Lecture 10

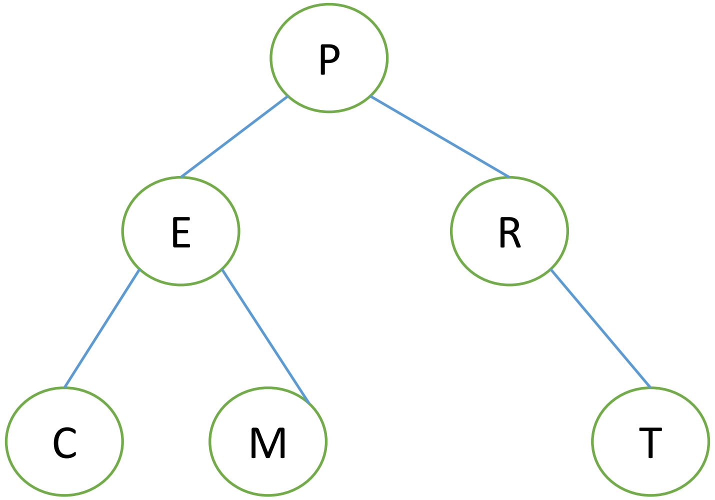
Indexes (II).
Binary trees. ISAM. 2-3 trees. B-tree. B+ tree.

Binary trees

- the heap or sorted files – useful for statistical tables
- Binary trees: are efficient for insert / delete records and are used binary search algorithms
- Memory structure for a node

K	Data	Pointer (left)	Pointer (right)

- Memory structure for a binary tree
  - node collection; referenced ***root***
  - list of free ***nodes*** (linked through pointer (left))
  - all the terminal nodes are on the same level

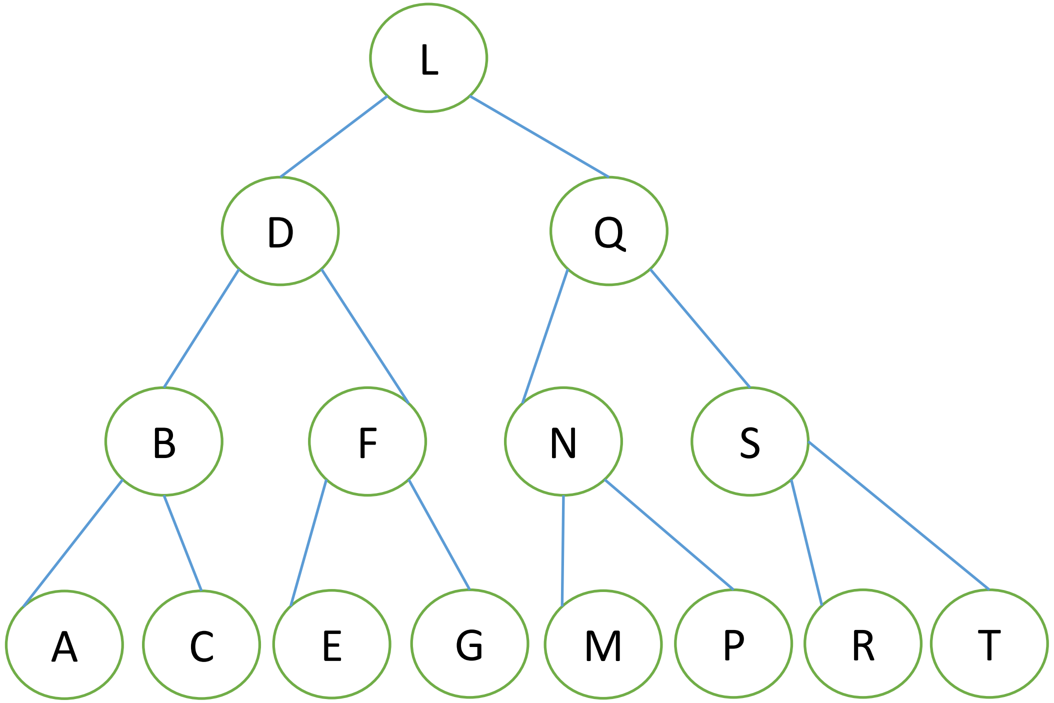


- Insert record
  - find the position of the record
  - store the new record in a free node
  - bound the node to the parent node
- Delete record
  - search record
  - 3 cases
    - no child: parent pointer=NULL
    - 1 child: put the child node to the parent node
    - 2 children: replace with the closest neighbour
  - add the node in the list of free nodes

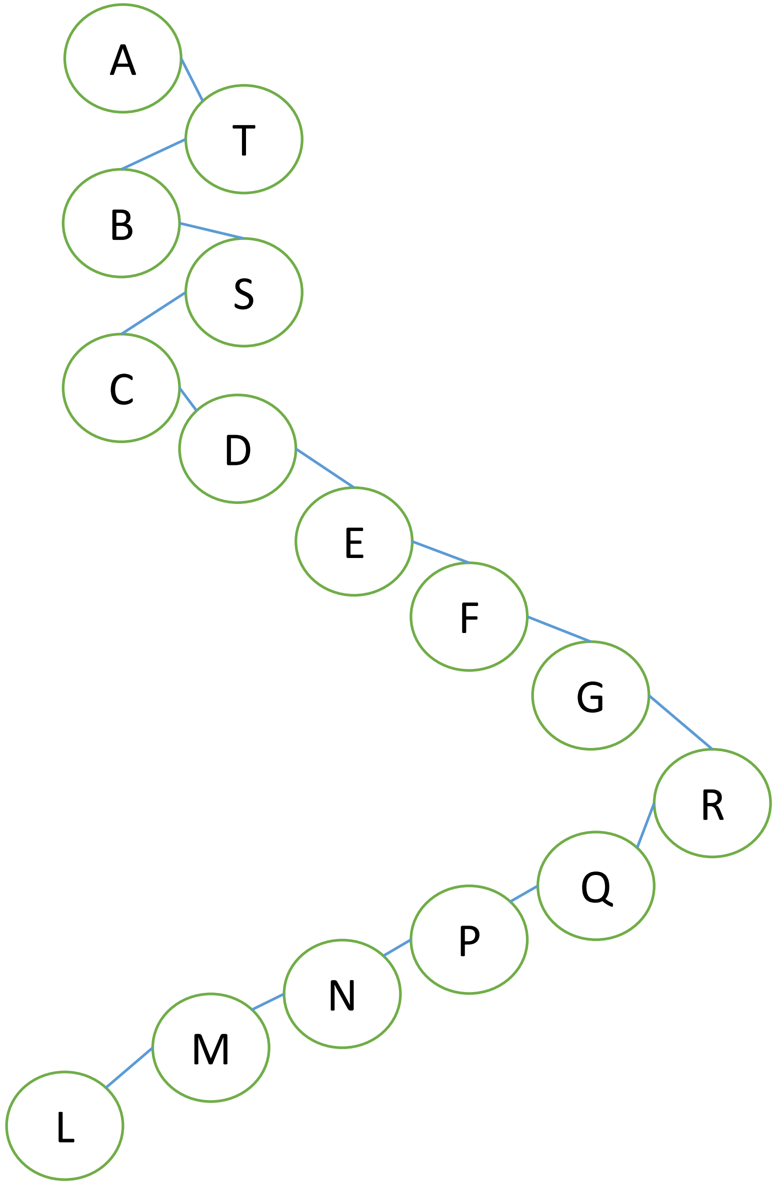
Binary trees

- anomalies in the insert

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



- A, T, B, S, C, D, E, F, G, R, Q, P, N, M, L
- the search time depends on the insert order of the records



