Databases

Lecture 9

The Physical Structure of Databases (II)

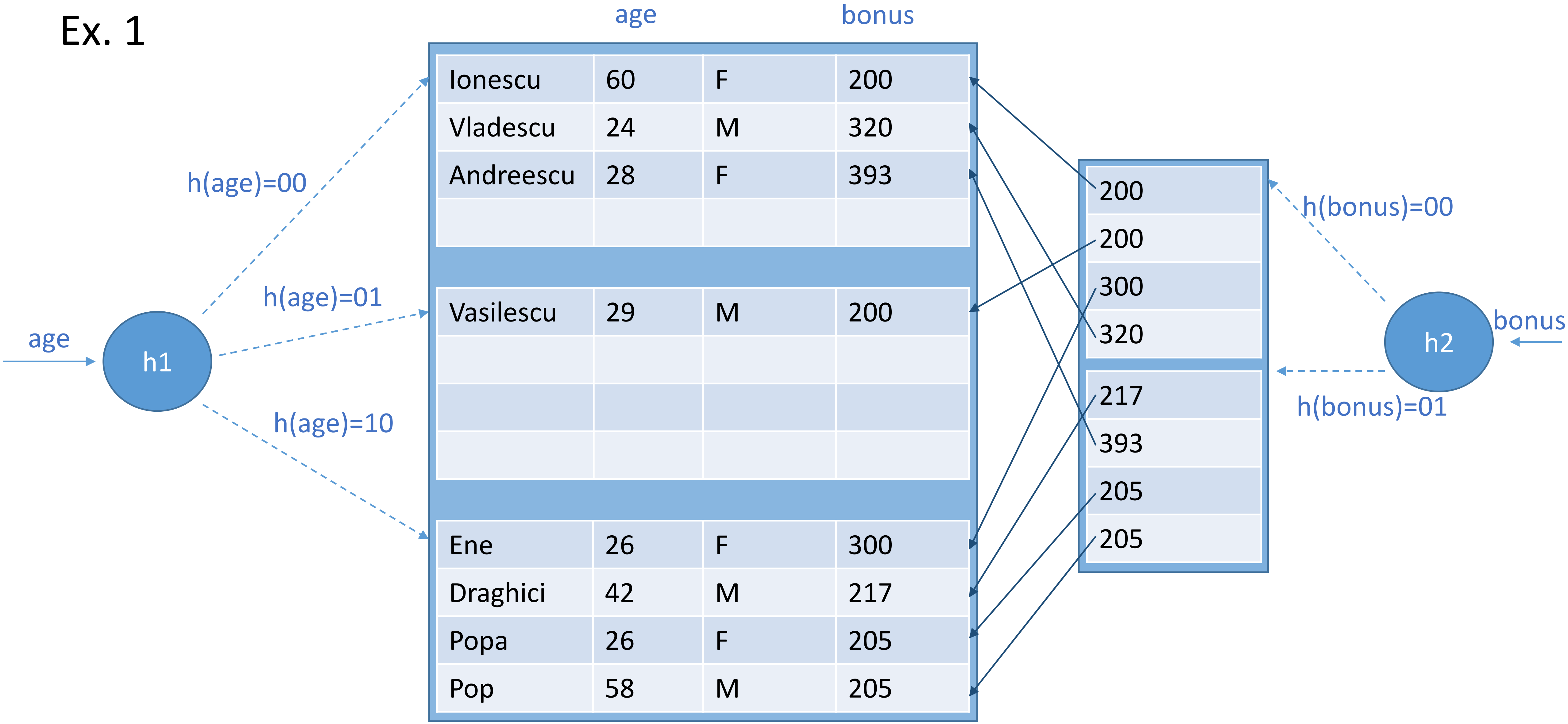
- Indexes. Tree-Structured Indexing -

Sabina S. CS

Indexes - Data Entries

h : $72_{(10)} = 0100\ 1000_{(2)}$

Ex. 1



Indexes - Data Entries

Ex. 1

- file with Employee records hashed on *age*
  - record <Ionescu, 60, F, 200>:
    - apply hash function to *age:* convert 60 to its binary representation, take the 2 least significant bits as the bucket identifier for the record
- index file that uses alternative 1 (data entries are the actual data records), search key *age*
- index that uses alternative 2 (data entries have the form <search key, rid>), search key *bonus*
- both indexes use hashing to locate data entries

Indexes - Data Entries

Ex. 2

	Name	Score	Age
rid_1	Popescu	3	44
rid_2	Ionescu	95	80
rid_3	Vladescu	3	45
…		…	
rid_p	Xulescu	90	14

search key

rid_1	3
rid_2	95
rid_3	3
…	
rid_p	90

3	rid_1, rid_3 …
95	rid_2 …
90	rid_p …
	…

…	
3	rid_1, rid_3 …
90	rid_p …
95	rid_2 …
…	

data entry

index file

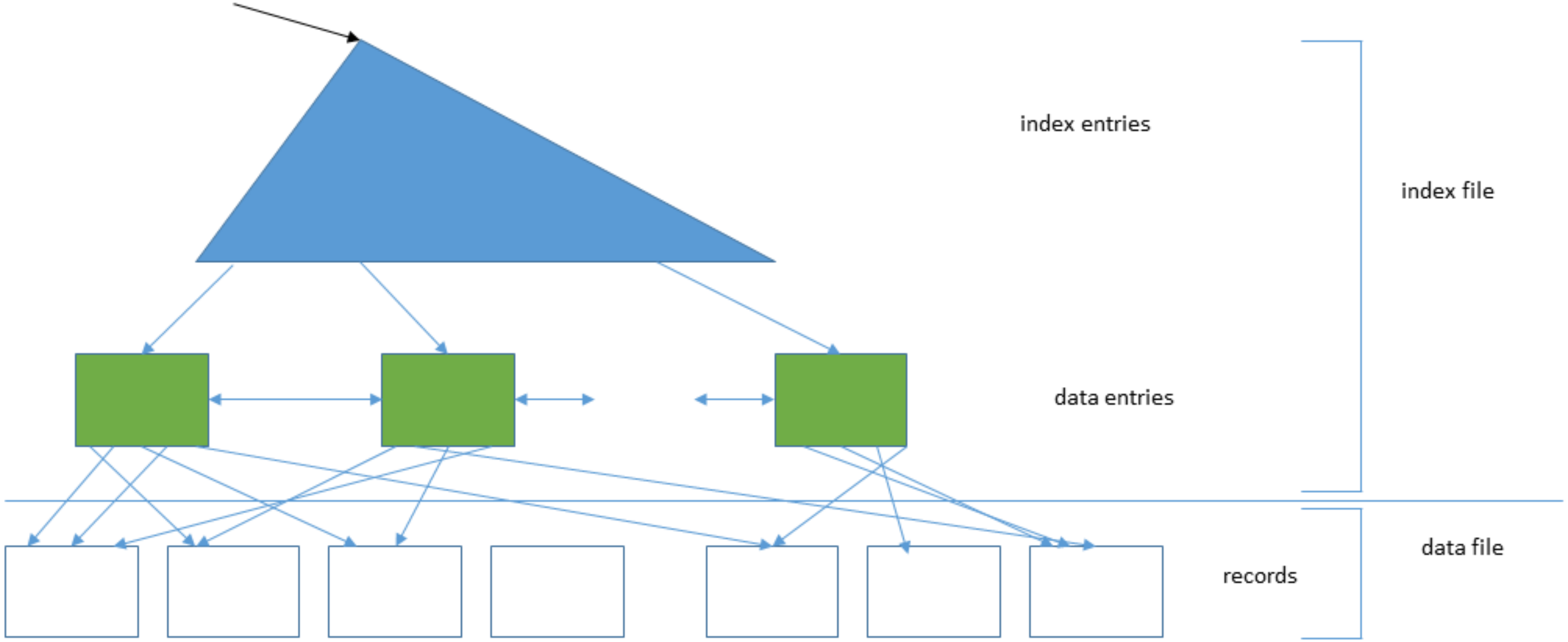
Clustered / Unclustered Indexes

A1 is clustered by definition

- clustered index: the order of the data records is close to / the same as the order of the data entries

A2, A3 → clustered only if the index is ordered on the search key

- unclustered index: index that is not clustered



unclustered index, alternative 2

index entries guide the search for data entries

Clustered / Unclustered Indexes

- index that uses alternative 1 - clustered (by definition, since the data entries are the actual data records)
- indexes using alternatives 2 / 3 are clustered only if the data records are ordered on the search key
- in practice:
  - expensive to maintain the sort order for files, so they are rarely kept sorted
  - a clustered index is an index that uses alternative 1 for data entries
  - an index that uses alternative 2 or 3 for data entries is unclustered
- on a collection of records:
  - there can be at most 1 clustered index
  - and several unclustered indexes

Clustered / Unclustered Indexes

- range search query (e.g., *where age between 20 and 30*)
  - cost of using an unclustered index
    - each data entry that meets the condition in the query could contain a rid pointing to a distinct page w.c.
    - the number of I/O operations could be equal to the number of data entries that satisfy the query's condition

Primary / Secondary Indexes

- primary index
  - the search key includes the primary key
- secondary index
  - index that is not primary
- unique index
  - the search key contains a candidate key

