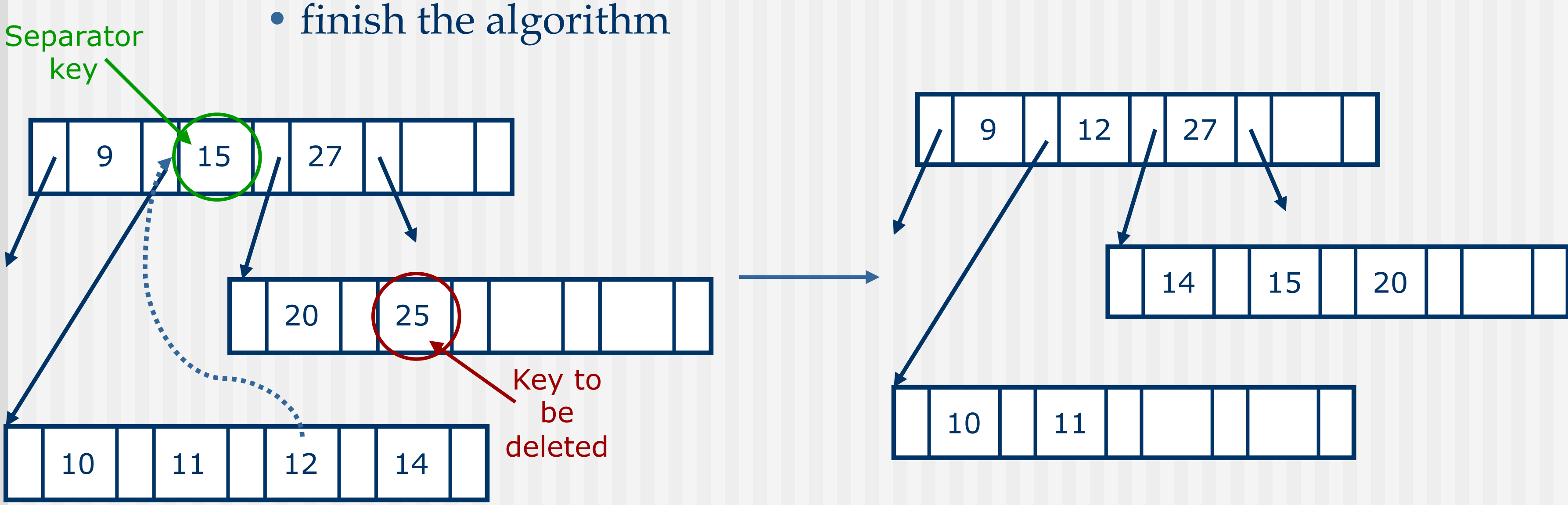


Deleting Records from B-Trees (cont.)

- **Algorithm of the Deletion Procedure (cont.)**

  B) If the number of remaining keys is < [m/2] and one of its neighbor nodes contains > [m/2] keys → *redistribution*

  - evenly divide the keys of both nodes + separator node from parent
  - chose the median key and put it in the parent
  - finish the algorithm