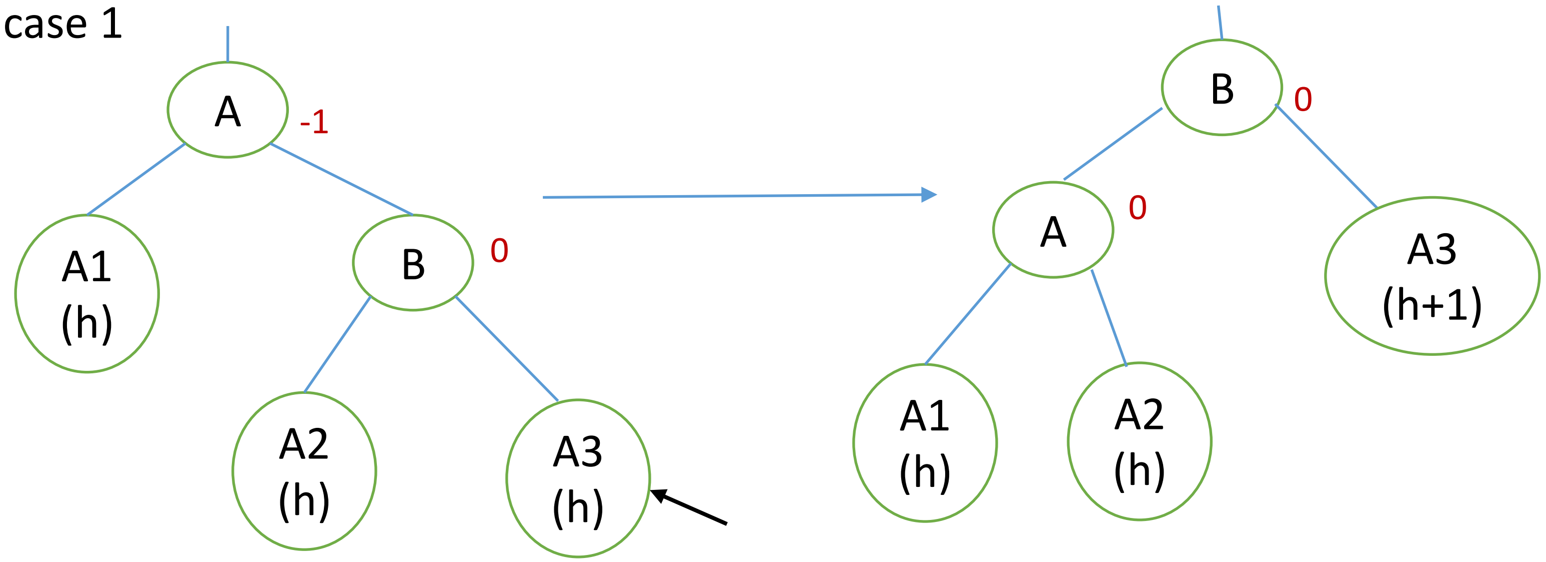
Binary trees

Optimal Binary trees

- the terminal nodes (leaf) are positioned on most 2 levels
- the maintenance is difficult: consume time to insert / delete / update

Balanced Binary trees

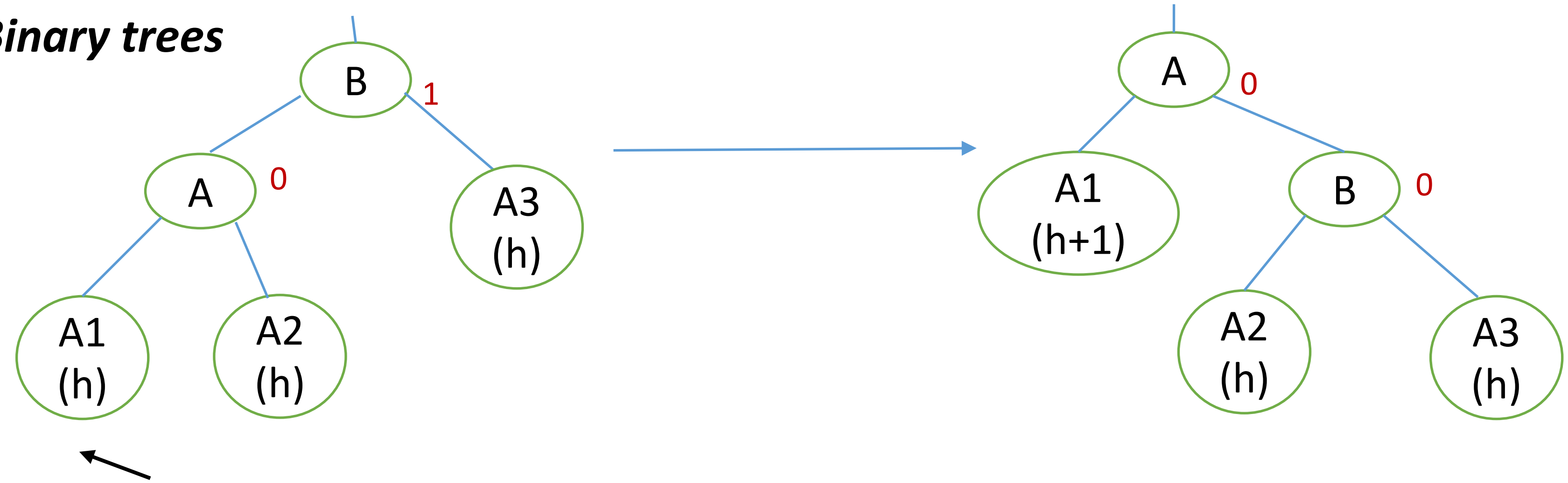
- for each node, the difference between the heights of the subtrees is 0, 1 or -1
- the height of the binary tree is the dimension of the longest way from the root to the terminal node (leaf)
- have a reduced number of operations to maintain
- 6 possible cases of non-balanced trees after an insert operation:
  - case 1