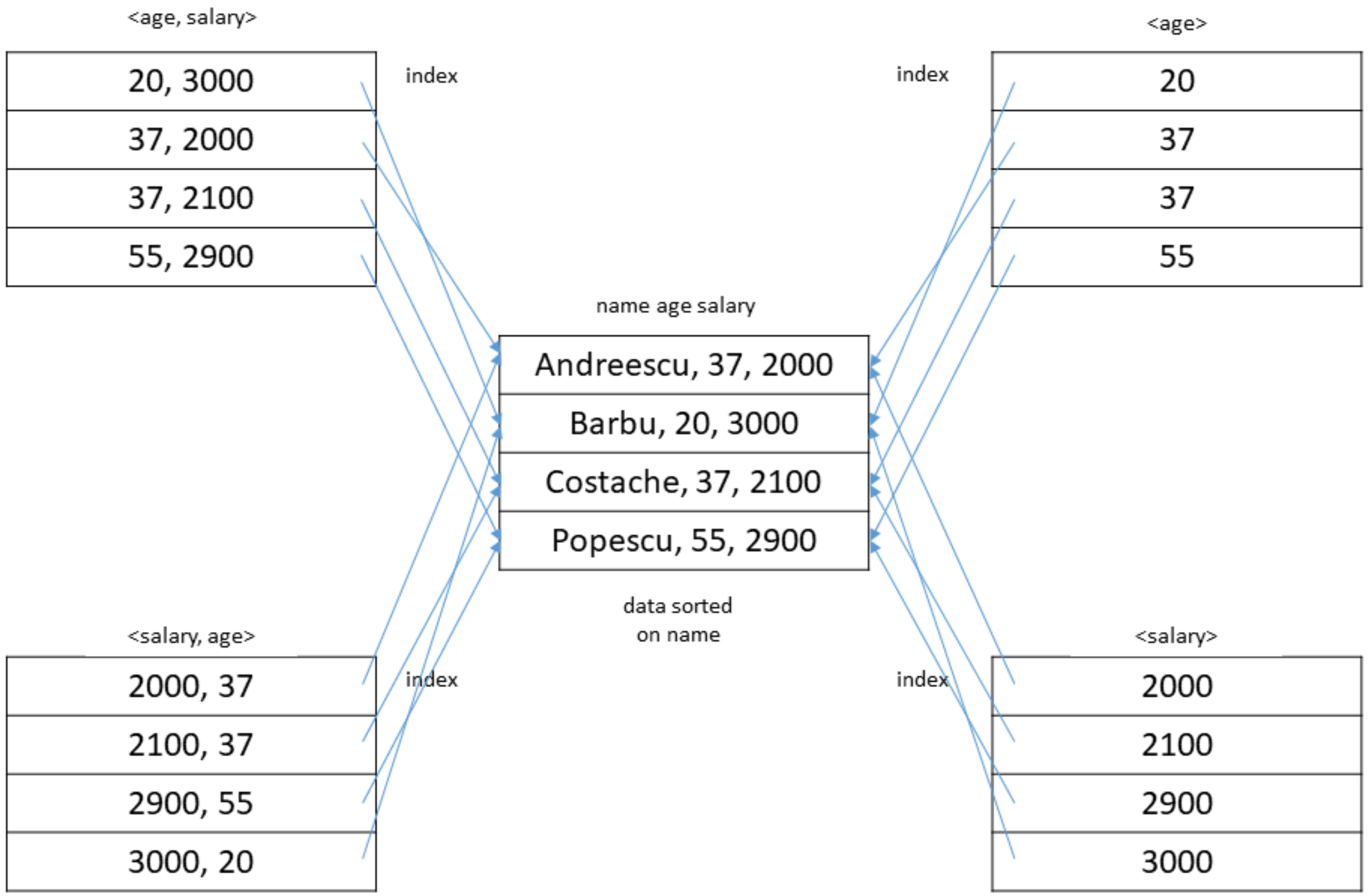
- duplicates
  - data entries with the same search key value

- primary indexes, unique indexes cannot contain duplicates
- secondary indexes can contain duplicates

Composite Search Keys

- composite (concatenated) search key - search key that contains several fields
  - examples

not good for "give me all empolies with salary over x



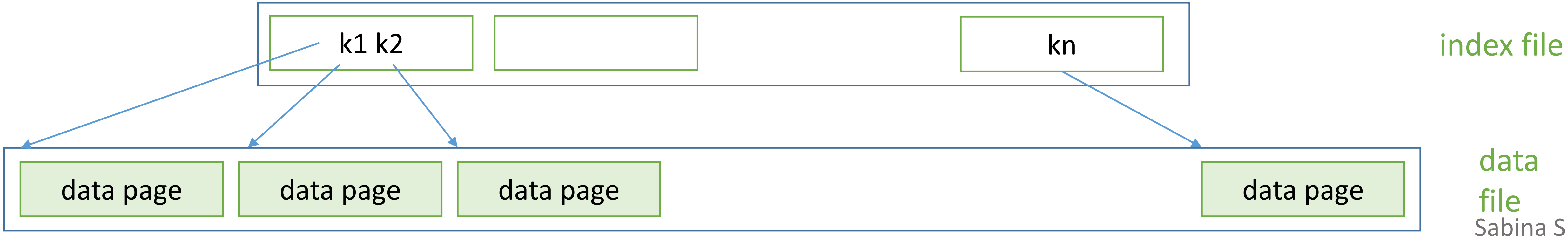
Indexed Sequential Access Method (ISAM)

* Example. Q: Find all phones with *rating* > 9 - range selection query

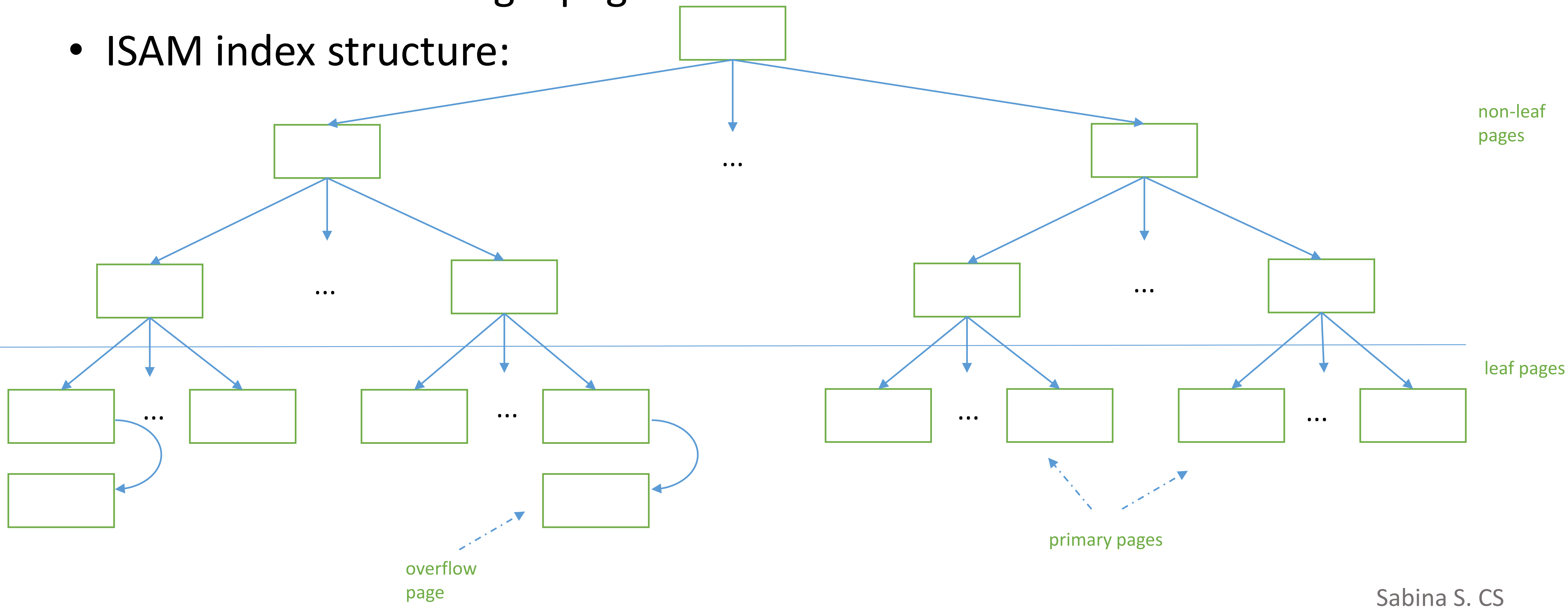
- data stored in sorted file (records sorted by *rating*) - identify 1st phone using binary search; scan file to get the rest of the phones
- large file => potentially expensive binary search
- create another file with records of the form *<1st key on the page, pointer to the page>*, sorted on the key (*rating* in the example)

- page:

- one-level index structure:



- size of index file - much smaller than size of data file => faster binary search
- index file can still be quite large => further optimization: auxiliary structures are created recursively on top of previously created ones, until one such structure fits on a single page
- ISAM index structure:



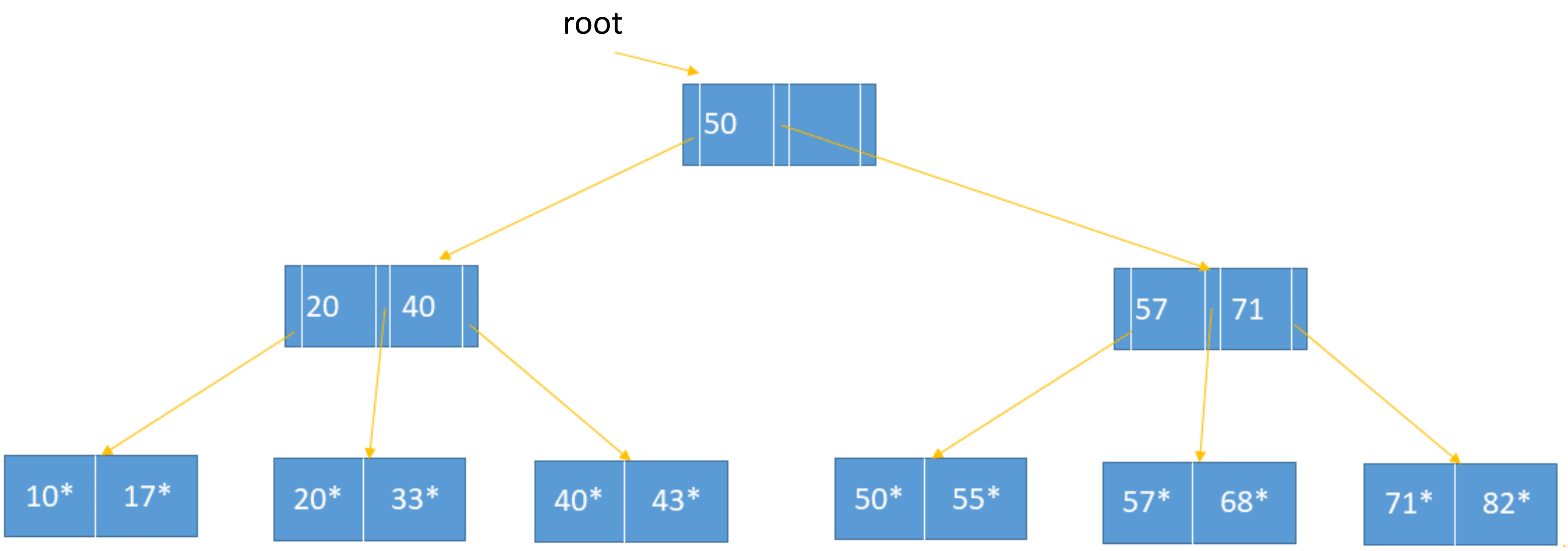
- file creation
  - allocate leaf pages - sequentially allocated, sorted on the key
  - allocate non-leaf pages
  - inserts that exceed a page's capacity – allocate overflow pages
- search
  - starts at the root
  - comparisons with the key to find the leaf page
  - cost – disk I/O

data pages
index pages
overflow pages

- insertion
  - find the corresponding leaf page, add the entry
  - if there is no space on the page, add an overflow page
- deletion
  - find the leaf page that contains the entry, remove the entry
  - if an overflow page is emptied, it can be eliminated
- inserts / deletes
  - only leaf pages are affected (static structure)