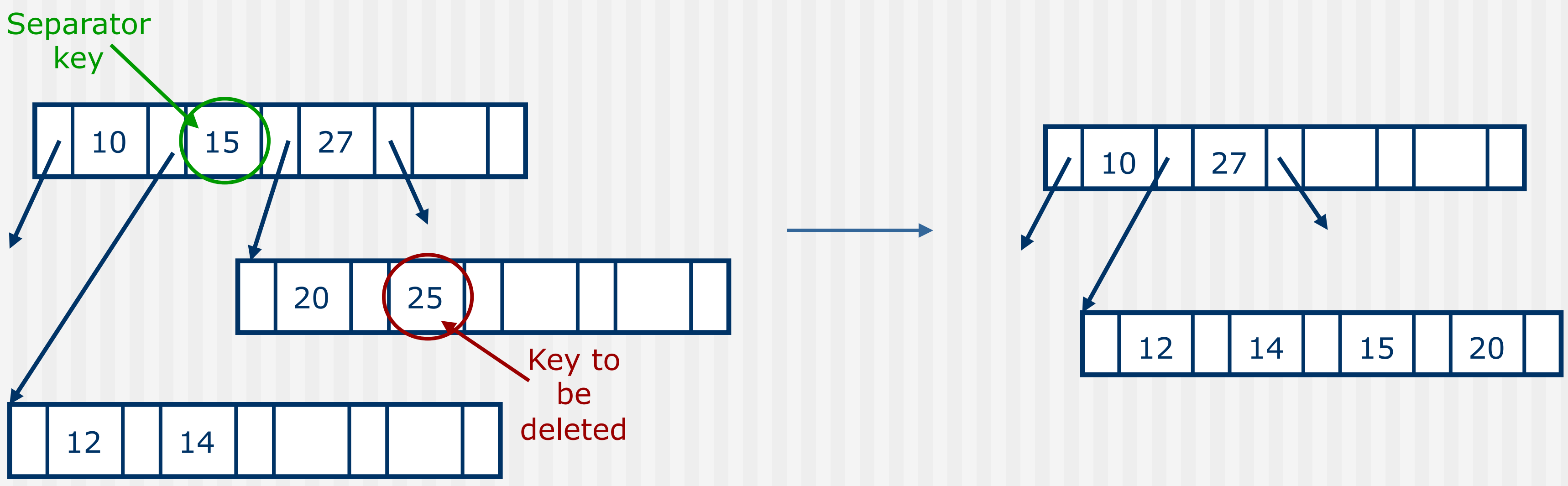


Deleting Records from B-Trees (cont.)

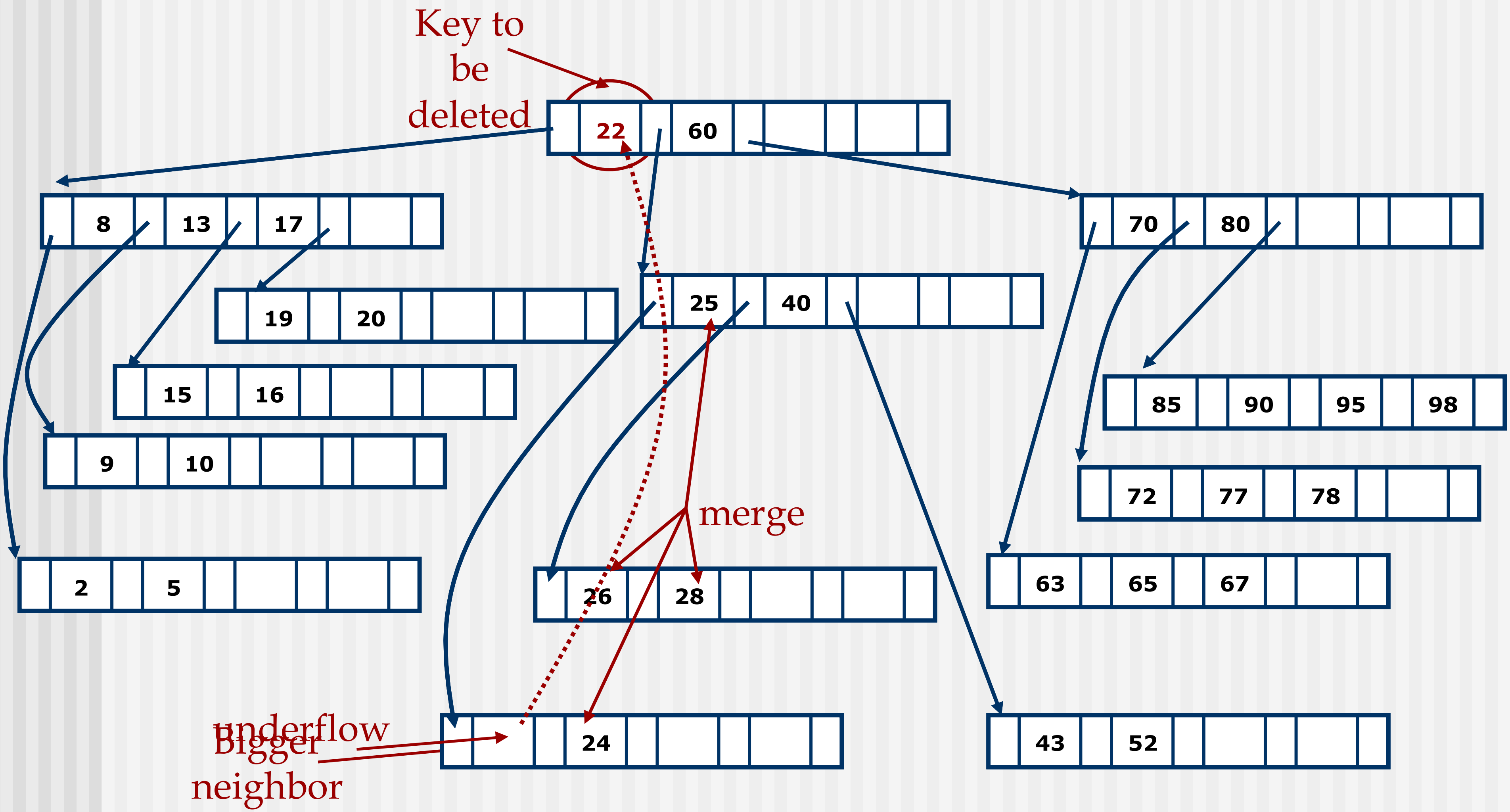
Algorithm of the Deletion Procedure (cont.)