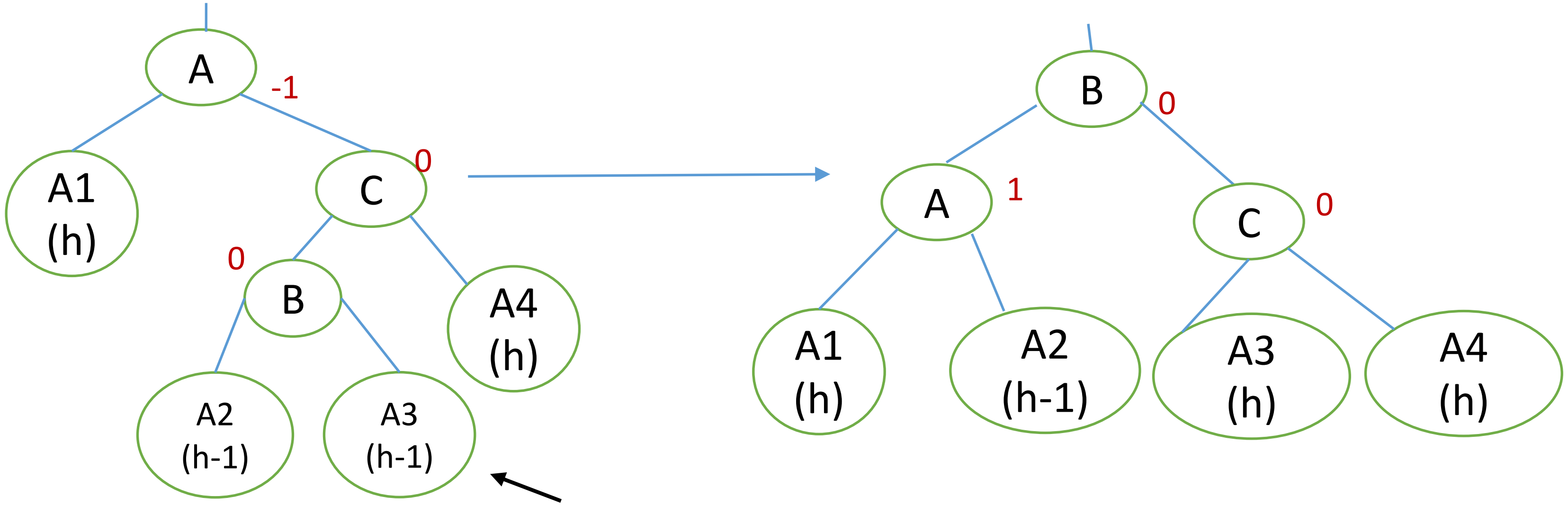
Binary trees

Balanced Binary trees

- case 2



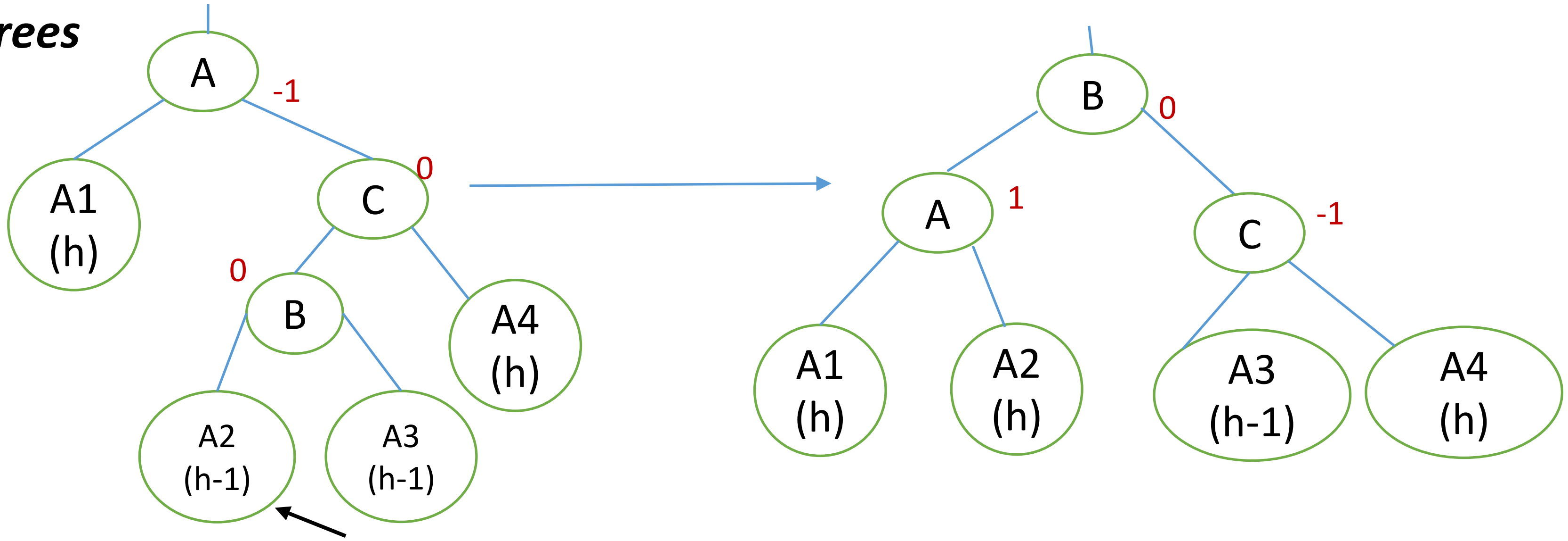
- case 3



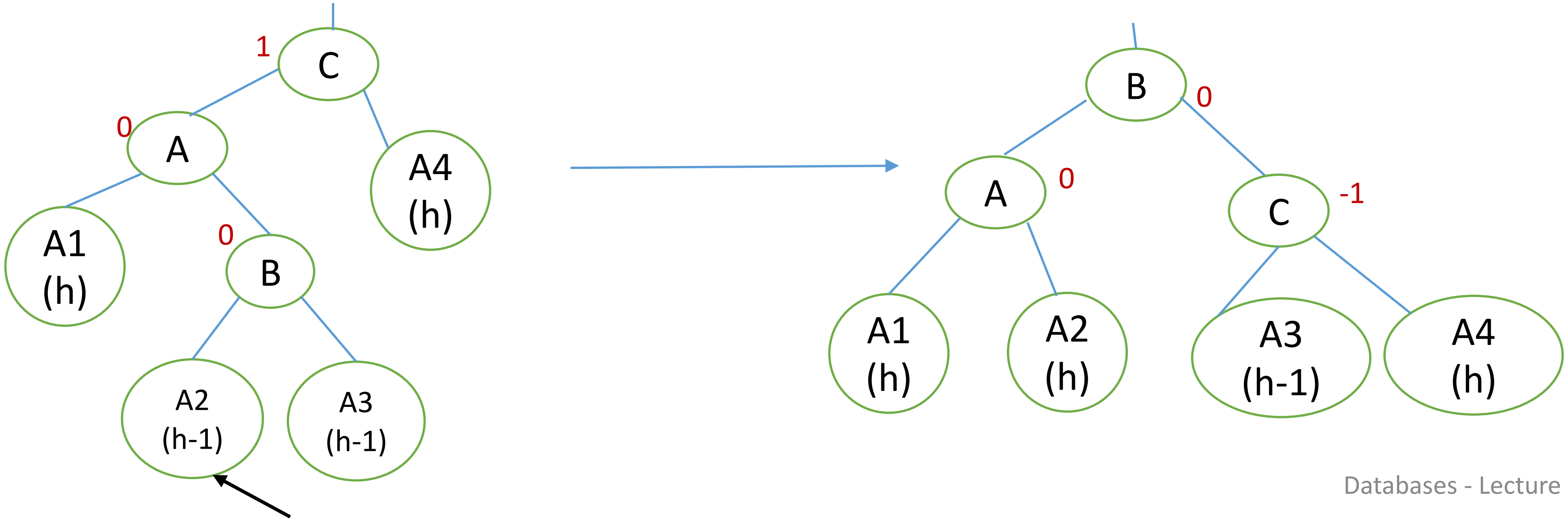
Binary trees

Balanced Binary trees

- case 4



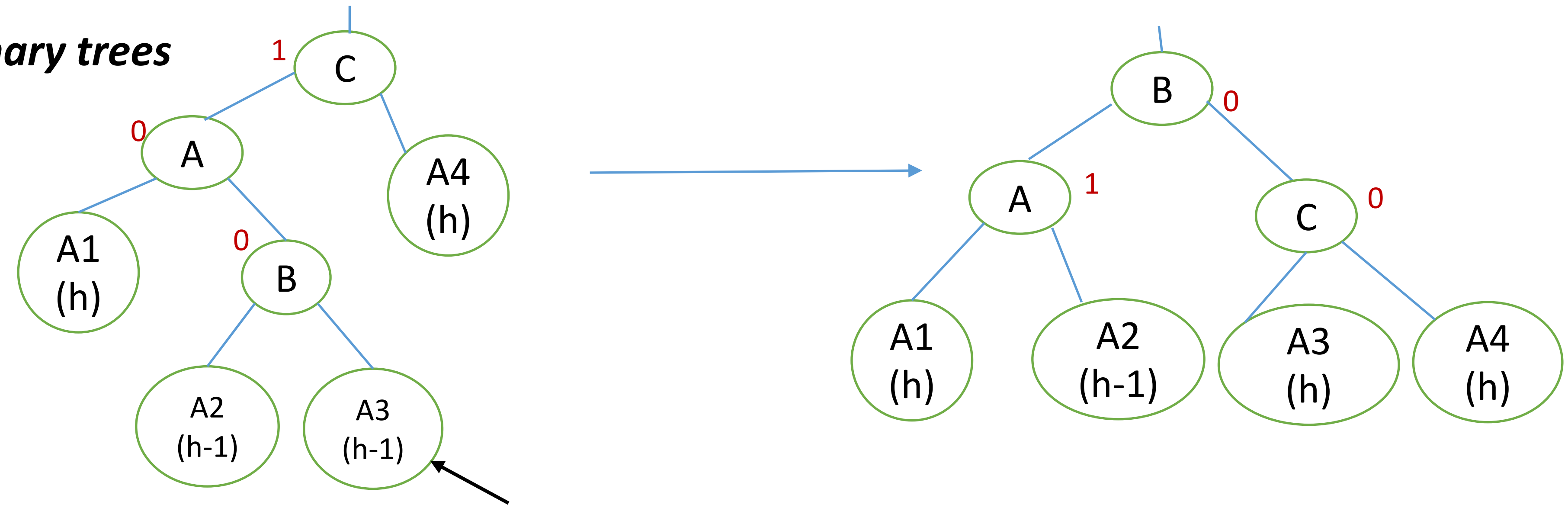
- case 5



Binary trees

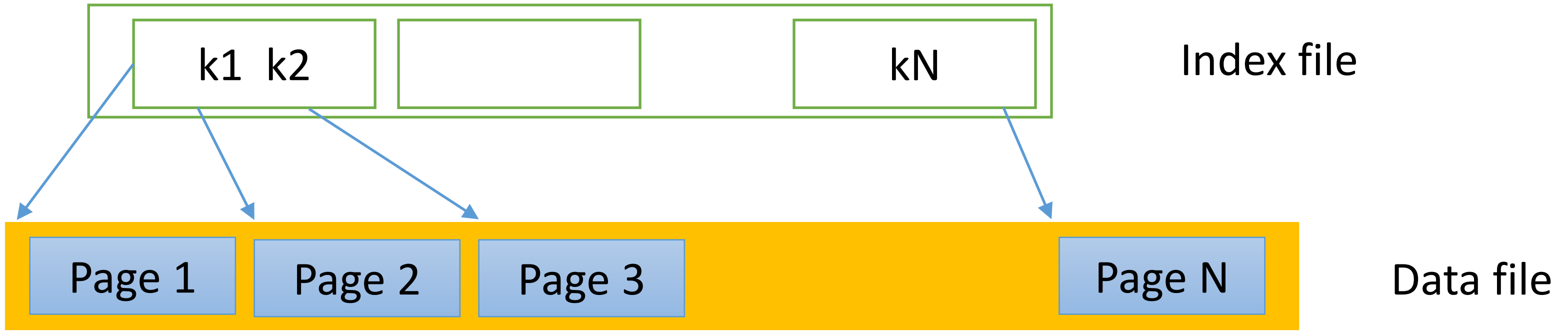
Balanced Binary trees

- case 6



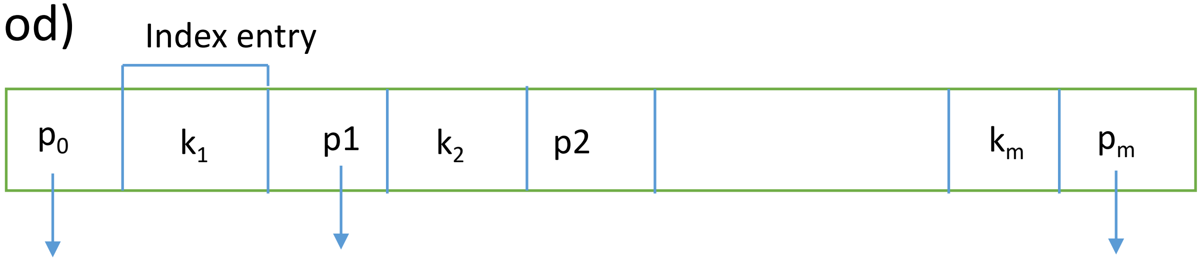
Search on intervals

- *Return the students with grade>9* – if the data records are store in an ordered file, through binary search can be found the first student and then are read all the following records; the cost can be high
- Solution: create an index file (the binary search will be done on the index file, so it will be cheaper)

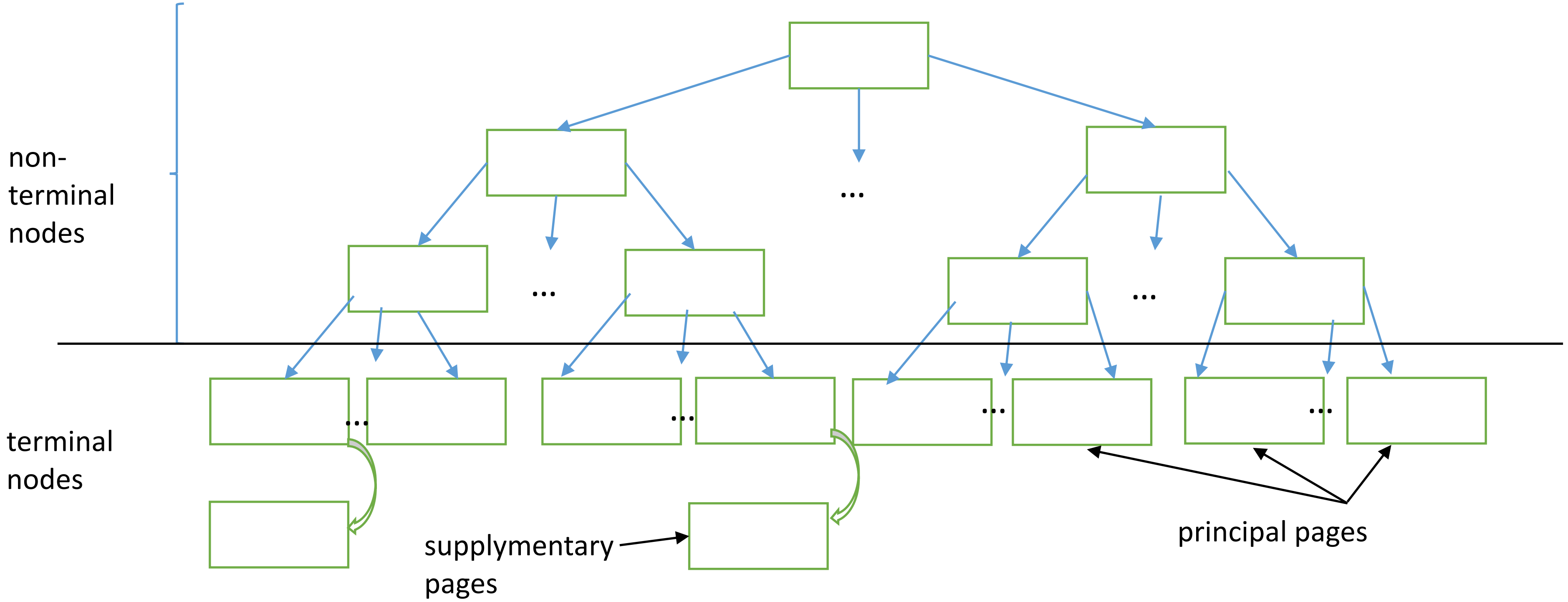