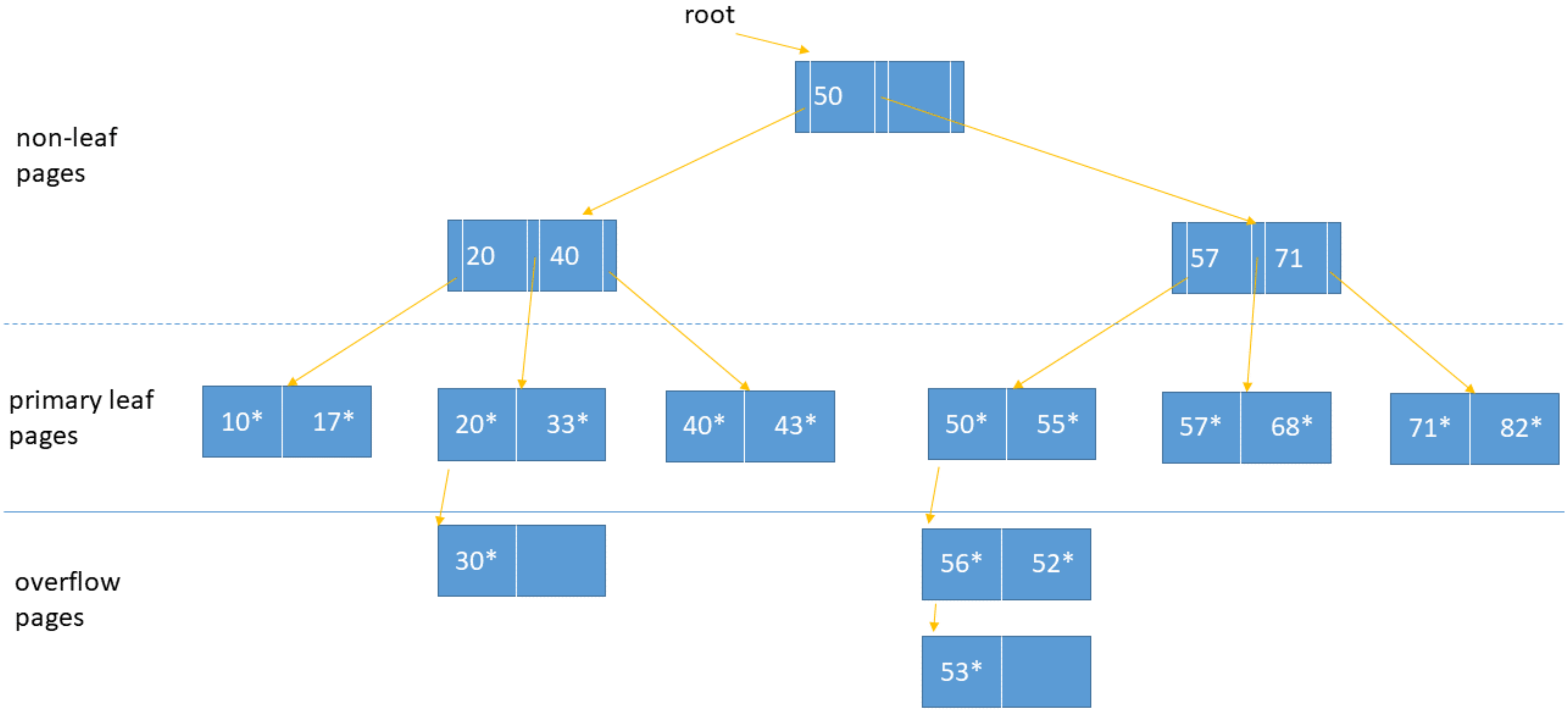
* Example - ISAM tree

- leaf page - 2 entries

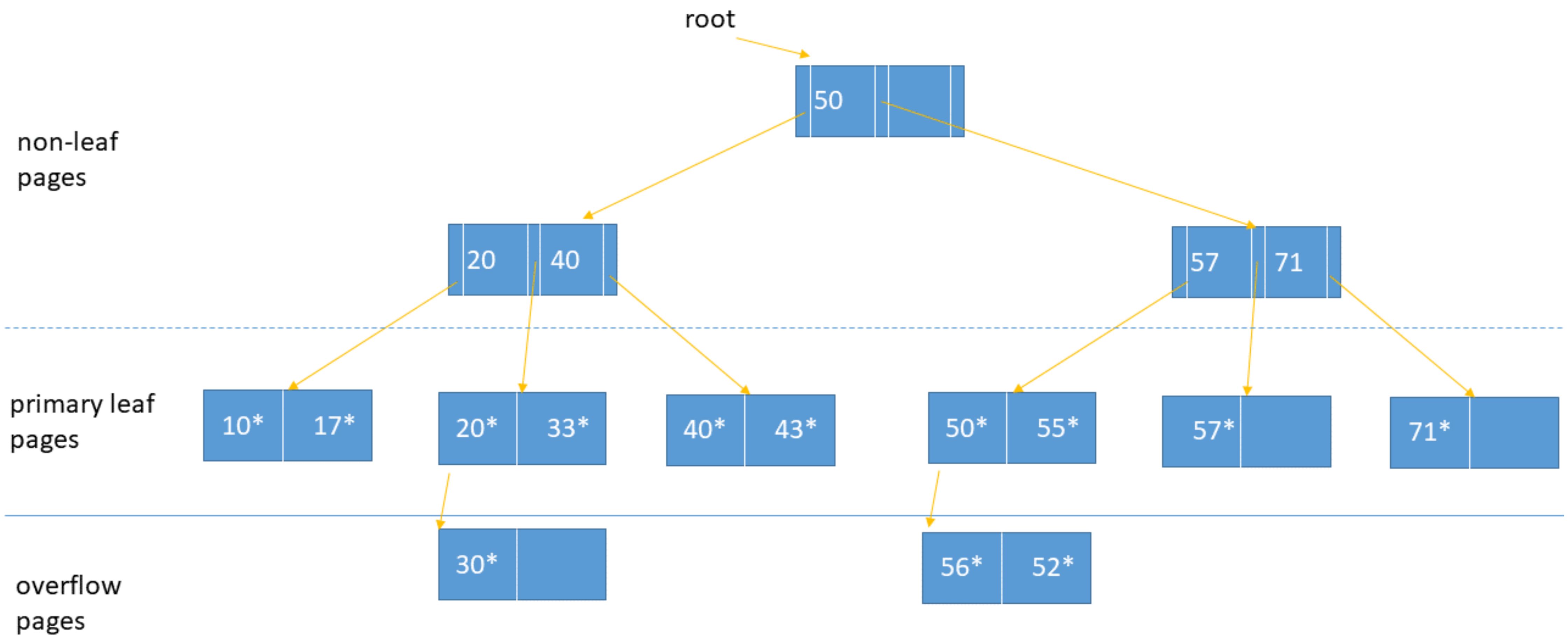


- only key values are shown

- after inserting 30*, 56*, 52*, 53*



- after deleting 53*, 68*, 82*

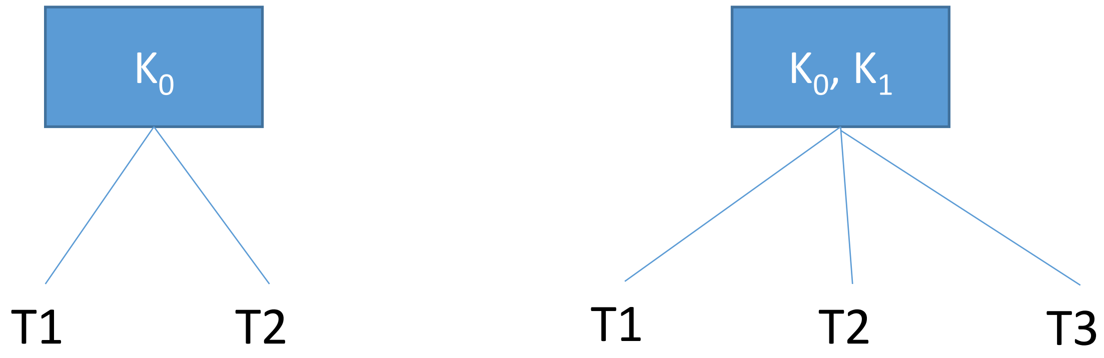


- benefits and drawbacks
  - better concurrent access, since only leaf pages are modified
  - long overflow chains can develop
    - usually not sorted (to optimize inserts)
    - irregular search time if structure not balanced
    - eliminated through deletes / file reorganization
  - when creating the tree - 20% of each page free for future inserts
  - ISAM - suitable when data size / distribution are relatively static

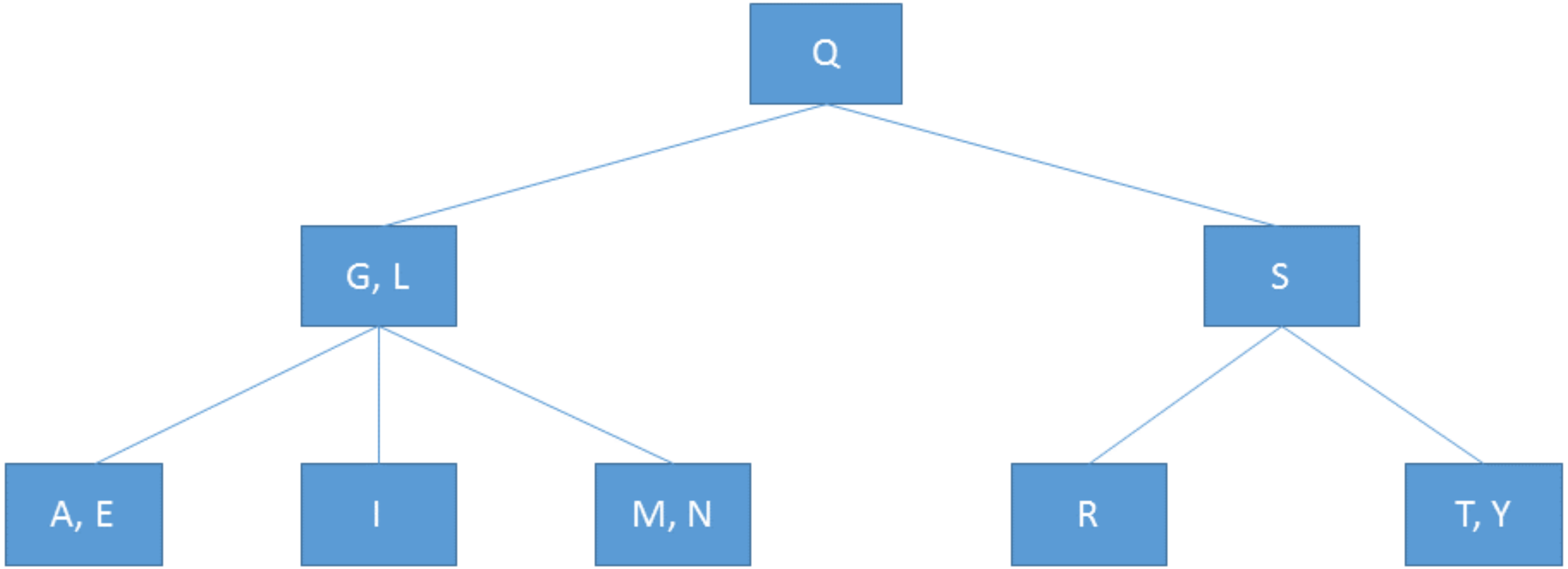
2-3 tree

2-3 tree storing key values (collection of distinct values)