C) If the sum of keys of node which contains the key to be deleted and of any neighbor < m → *concatenation*

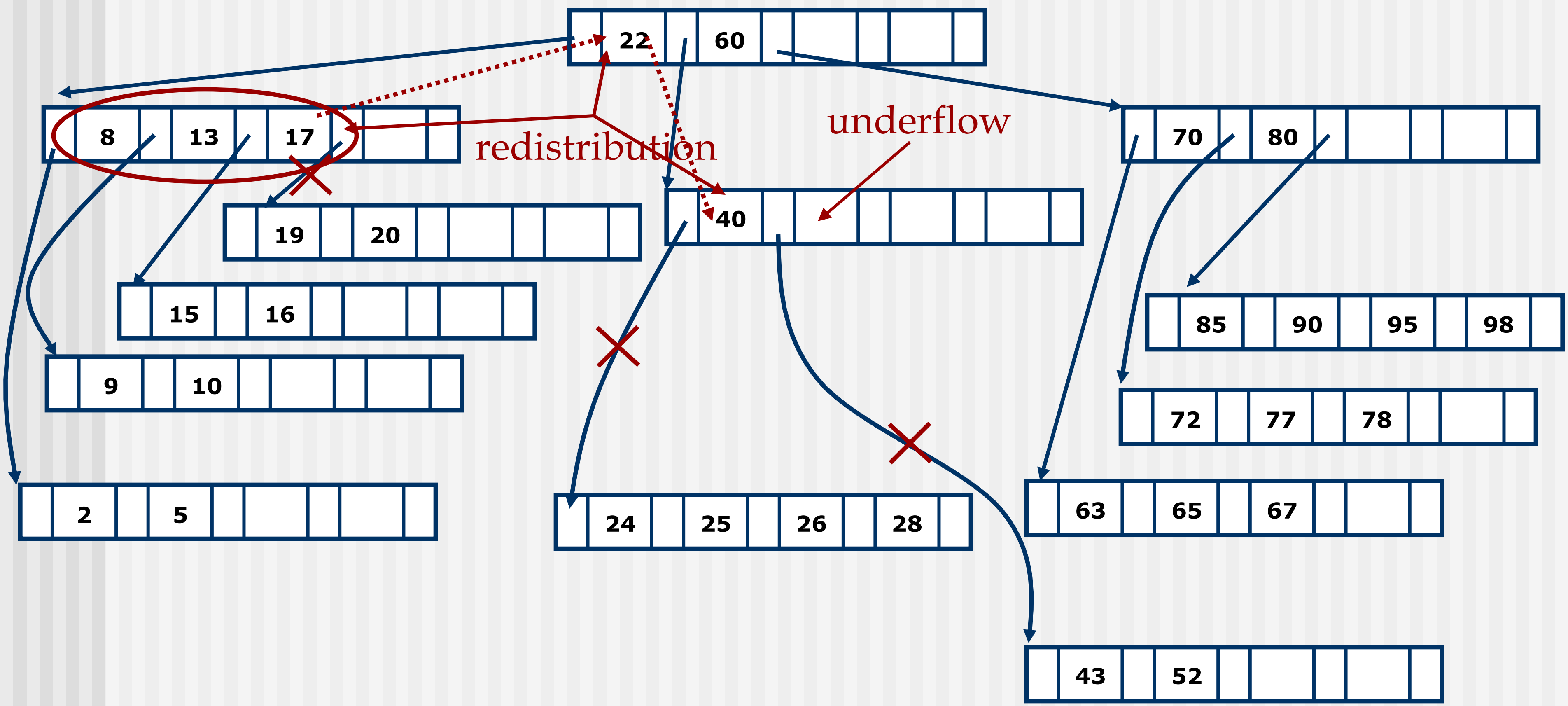
- merge the nodes + separator node from the parent
- repeat step **2.** for the parent node
- if parent node is the root and has no keys → current node become the root