ISAM

ISAM (Indexed Sequential Access Method)



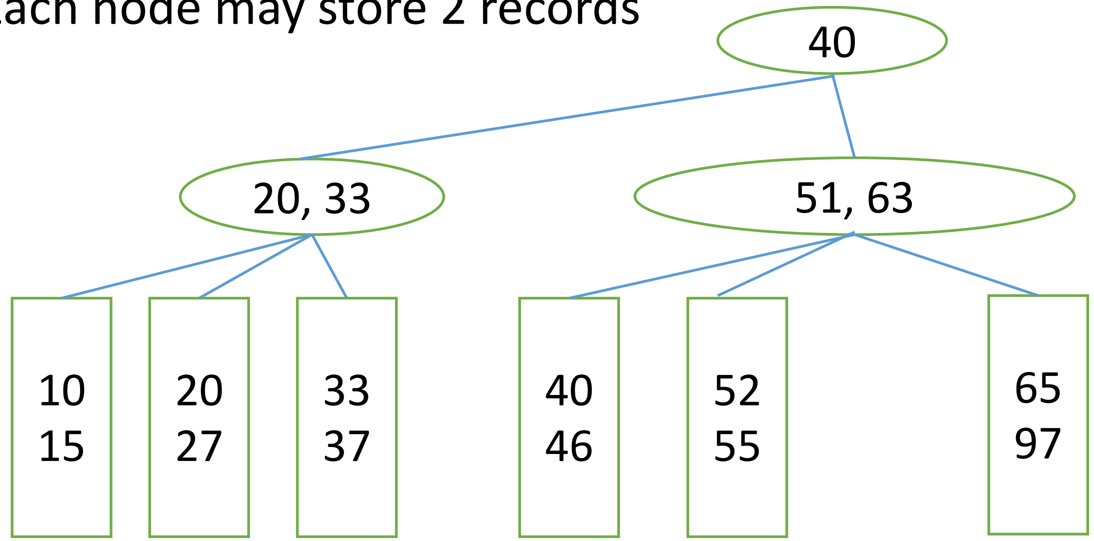
- the index file can be big, but the same idea can apply repeatable

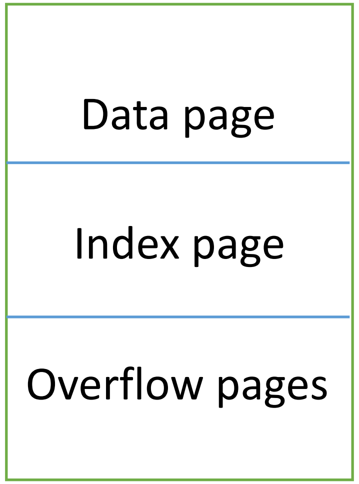


ISAM

- Create file: non-terminal pages are sequentially allocated, sort by the search key, then are memorized the indexed pages and then the space for supplementary pages
- entry index: <value_search_key, id_page> will direct the search to the non-terminal pages
- search: compares of the key starting from the root to the terminal pages. Cost $\log_F N$, where F is the number of entries per index page and N is the number of terminal pages
- insert: search the terminal page and add record
- delete: find and remove the record from the terminal node; an extra page can be deallocated (overflow)