- all the terminal nodes are on the same level
- every node has 1 or 2 key values
  - a non-terminal node with one value K_0 has 2 subtrees: one with values less than K_0, and one with values greater than K_0
  - a non-terminal node with 2 values K_0 and K_1, $K_0<K_1$, has 3 subtrees: one with values less than K_0, a subtree with values between K_0 and K_1, and a subtree with values greater than K_1



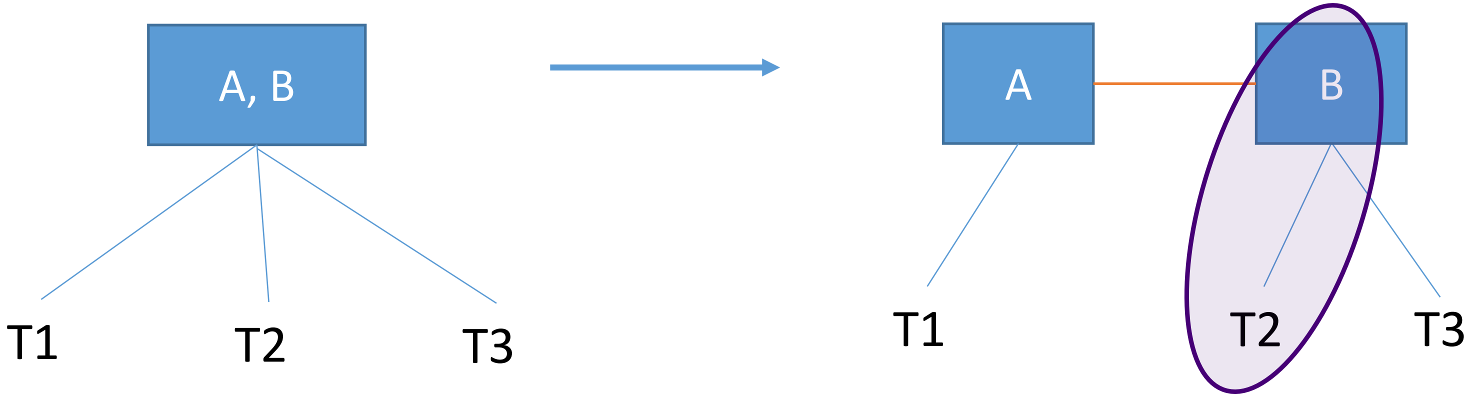
* Example (key values are letters)



- <u>storing a 2-3 tree</u>
  - 2-3 tree index storing the values of a key
  - tree - key value + address of record (file / DB address of record with corresponding key value)

- 2 options

  1. transform 2-3 tree into a binary tree
  - nodes with 2 values are transformed (see figure below)
  - nodes with 1 value - unchanged



  - the structure of a node

K	ADDR	PointerL	PointerR	IND

    - K - key value
    - ADDR - address of the record with the current key value (address in the file)
    - PointerL, PointerR - the 2 subtrees' addresses (address in the tree)

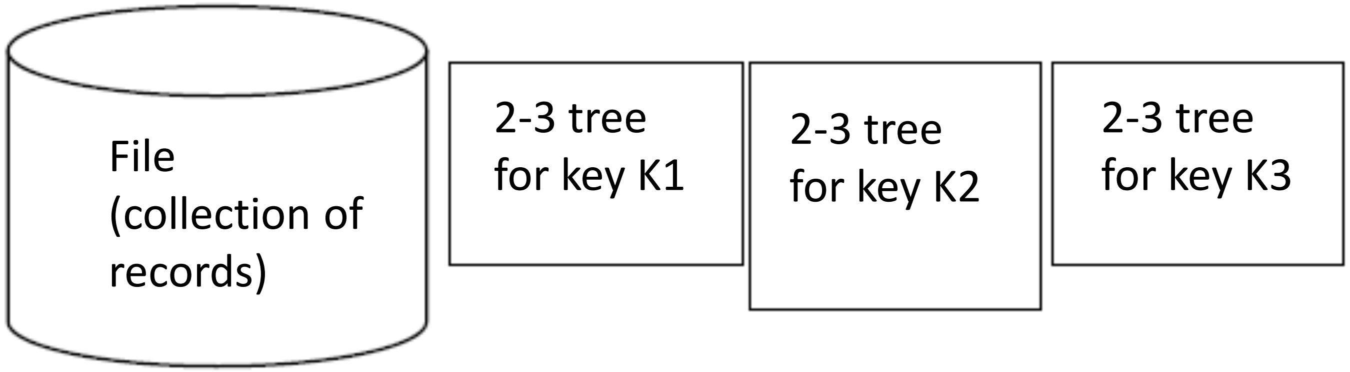
- IND - indicator that specifies the type of the link to the right (the 2 possible values can be seen in the previous figure)

2. the memory area allocated for a node can store 2 values and 3 subtree addresses

NV	K_1	$ADDR_1$	K_2	$ADDR_2$	$Pointer_1$	$Pointer_2$	$Pointer_3$

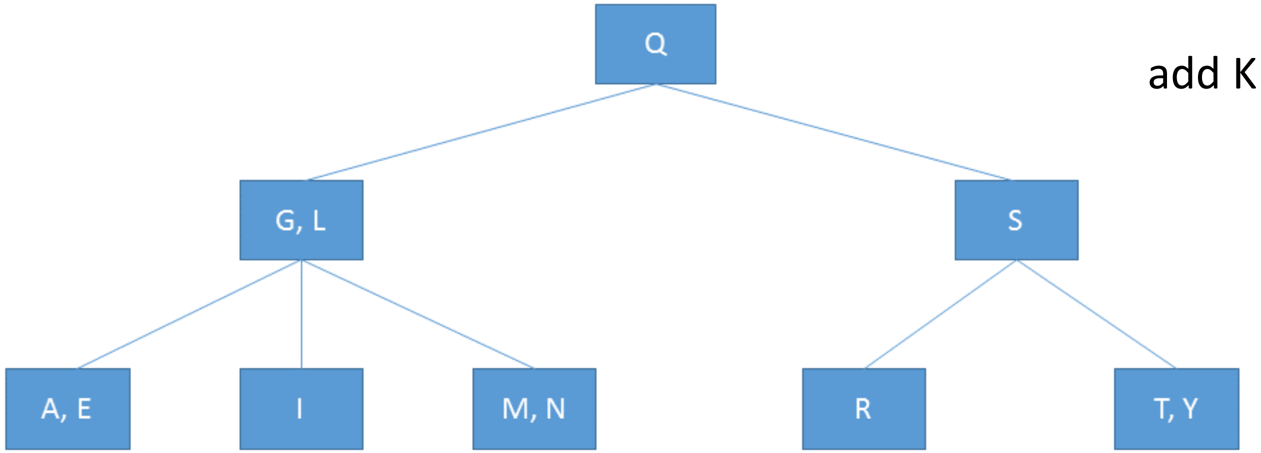
- NV – number of values in the node (1 or 2)
- K_1, K_2 – key values
- $ADDR_1$, $ADDR_2$ – the records' addresses (corresponding to K_1 and K_2)
- $Pointer_1$, $Pointer_2$, $Pointer_3$ – the 3 subtrees' addresses

- obs. a file (a relation in a relational DB) can have several associated 2-3 trees (one tree / key)

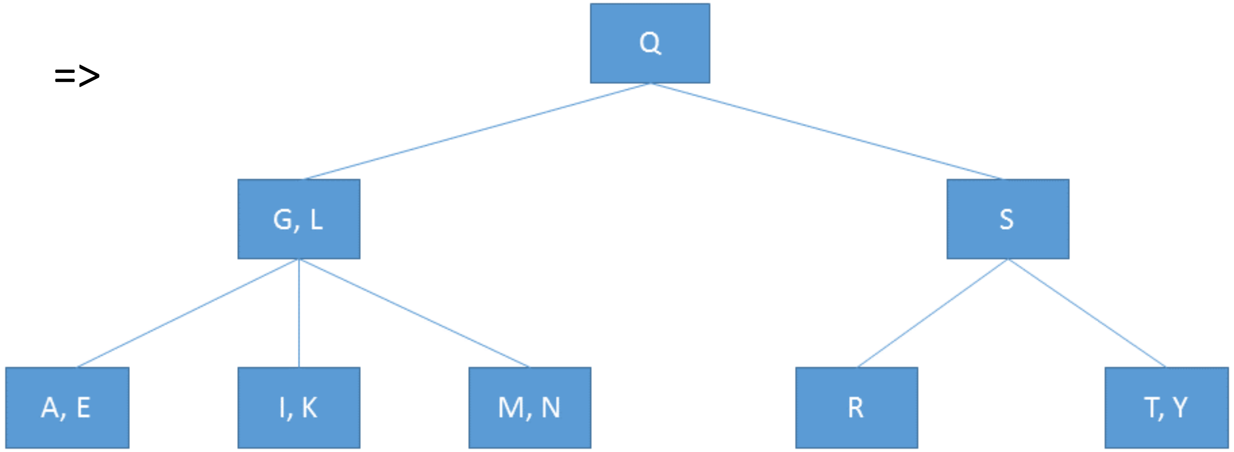


- operations in a 2-3 tree
  - searching for a record with key value K_0
  - inserting a record - description
  - removing a record - description
  - tree traversal (partial, total)