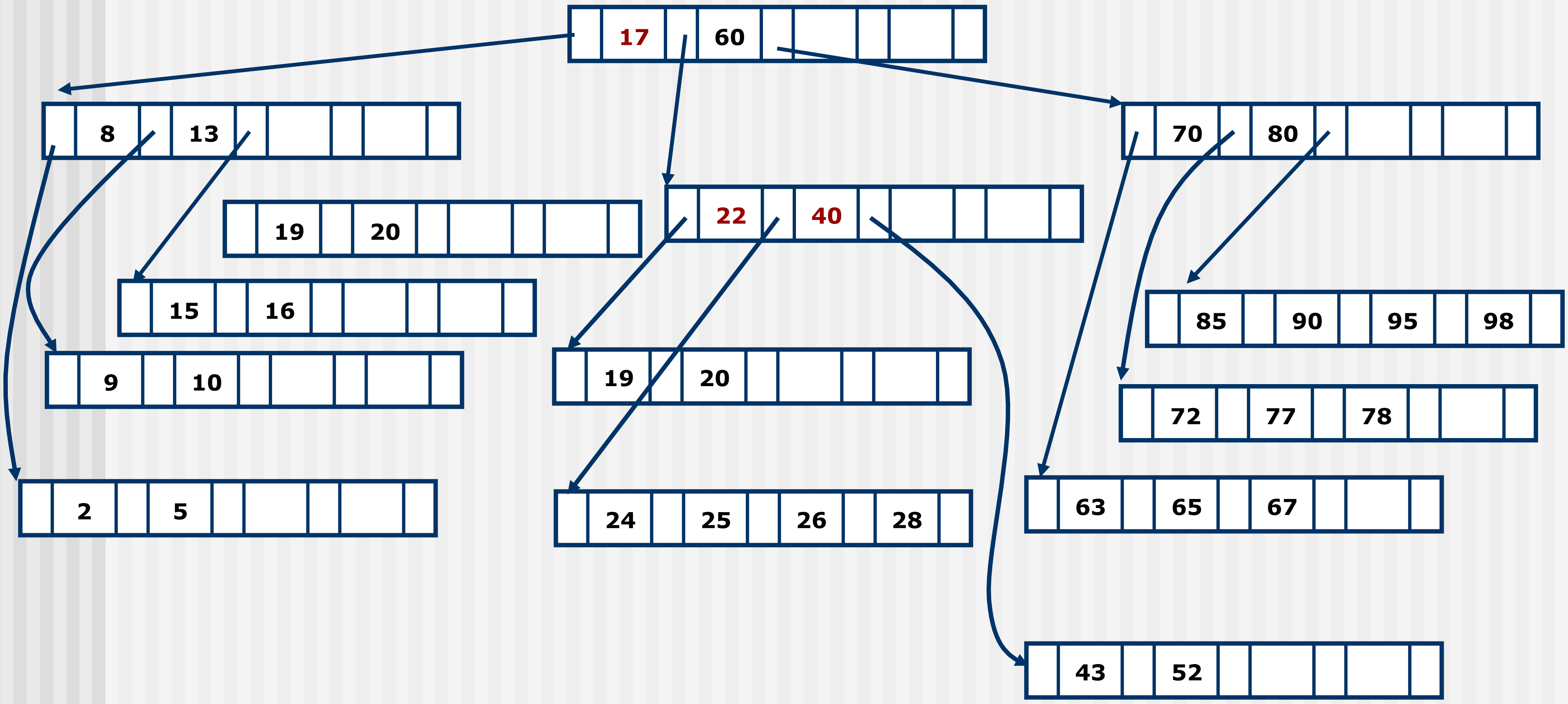
Deleting Records from B-Trees (cont.)