- Each node may store 2 records

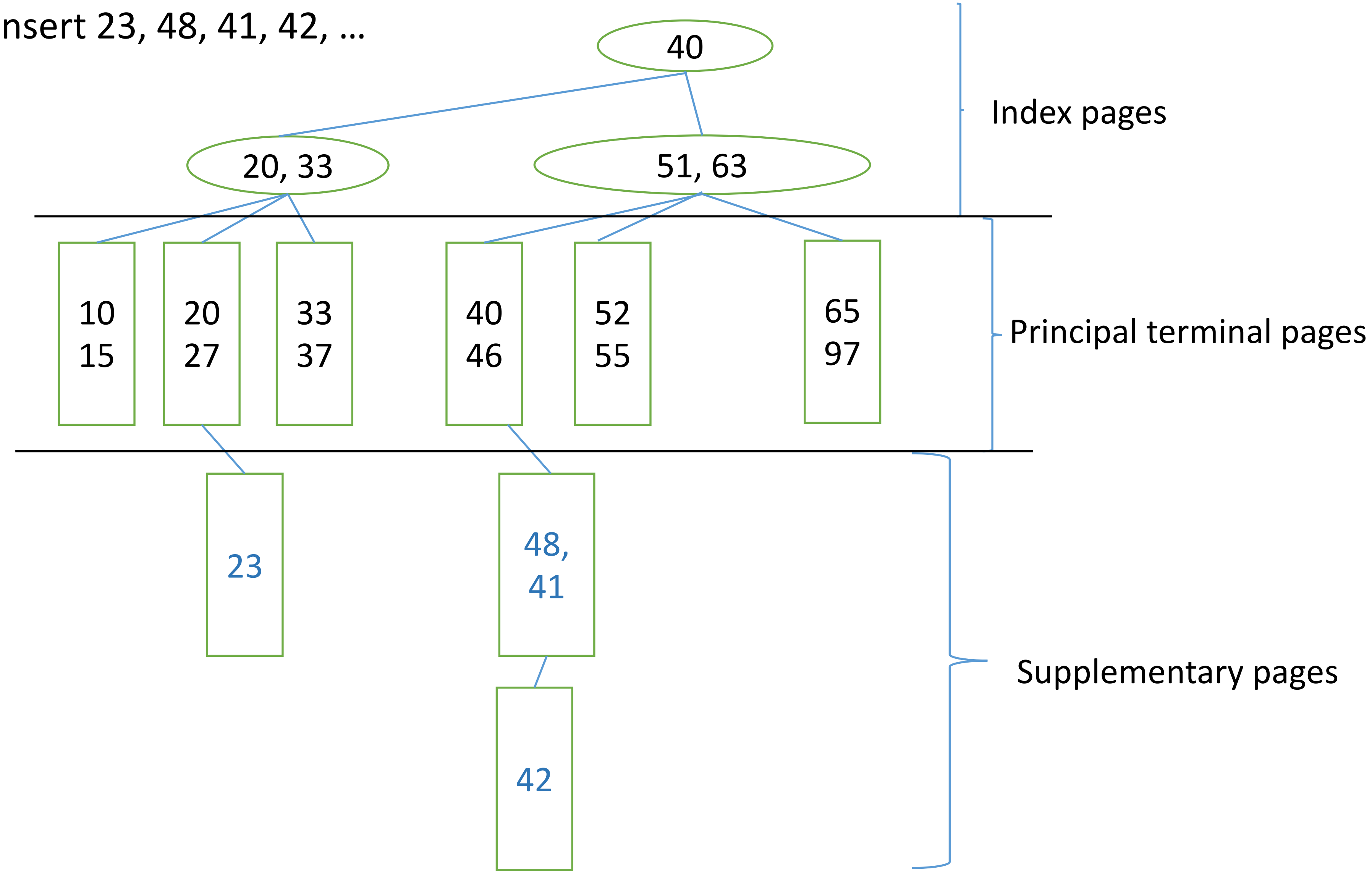




Static tree structure: insert, remove affect the terminal nodes

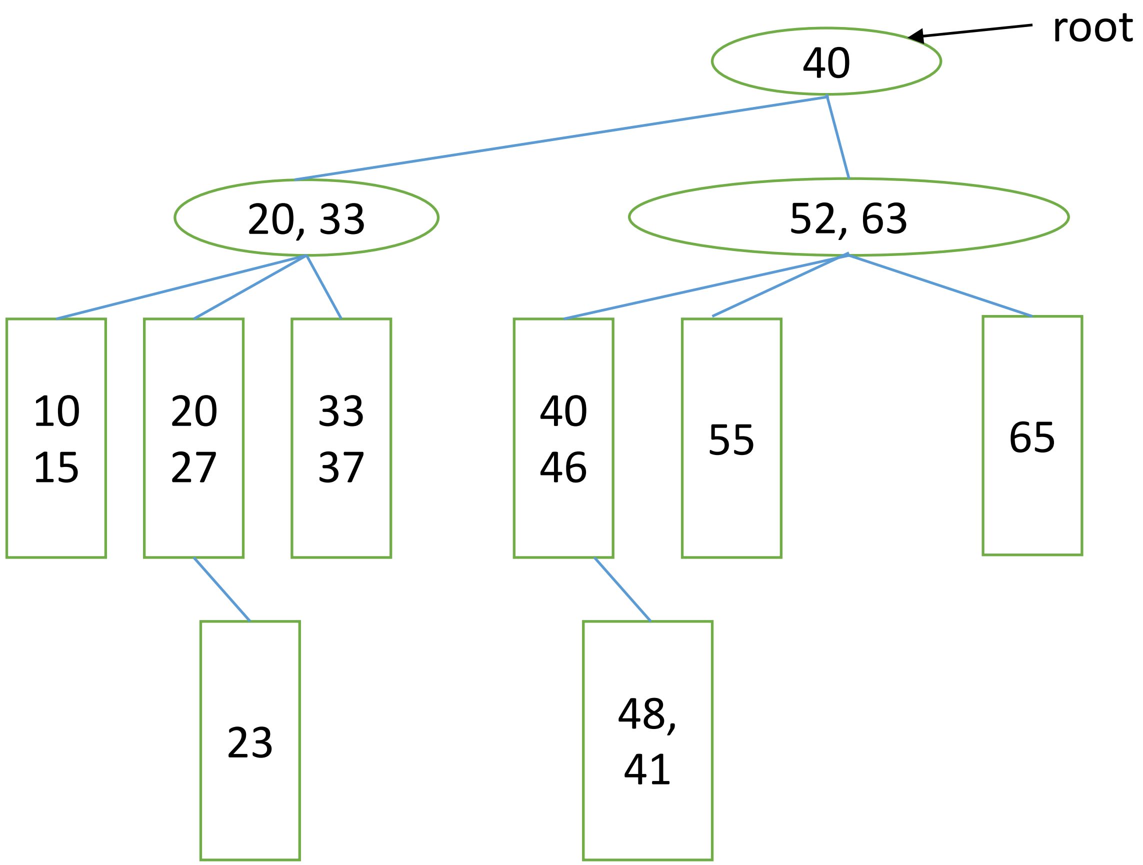
ISAM

- After insert 23, 48, 41, 42, …



ISAM

- After delete 42, 51, 97
- 51 is on the index level, but not in the terminal nodes



Conclusions:

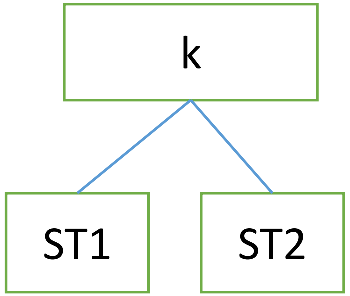
- Are out of balance – as result of insert and delete (non-uniform time)
- The records from the reserved pages are not sorted ussualy (but could be)
- Fast insert / delete (without balanced tree)
- Improved concurrent access (the nodes of the tree are never blocked)

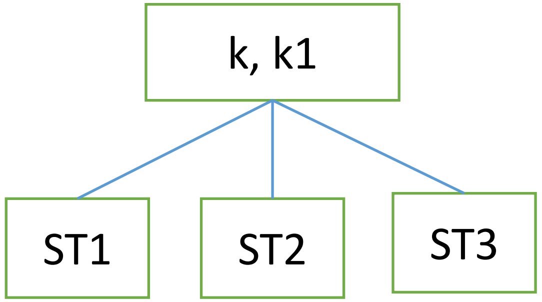
- Useful only for the tables that are modified rarely