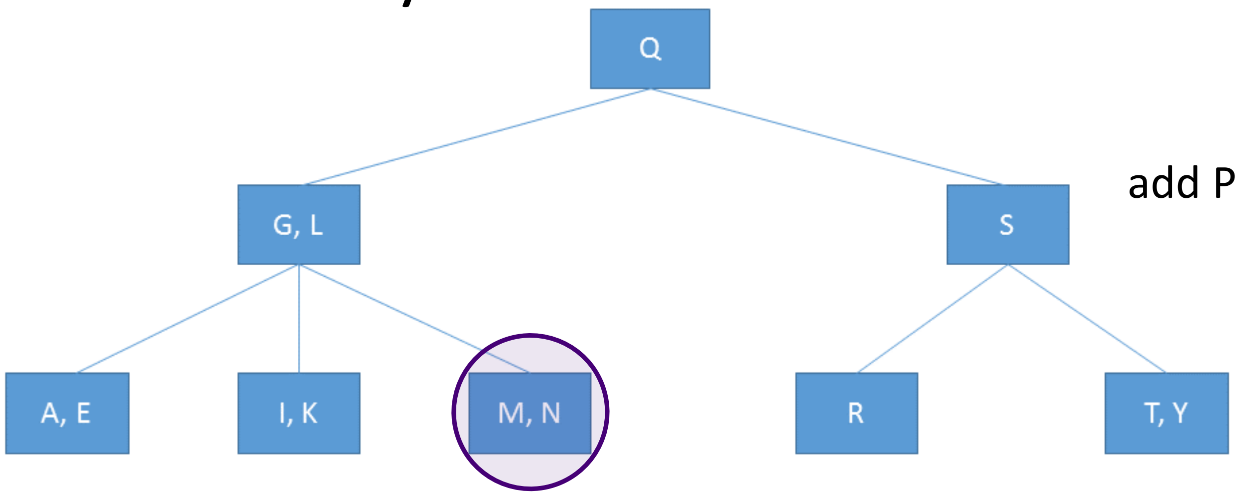
- <u>add a new value</u>
  - values in the tree must be distinct (the new value should not exist in the tree)
  - perform a test: search for the value in the tree; if the new value can be added, the search ends in a terminal node
  - if the reached terminal node has 1 value, the new value can be stored in the node



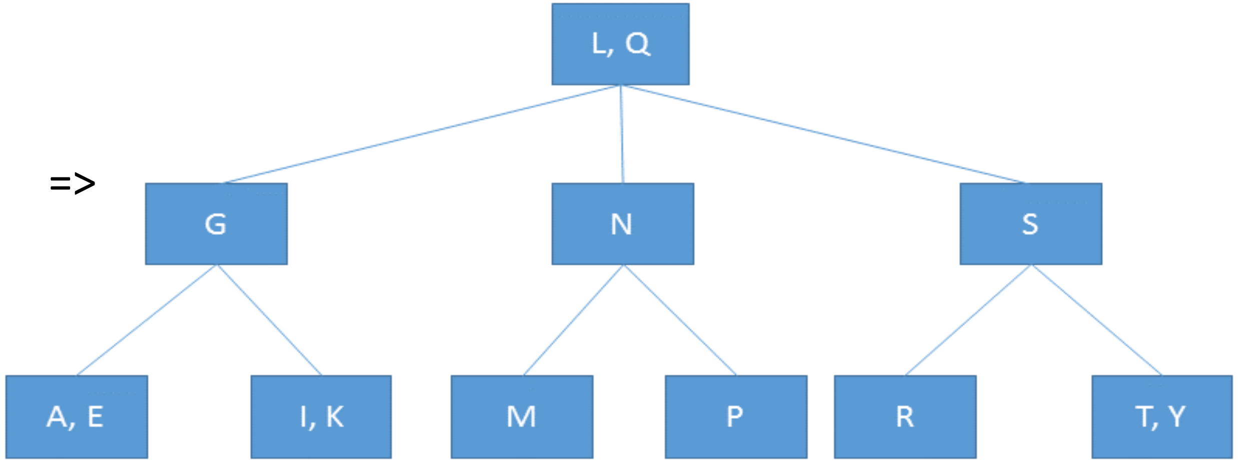
add K =>



- if the reached terminal node has 2 values, the new value is added to the node, the 3 values are sorted, the node is split into 2 nodes: one node will contain the smallest value, the 2nd node - the largest value, and the middle value is attached to the parent node; the parent is then analyzed in a similar manner



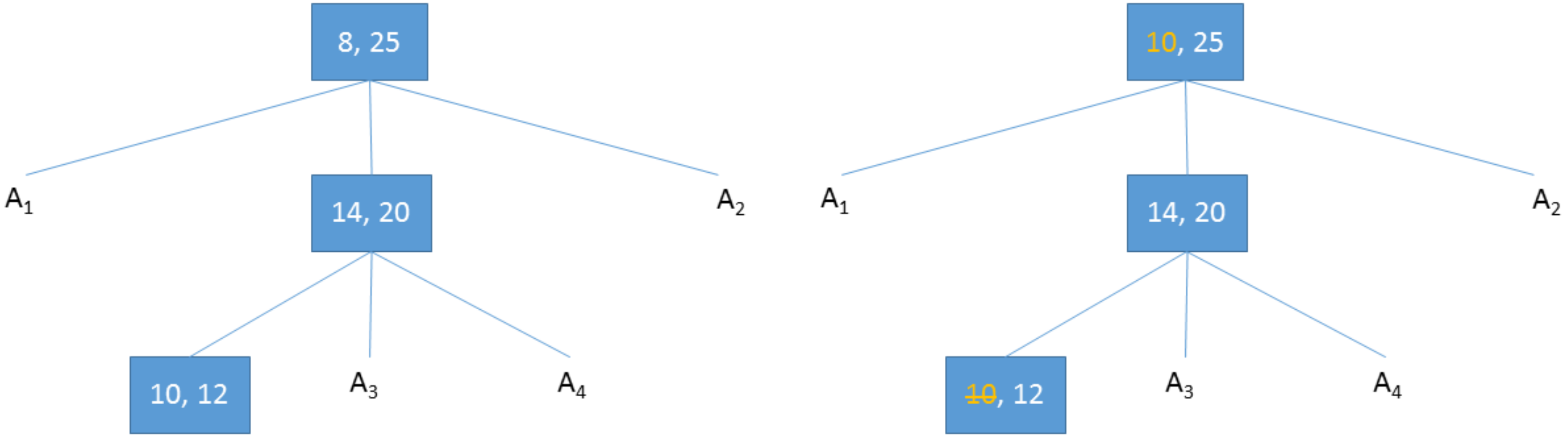
add P =>



- delete a value K_0

1. search for K_0; if K_0 appears in an inner node, change it with a neighbor value K_1 from a terminal node (there is no other value between K_0 and K_1)
    - K_1's previous position (in the terminal node) is eliminated

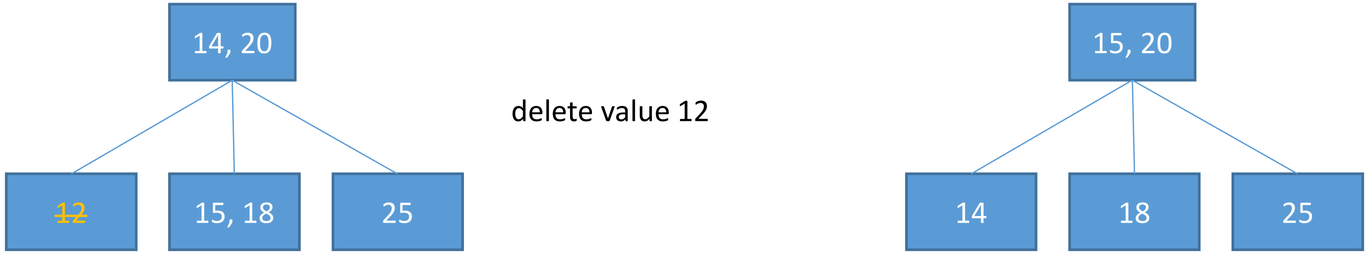
- e.g., remove 8:

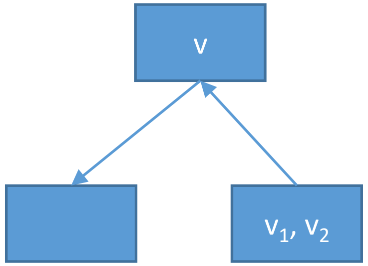


2. perform this step until case a / b occurs

a. if the current node (from which a value is removed) is the root or a node with 1 remaining value, the value is eliminated; the algorithm ends

b. if the delete operation empties the current node, but 2 values exist in one of the sibling nodes (left / right), 1 of the sibling's values is transferred to the parent, 1 of the parent's values is transferred to the current node; the algorithm ends