Deleting Records from B-Trees (cont.)

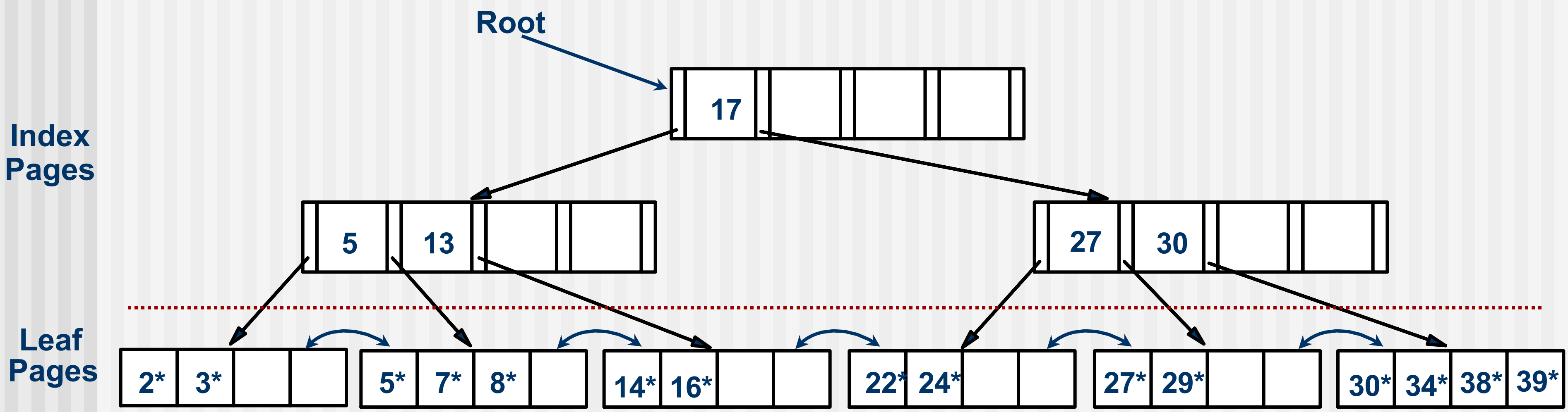


Deleting Records from B-Trees (cont.)