2-3 tree

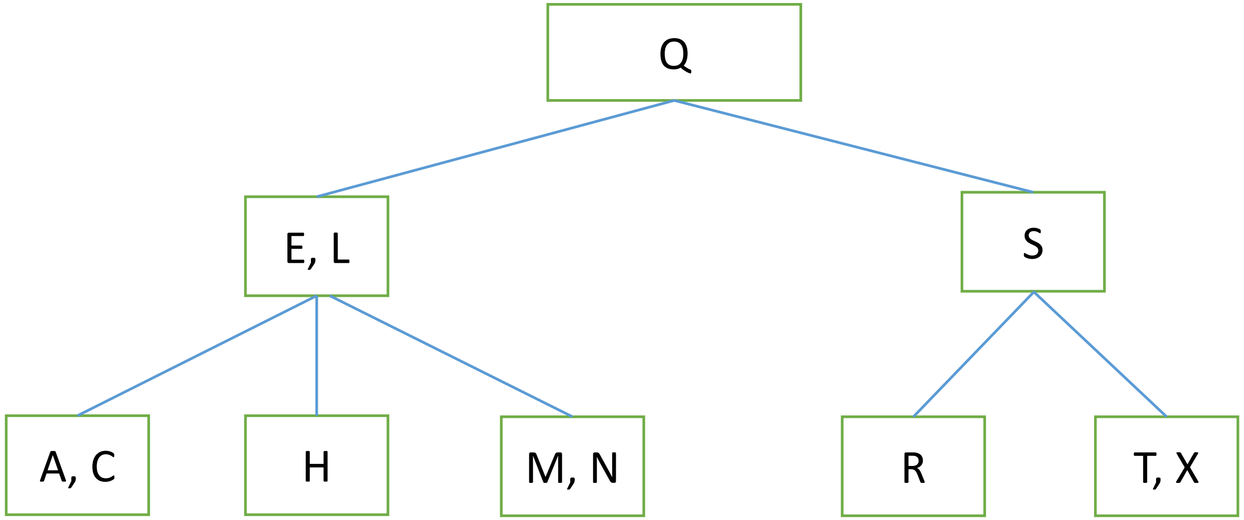
2-3 tree storing key values

- a 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 a value k has 2 subtrees: one with the values less that k and one with the values greater than k
  - a non-terminal node with 2 values k and k1, k<k1, has 3 subtrees: one with values less than k, one with values between k and k1 and one with values greater than k1





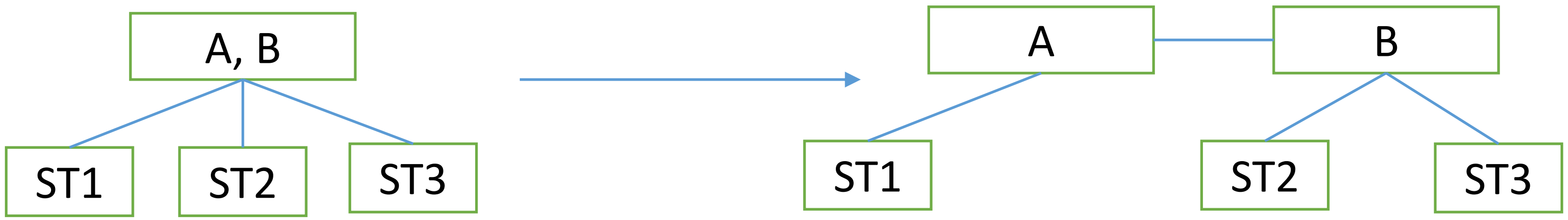
Example: key values are letters



2-3 tree

2-3 tree store

- 2-3 tree storing the values of a key
- tree – key value + address of the record (file/database address of record with corresponding key values)
- 2 possible options:
  - a. transform 2-3 tree into a binary tree: the nodes with 2 values are transformed and the nodes with one value remain unchanged



  - the structure of a node

k	address	Pointer(left)	Pointer(right)	indicator

    - k – key value
    - address – the address of the record with the current key value (address in the file)
    - Pointer(left), Pointer(right) – the 2 subtrees' address (address in the tree)
    - indicator – indicator that specifies the type of the link to the right

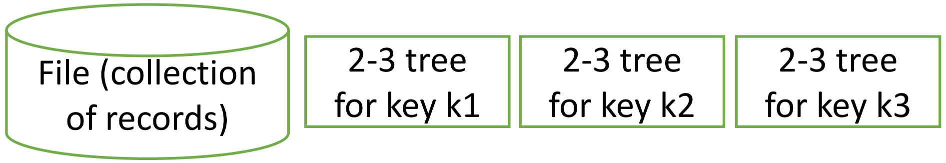
2-3 tree

2-3 tree store

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

NV	k1	address1	k2	address2	Pointer1	Pointer2	Pointer3

  - NV – number of values in the node (1 or 2)
  - k1, k2 – key values
  - address1, address2 – the records' addresses (corresponding to k1 and k2)
  - pointer1, pointer2, pointer3 – the 3 subtrees' addresses
- a file (a relation in a relational database) can have several associated 2-3 trees (one tree / key)

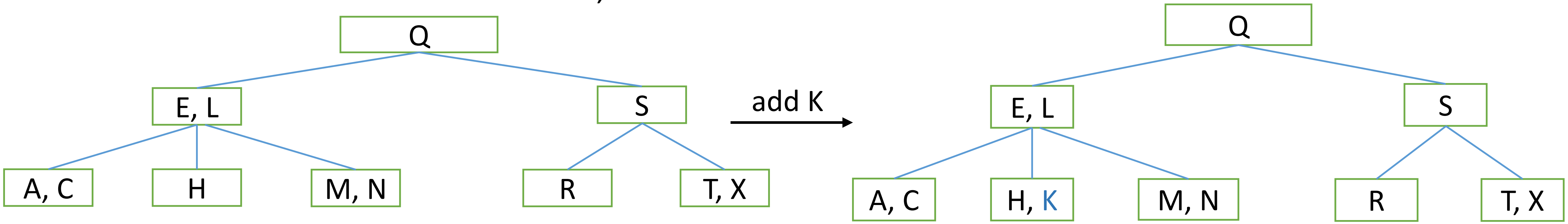


- Operations:
  - search for a record with a key value k
  - insert a record
  - remove a record
  - tree traversal (partial, total)

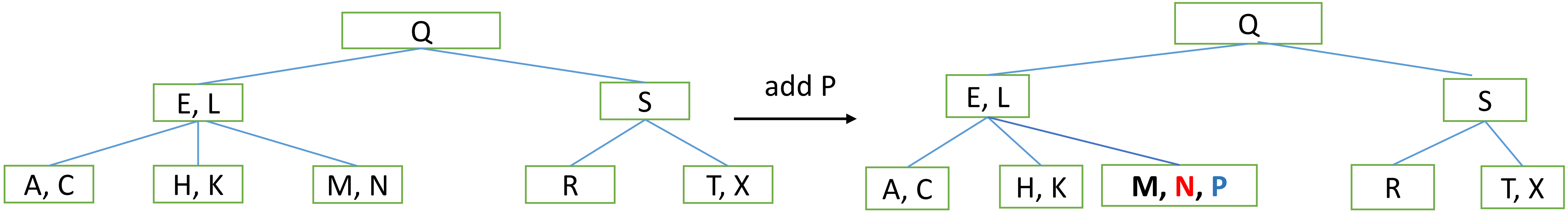
2-3 tree

Add a new value

- 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



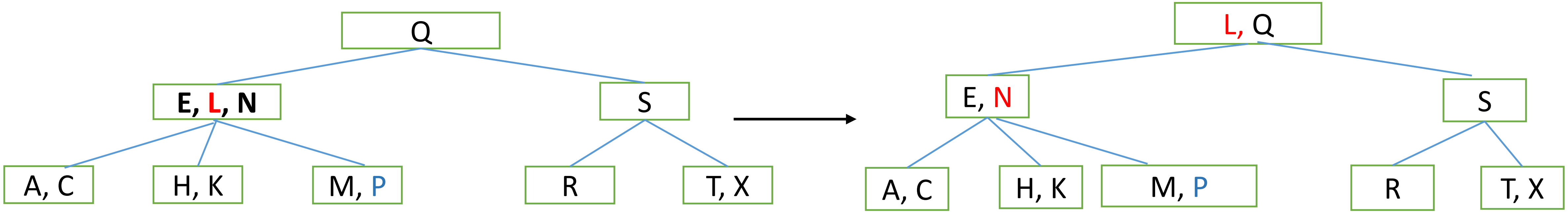
- if the reached terminal node has 2 values, the new value is added to the node, the 3 values are sorted, the node is split into 2 nodes: one node will contain the smallest value, the 2nd node – the largest value, and the middle value is attached to the parent node; the parent is then analysed in a similar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