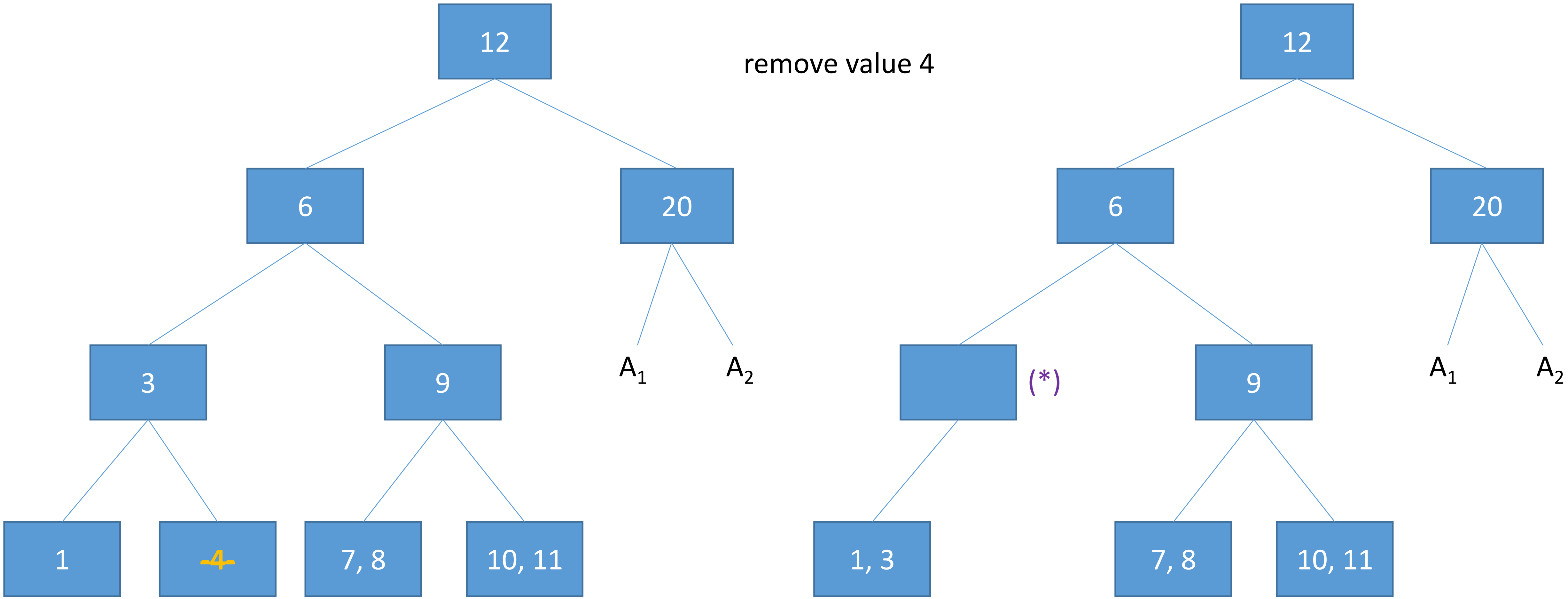


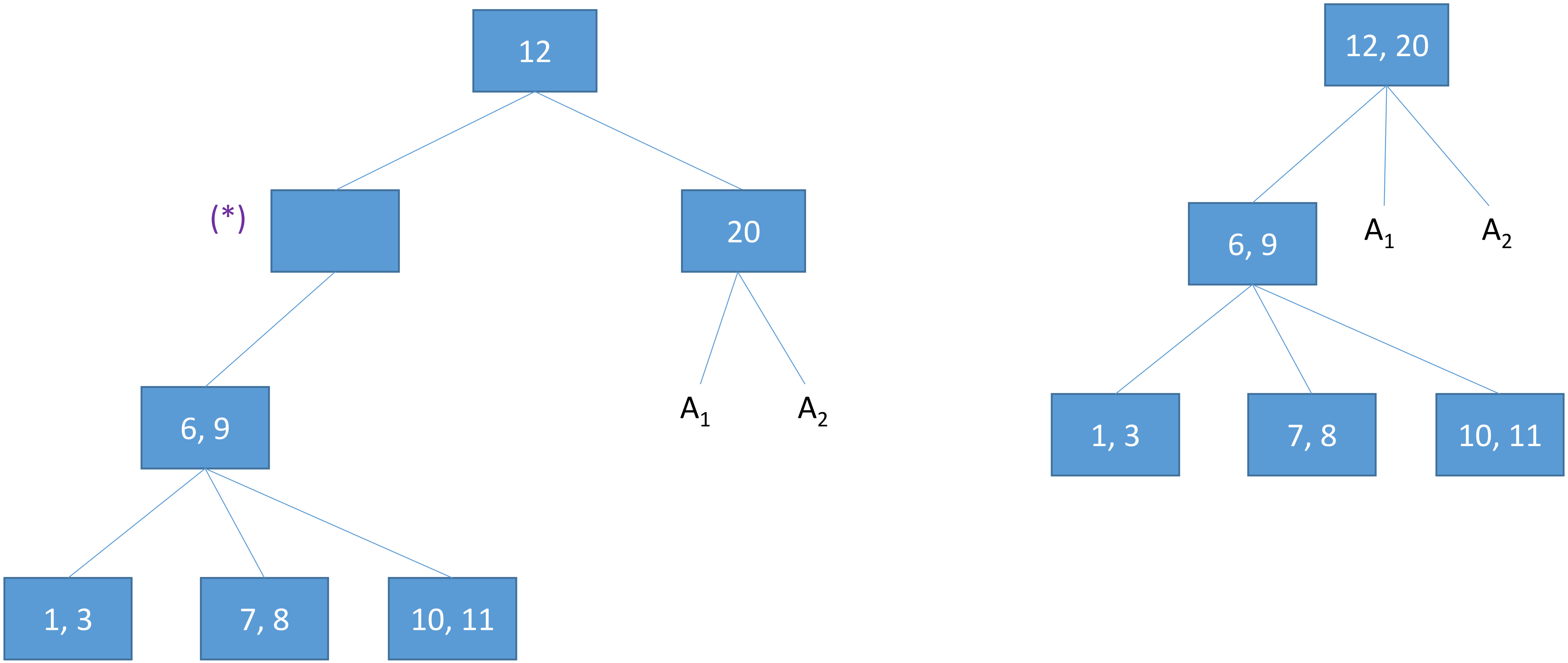


c. if the previous cases do not occur (current node has no values, sibling nodes have 1 value each), then the current node is merged with a sibling and a value from the parent node; case 2 is then analyzed for the parent

- if the root is reached and it has no values, it is eliminated and the current node becomes the root

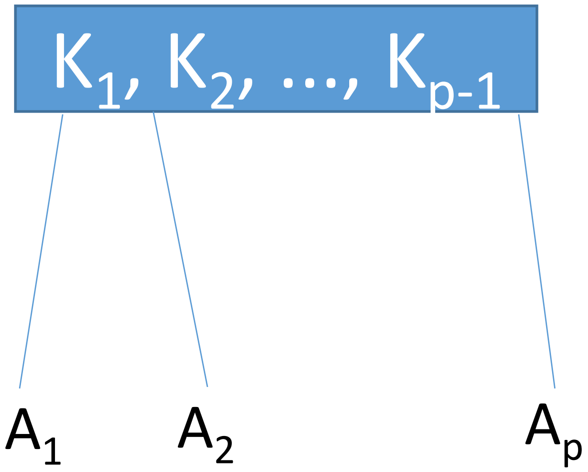
- example: case c for the node marked with (*)




12
(*)
20
6, 9
A1
A2
1, 3
7, 8
10, 11
12, 20
6, 9
A1
A2
1, 3
7, 8
10, 11

B-tree

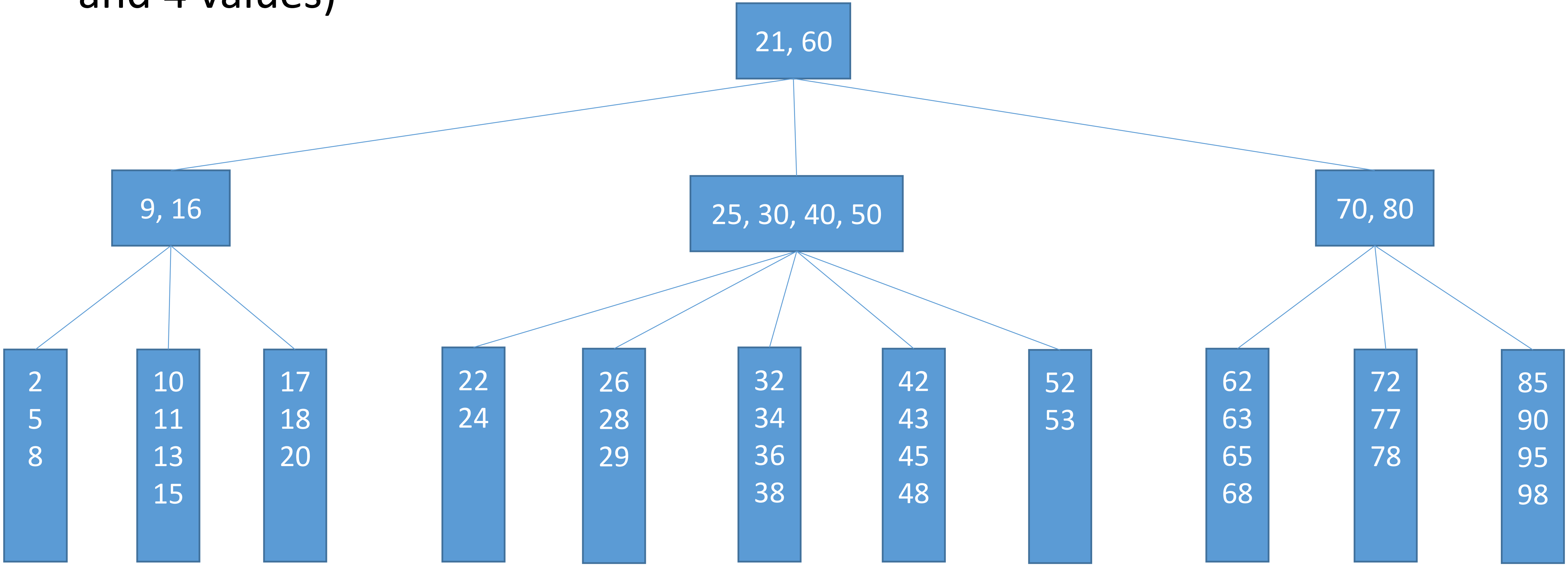
- generalization of 2-3 trees
- B-tree of order m balanced!
  1. if the root is not a terminal, it has at least 2 subtrees
  2. all terminal nodes – same level
  3. every non-terminal node – at most m subtrees
  4. a node with p subtrees has p-1 ordered values (ascending order): $K_1<K_2<\ldots<K_{p-1}$
     - A_1: values less than K_1
     - A_i: values between K_{i-1} and K_i, $i=2,\ldots,p-1$
     - A_p: values greater than K_{p-1}
  5. every non-terminal node – at least $\left\lceil \frac{m}{2} \right\rceil$ subtrees



- obs. limits on number of subtrees (and values) / node result from the manner in which inserts / deletes are performed so that the second requirement in the definition is met

* Example - B-tree of order 5

- non-terminal, non-root node – at most 5, at least 3 subtrees (between 2 and 4 values)



- B-tree of order m
  - storing the values of a key (a database index)
  - tree
    - key value + address of record

1. transformed into a binary tree
    - 2-3 tree method

2. the memory area allocated for a node can store the maximum number of values and subtree addresses

NV	K_1	$ADDR_1$	...	K_{m-1}	$ADDR_{m-1}$	$Pointer_1$	...	$Pointer_m$

- NV - number of values in the node
- K_1, ..., K_{m-1} - key values
- $ADDR_1$, ..., $ADDR_{m-1}$ - the records' addresses (corresponding to the key's values)
- $Pointer_1$, ..., $Pointer_m$ – subtree addresses