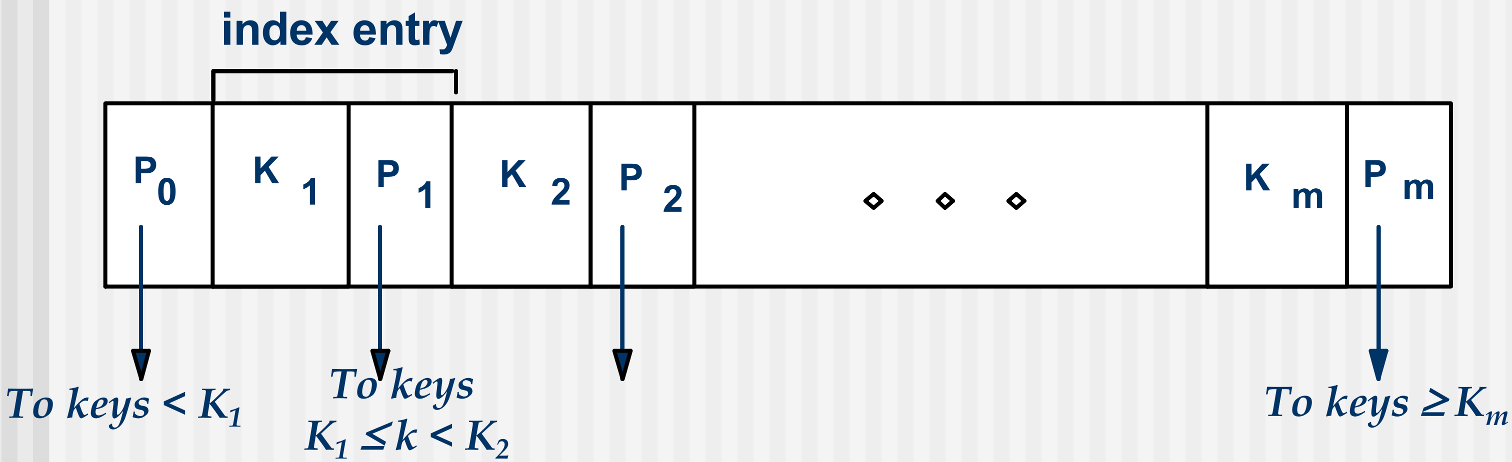
B+-Trees

- Combination between B-Tree and ISAM:
  - Search begins at root, and key comparisons direct it to a leaf
  - In a B+-Tree all pointers to data records exist only at the leaf-level nodes
- A B+-Tree can have less levels (or higher capacity of search values) than the corresponding B-tree