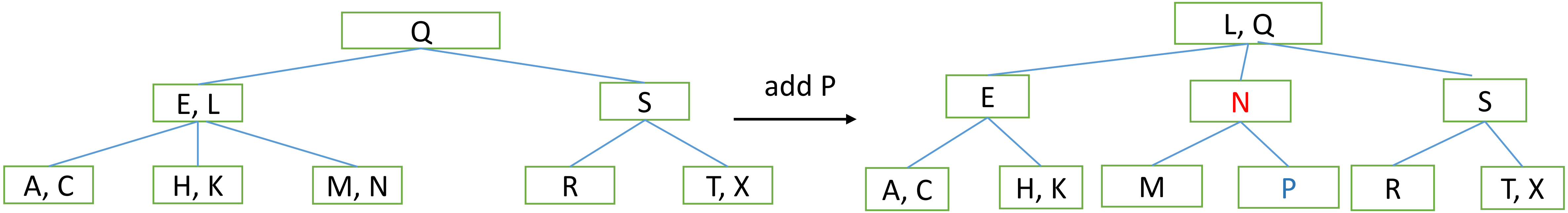
2-3 tree

Add a new value

- 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 analysed in a similar manner



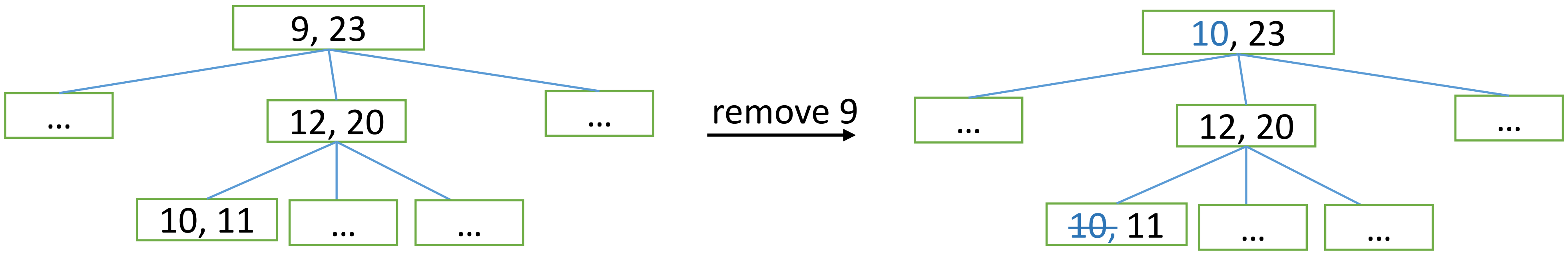
Because, N > L, and (L, Q) is the node that should have 3 subtrees, follows that the initial 2-3 tree becomes:



2-3 tree

Delete a value k

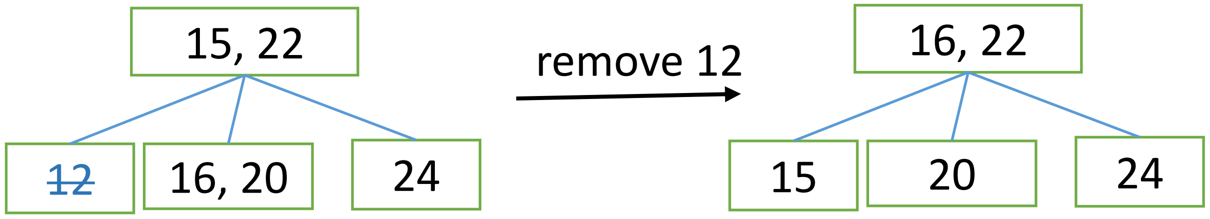
- 1. search for k; if k appears in an inner node, change it with a neighbour value k1 from a terminal node (there is no other value between k and k1); k1 previous position (in the terminal node) is eliminated

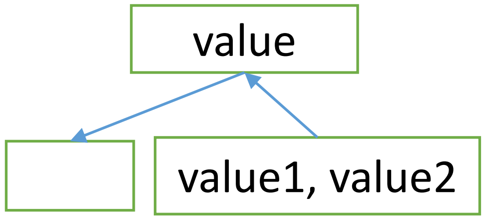


- 2. perform this step until case a / b occurs

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



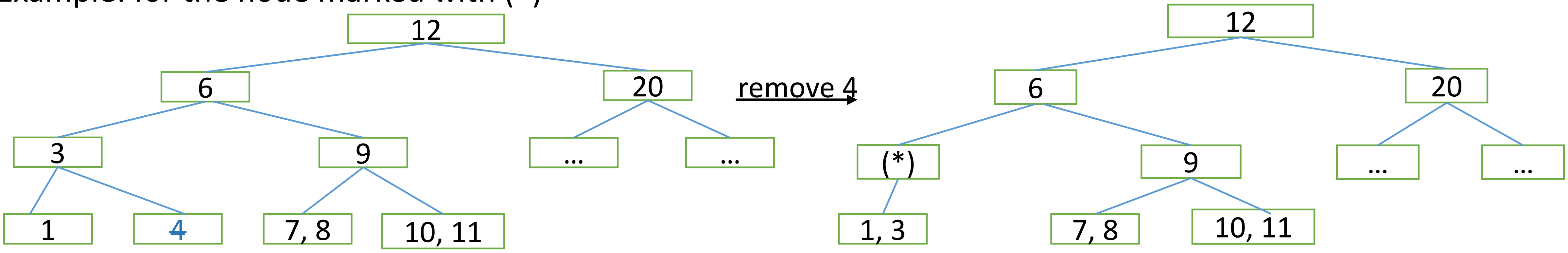


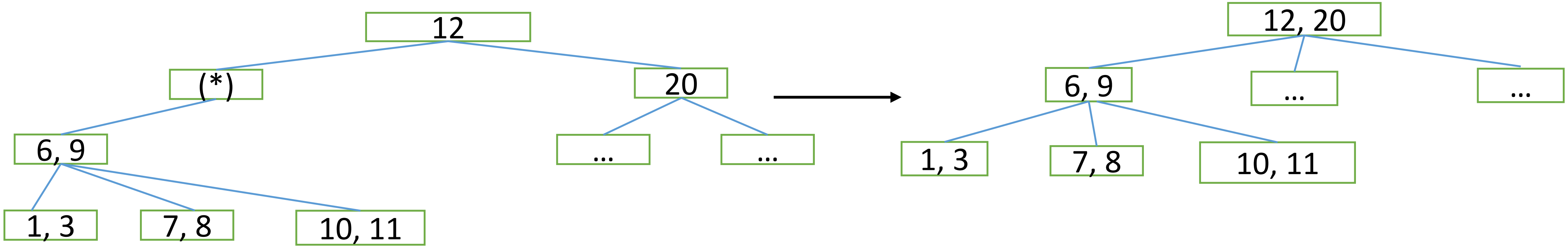
2-3 tree

Delete a value k

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 analysed for the parent; if the root is reached and it has no values, it is eliminated and the current node becomes the root

Example: for the node marked with (*)