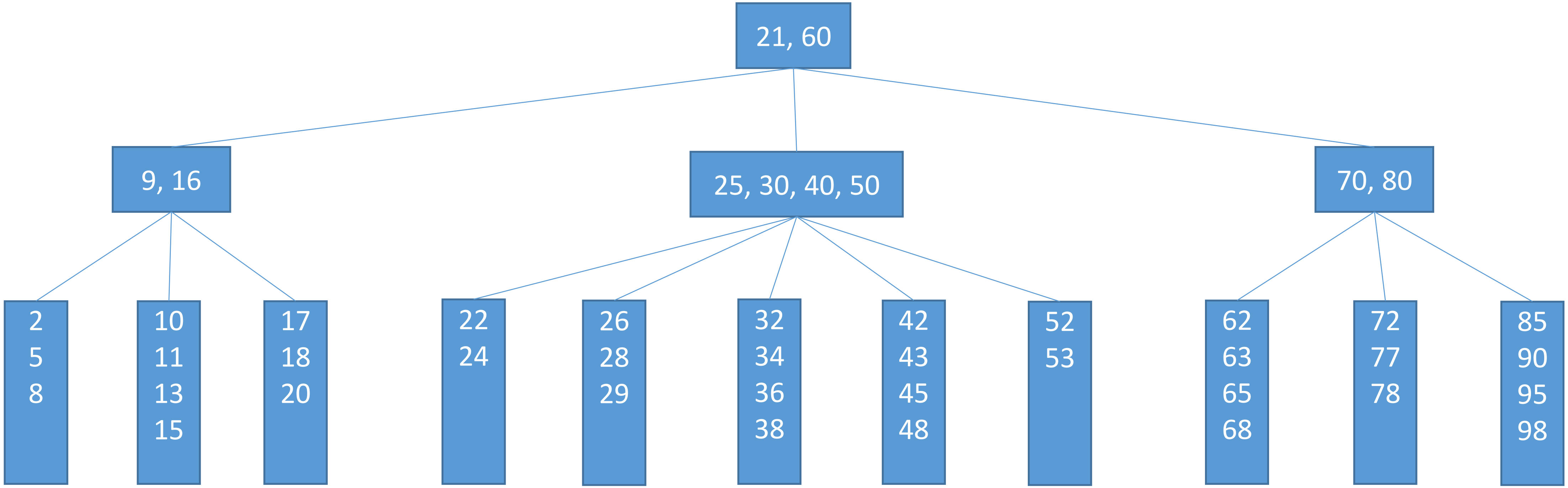
- B-tree of order m
  - useful operations in a B-tree
    - searching for a value
    - adding a value - description
    - removing a value- description
    - tree traversal (partial, total)

- B-tree of order m
  - <u>adding a new value</u>

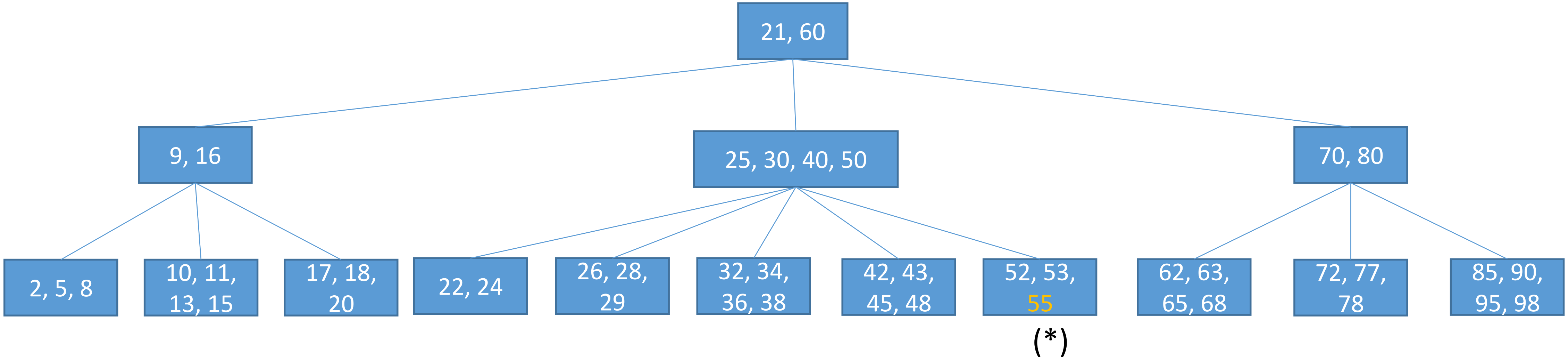
    1. values in the tree must be distinct (the new value should not exist in the tree); perform a test (search for the value in the tree)

    - if the new value can be added, the search ends in a terminal node

    2. if the reached terminal node has less than m-1 values, the new value can be stored in the node, e.g., 55 is added to the tree below:



- B-tree of order m
  - adding a new value
    - the resulting tree is shown below:

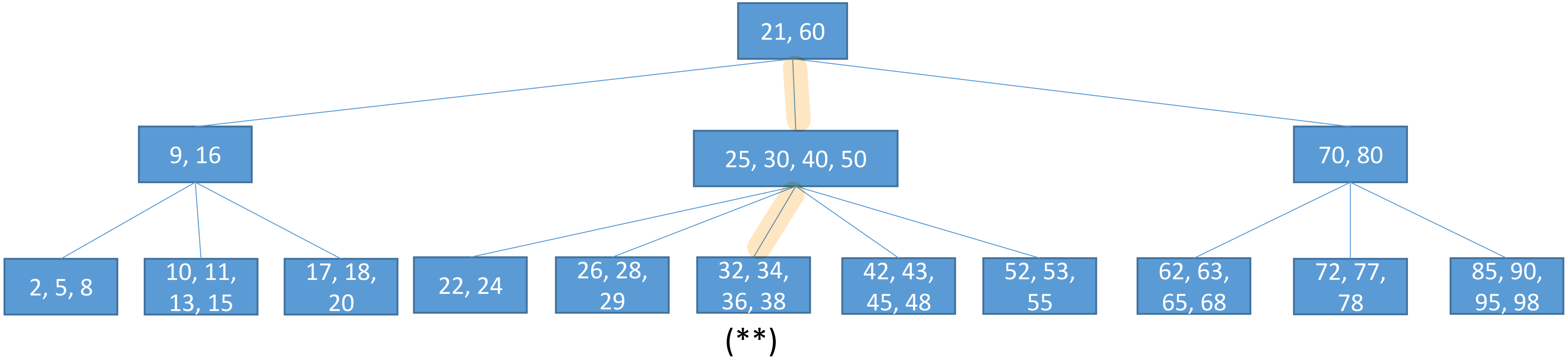


    - 55 belongs to the node marked with (*), which can store at most 4 values

- B-tree of order m
  - adding a new value

    3. if the terminal node already has m-1 values, the new value is attached to the node, the m values are sorted, the node is split into 2 nodes, and the middle value (median) is attached to the parent node; the parent is then analyzed in a similar manner

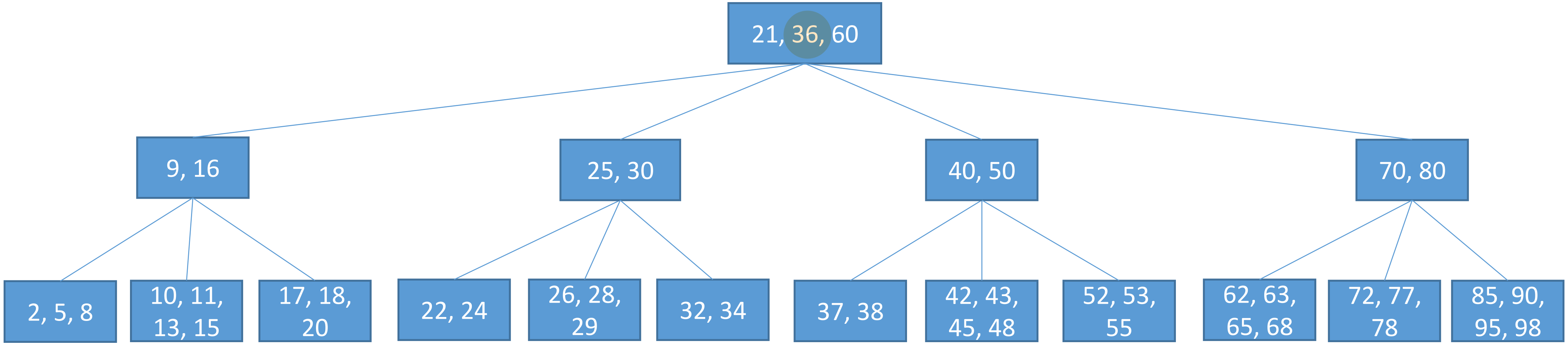
      - e.g., add 37 to the tree below



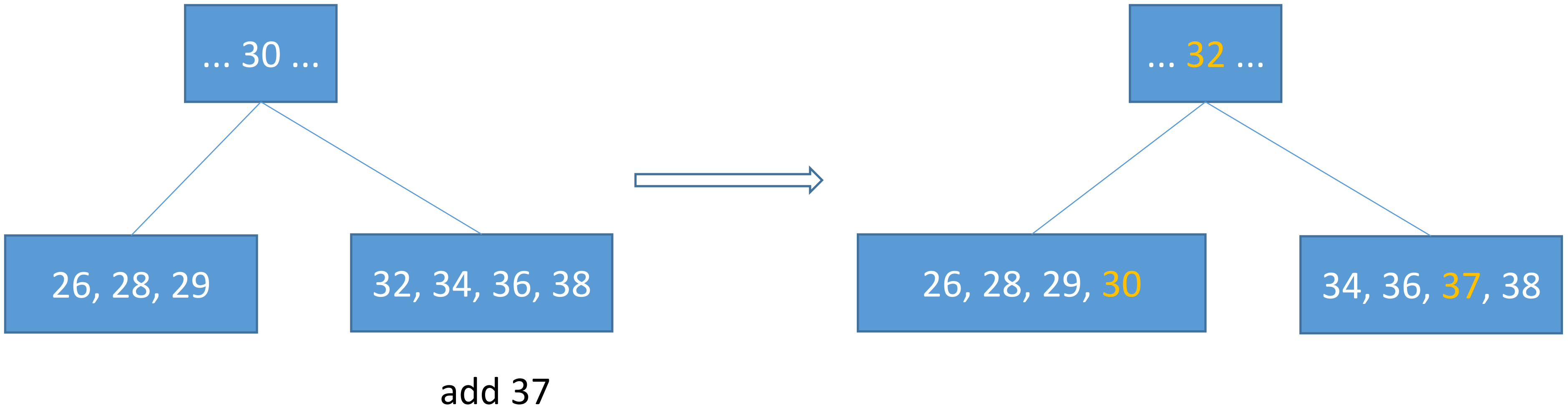
      - the node marked with (**) should contain values *32, 34, 36, 37, 38*

->

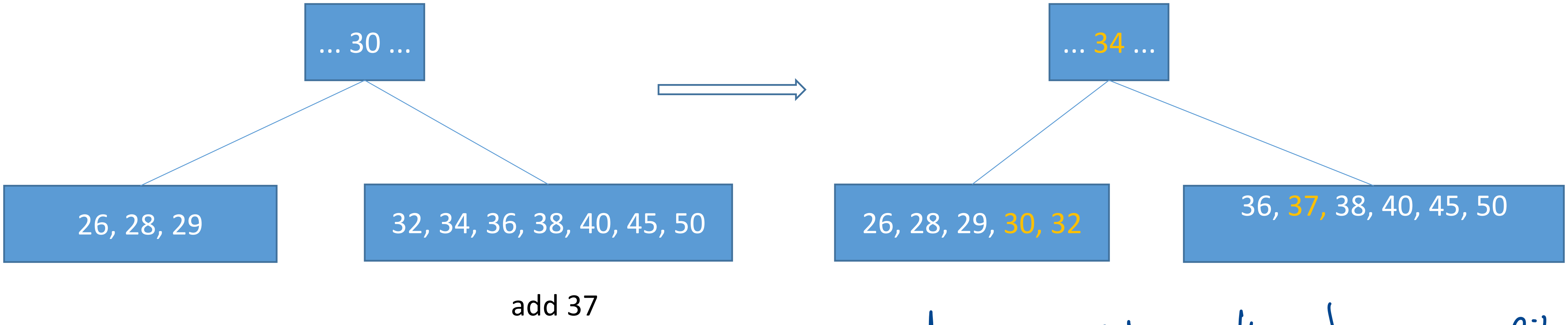
- B-tree of order m
  - adding a new value
    - since the node's capacity is exceeded, it is split into nodes *32, 34*, and *37, 38*, and 36 is attached to the parent node (with values *25, 30, 40, 50*)
    - in turn, the parent must be split into 2 nodes (values *25, 30*, and *40, 50*), and 36 is attached to its parent