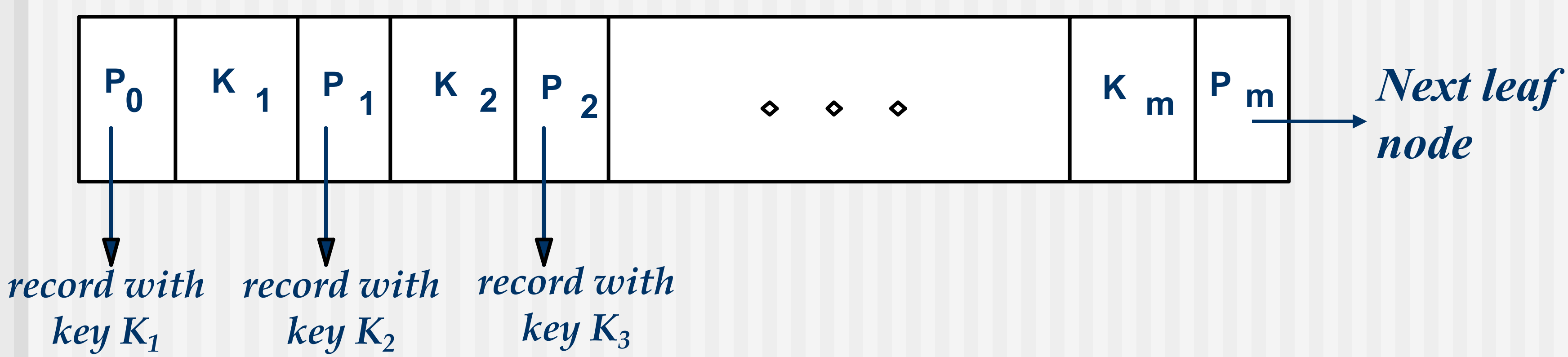


Node structure

Non-leaf nodes



Leaf nodes



B+-Trees in Practice

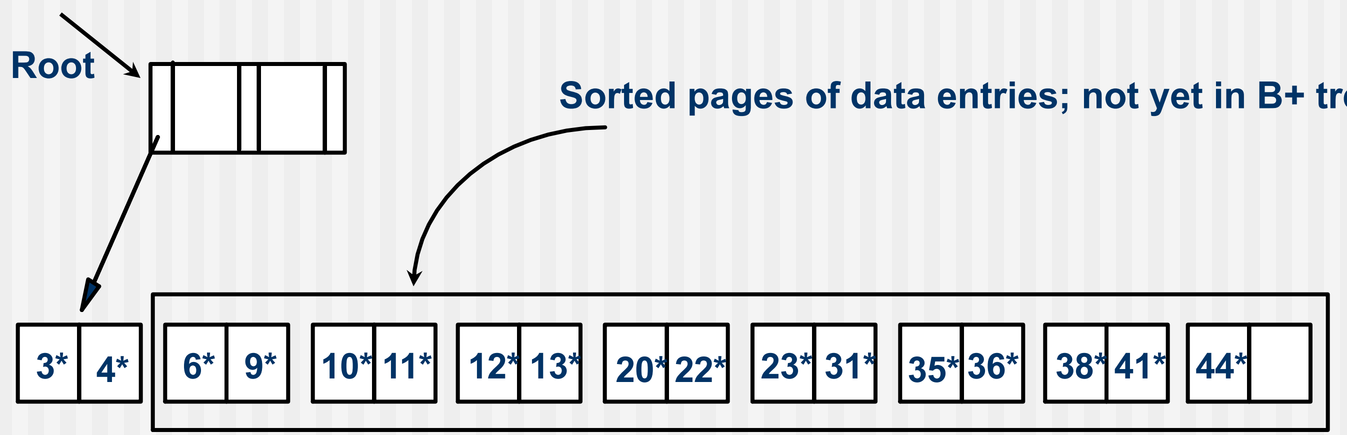
- Typical order: 200. Typical fill-factor: 67%.
  - Average fan-out (no. of entries/index pg) = 133
- Typical capacities:
  - Height 4: 133^4 = 312,900,700 records
  - Height 3: 133^3 = 2,352,637 records
- Can often hold top levels in buffer pool:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes

Advantages & Disadvantages of B+ Trees

- Index stays balanced…therefore uniform search time
- Rarely more than 3 - 5 levels…the system often maintains the top levels in memory … thus you can search for a record in 2 or 3 I/Os.
- Occupancy normally about 67% (thus 150% as much space as you need for the data records)
- Most widely used index in database management systems because of its versatility. One of the most optimized components of a DBMS.
- B+ trees can be used for a clustered, sparse index (if the data is sorted) or for an un-clustered, dense index (if not).

Bulk Loading of a B+-Tree

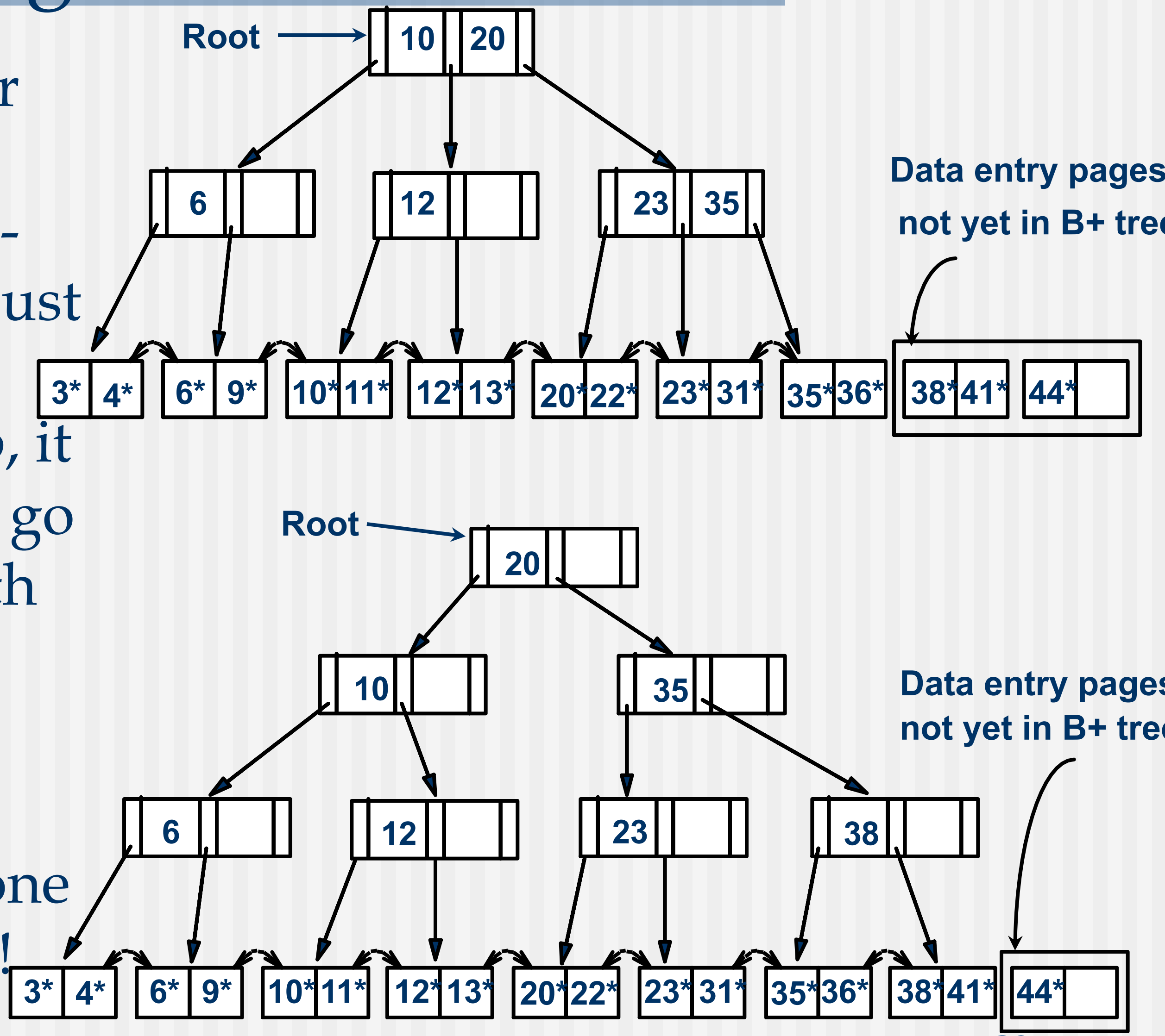
- If we have a large collection of records, and we want to create a B+-Tree on some field, doing so by repeatedly inserting records is very slow.
- *Bulk Loading* can be done much more efficiently.
- *Initialization*: Sort all data entries, insert pointer to first (leaf) page in a new (root) page.



Bulk Loading

- Index entries for leaf pages always entered into right-most index page just above leaf level. When this fills up, it splits. (Split may go up right-most path to the root.)

Much faster than repeated inserts, especially when one considers locking!



Summary of Bulk Loading

- Option 1: multiple inserts.
  - Slow.
  - Does not give sequential storage of leaves.
- Option 2: *Bulk Loading*
  - Has advantages for concurrency control.
  - Fewer I/Os during build.
  - Leaves will be stored sequentially (and linked, of course).
  - Can control "fill factor" on pages.

Prefix B+-Trees (Key Compression)

- Important to increase fan-out
- Key values in index entries only `direct traffic'; can often compress them.
  - E.g., If we have adjacent index entries with search key values *Dan Yogurt*, *David Smith* and *Demy Moore*, we can abbreviate *David Smith* to *Dav*. (The other keys can be compressed too ...)
    - Is this correct? Not quite! What if there is a data entry *Davey Jones*? (Can only compress *David Smith* to *Davi*)
    - In general, while compressing, must leave each index entry greater than every key value (in any sub-tree) to its left.
- Insert/delete must be modified accordingly.

B+-Tree order in practice

- *Order* concept replaced by physical space criterion in practice (`*at least half-full*').
  - Index pages can typically hold many more entries than leaf pages.
  - Variable sized records and search keys mean different nodes will contain different numbers of entries.
  - Even with fixed length fields, multiple records with the same search key value (*duplicates*) can lead to variable-sized data entries (if we use Alternative (3)).

Summary

- Tree-structured indexes are ideal for range-searches, also good for equality searches.
- ISAM is a static structure.
  - Only leaf pages modified; overflow pages needed.
  - Overflow chains can degrade performance unless size of data set and data distribution stay constant.
- B+ tree is a dynamic structure.
  - Inserts/deletes leave tree height-balanced; $\log_F N$ cost.
  - High fanout (**F**) means depth rarely more than 3 or 4.
  - Almost always better than maintaining a sorted file.

Summary (cont.)

  - Typically, 67% occupancy on average.
  - Usually preferable to ISAM, modulo *locking* considerations; adjusts to growth gracefully.
  - If data entries are data records, splits can change rids!
- Key compression increases fan-out, reduces height.
- Bulk loading can be much faster than repeated inserts for creating a B+ tree on a large data set.
- Most widely used index in database management systems because of its versatility. One of the most optimized components of a DBMS.