- B-tree of order m
    - adding a new value
        - <u>optimizations</u>
            - before performing a split - analyze whether one or more values can be transferred from the current node (with m-1 values) to a sibling node
            - e.g., B-tree of order 5 (non-terminal node - between 2 and 4 values, i.e., between 3 and 5 subtrees):



- B-tree of order m
  - adding a new value
    - optimizations
    - e.g., B-tree of order 8 (non-terminal node - between 3 and 7 values, i.e., between 4 and 8 subtrees):

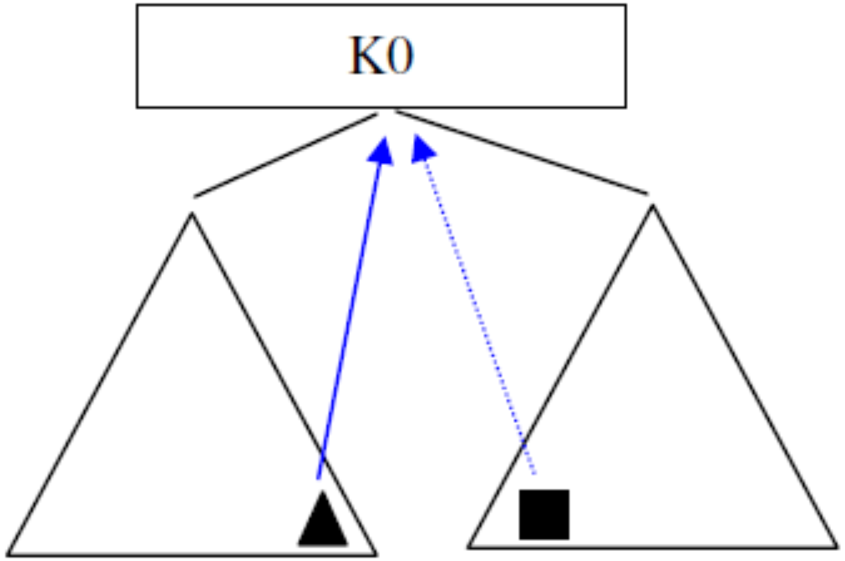


* cheaper, since there's no split

- B-tree of order m
  - <u>removing a value</u>
    - a node can have at most m subtrees, i.e., a maximum of m-1 values, and at least $\left\lceil \frac{m}{2} \right\rceil$ subtrees, i.e., at least $\left\lceil \frac{m}{2} \right\rceil - 1 = \left\lfloor \frac{m-1}{2} \right\rfloor$ values
    - when eliminating a value from a node, an underflow can occur (the node can end up with less values than the required minimum)
  - eliminate value K_0

    1. search for K_0; if it doesn't exist, the algorithm ends

    2. if K_0 is found in a non-terminal node (like in the figure on the right), K_0 is replaced with a *neighbor value* from a terminal node (this value can be chosen between 2 values from the trees separated by K_0)



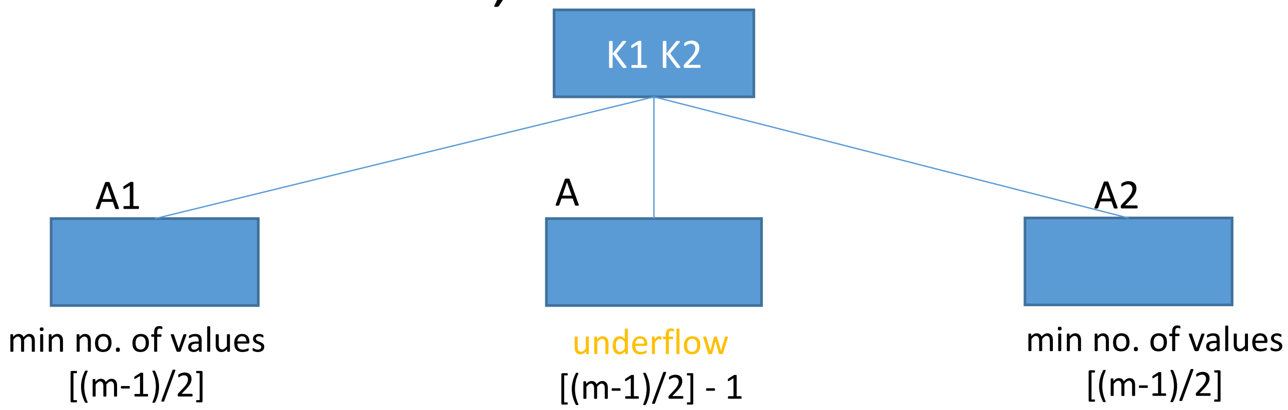
- B-tree of order m
  - removing a value

3. perform this step until case a / b occurs

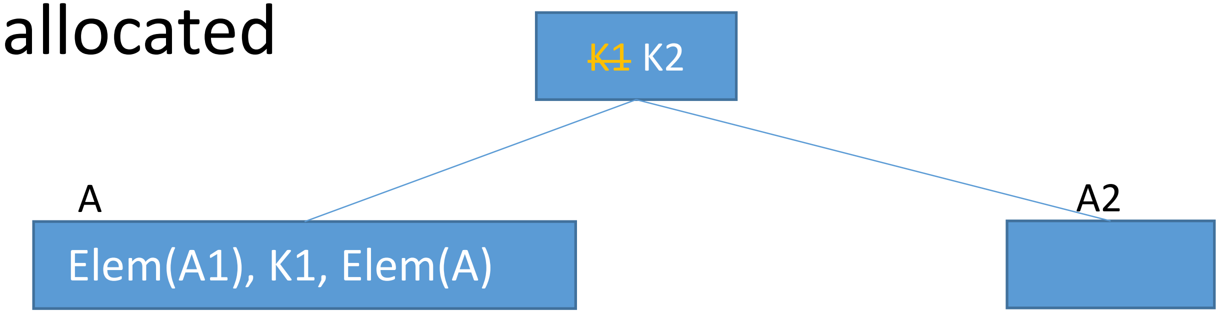
a. if the current node (from which a value is removed) is the root or underflow doesn't occur, the value is eliminated; the algorithm ends

b. if the delete operation causes an underflow in the current node (A), but one of the sibling nodes (left / right - B) has at least 1 extra value, values are transferred between A and B via the parent node; the algorithm ends

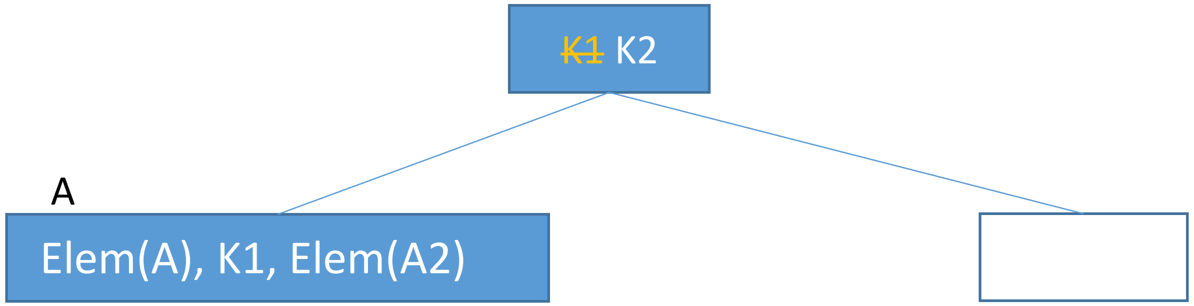
c. if there is an underflow in A, and sibling nodes A1 and A2 have the minimum number of values, nodes must be concatenated:



- B-tree of order m
  - removing a value
    - if A1 exists, A1 is merged with A and value K1 (separating A1 from A); the node at address A1 is deallocated



    - if there is no A1 (A is the first subtree for its parent), A is merged with A2 and K1 (separating A from A2); the node at address A2 is deallocated



    - case 3 is then analyzed for the parent node
    - if the root is reached and has no values, it is removed and the current node becomes the root

- B-tree of order m
  - obs. a block stores a node from a B-tree
- e.g.:
  - key size: 10b
  - record address / node address: 10b
  - NV value (number of values in the node): 2b
  - block size: 1024b (10b for the header)
- then: 2+(m-1)*(10+10)+m*10=1024-10 => m=34
- if the size of a block is 2048b and the other values are unchanged, then the order of the tree is m = 68, i.e., a node can have between 33 and 67 values

- B-tree of order m
- the maximum number of required blocks (from the file that stores the B-tree) when searching for a value - the maximum number of levels in the tree; for m=68, if the number of values is 1.000.000, then:
  - the root node (on level 0) contains at least 1 value (2 subtrees)
  - on the next level (level 1) - at least 2 nodes * 33 values/node = 66 values
  - level 2 – at least 2*34 nodes * 33 values/node = 2.244 values
  - level 3 – at least 2*34*34 nodes * 33 values/node = 76.296 values
  - level 4 – at least 2*34*34*34 nodes * 33 values/node = 2.594.064 values, which is greater than the number of existing values => this level does not appear in the tree

    because 2.594.064 > 1000000

=> at most 4 levels in the tree

- after at most 4 block reads and a number of comparisons in main memory, it can be determined whether the value exists (the corresponding record's address can then be retrieved) or the search was unsuccessful

References

- [Ra02] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems (3rd Edition), McGraw-Hill, 2002
- [Ra07] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems, McGraw-Hill, 2007, http://pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed.html
- [Ta13] ȚÂMBULEA, L., Curs Baze de date, Facultatea de Matematică și Informatică, UBB, 2013-2014
- [Si11] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts (6th Edition), McGraw-Hill, 2011
- [Kn76] KNUTH, D.E., Tratat de programare a calculatoarelor. Sortare și căutare. Ed. Tehnică, București, 1976
- [Ga09] GARCIA-MOLINA, H., ULLMAN, J., WIDOM, J., Database Systems: The Complete Book (2nd Edition), Pearson Education, 2009