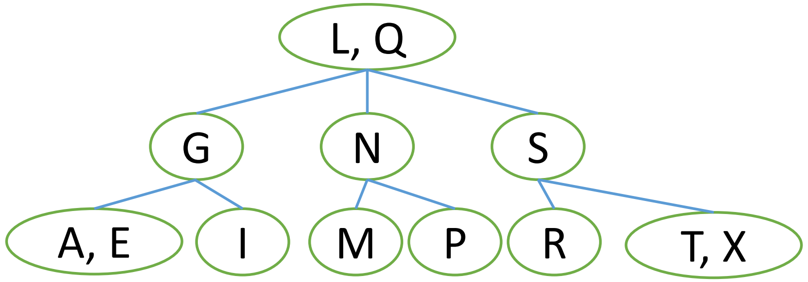




B tree

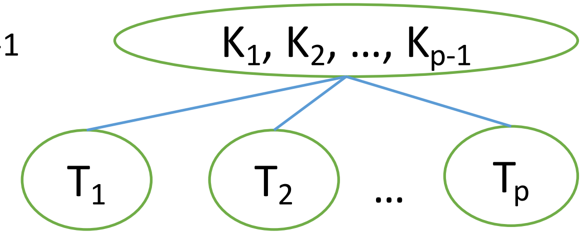
B-tree (B – balanced, B – broad)

- the most popular method to organize the indexes in databases
- ordered tree; a node has multiple subtrees; a node has keys and pointers to subtrees; the ways from the root to terminal nodes (leaf) have the same length



Generalization of 2-3 trees

- B-tree of order m
  - if the root is not a terminal, it has at least 2 subtrees
  - every non-terminal node – at least $m/2$ subtrees (exception: the root)
  - all terminal nodes have the same level
  - every non-terminal node has at most m subtrees
  - a node with p subtrees has p-1 ordered values (ascending order): $K_1<K_2<\ldots K_{p-1}$
    - T_1: values less than K_1
    - T_i: values between K_{i-1} and K_i where i=1, 2, …, p-1
    - T_p: values greater than K_{p-1}

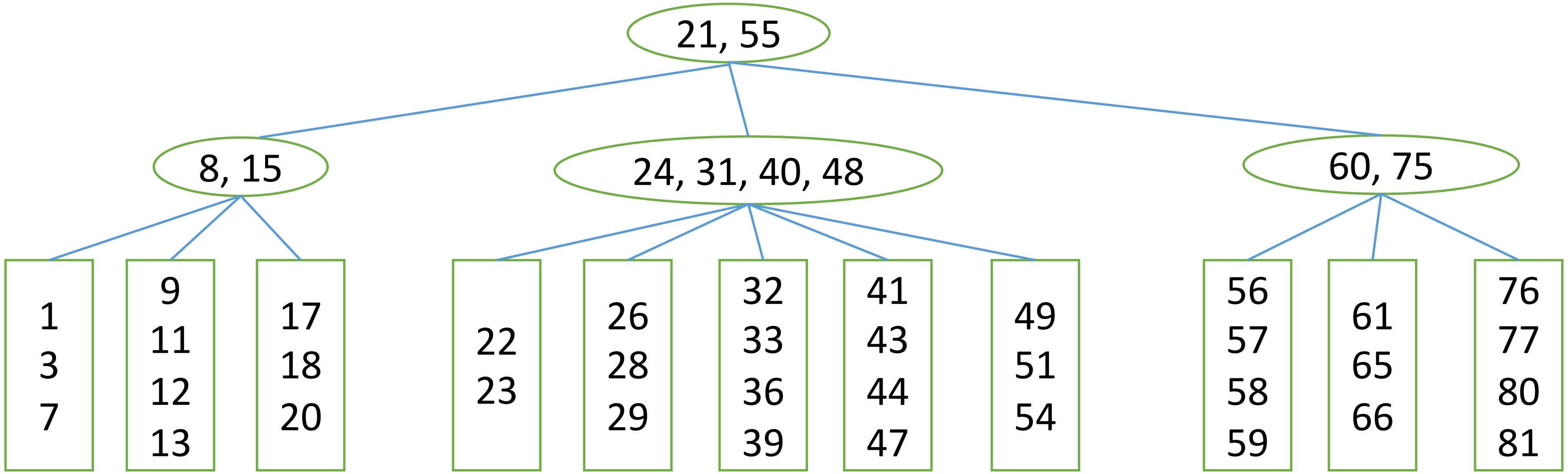


  - limits on number of subtrees (and values) / node result from the manner in which the inserts / deletes are performed so that the second requirement in the definition is met

B tree

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

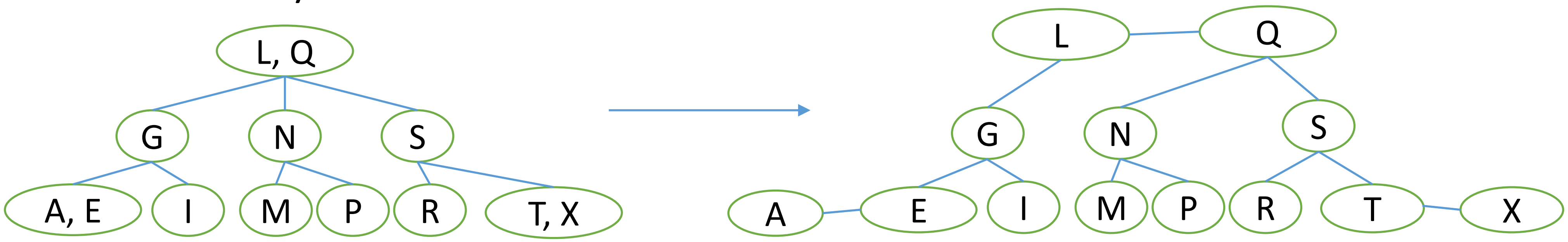
- store the values of a key (a database index)
- the tree contains: key values + address of the record
- 1. Transformed into a binary tree
    - pointer(left) – reference the first key of the left subtree from the B-tree
    - pointer(right) – reference the right neighbour in the node of the B-tree (reference the first key of the right subtree from the B-tree – if is the last value of the key in the node of the B-tree)
    - additional flag to the right pointer

k	data value	pointer(left)	pointer(right)/h

B tree

B-tree of order m

1. Transformed into a binary tree



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

- can be stored m-1 keys per node
- NV – number of values in the node
- K_i – key values, i=1, …, m-1
- $address_i$ – the records' address (corresponding to the key's values)
- $pointer_i$ – subtree address

NV	k_1	$address_1$	…	$address_{m-1}$	$pointer_1$	…	$pointer_m$

Operations:

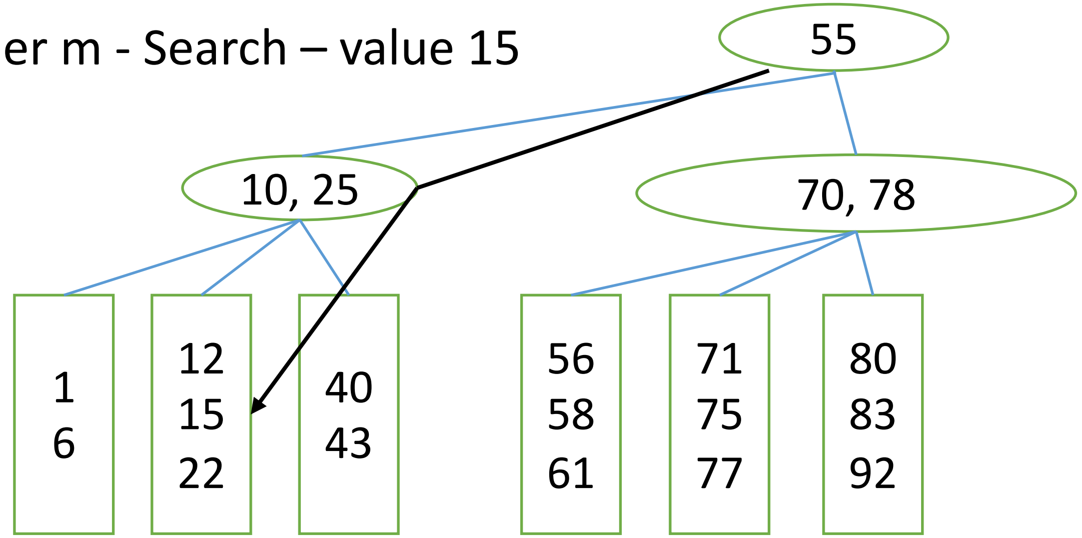
- search for a value
- add a value
- remove a value
- tree traversal (partial, total)

Memory space usage for pointers for store the terminal nodes:

- m/2<=NV<=m-1 – for nodes
- m/2<=NV<=m – for terminal nodes
- additional flag / sign for NV

B tree

B-tree of order m - Search – value 15



B-tree of order m - Insert

- 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 less than m-1 values, the new value can be stored in the node

Steps:

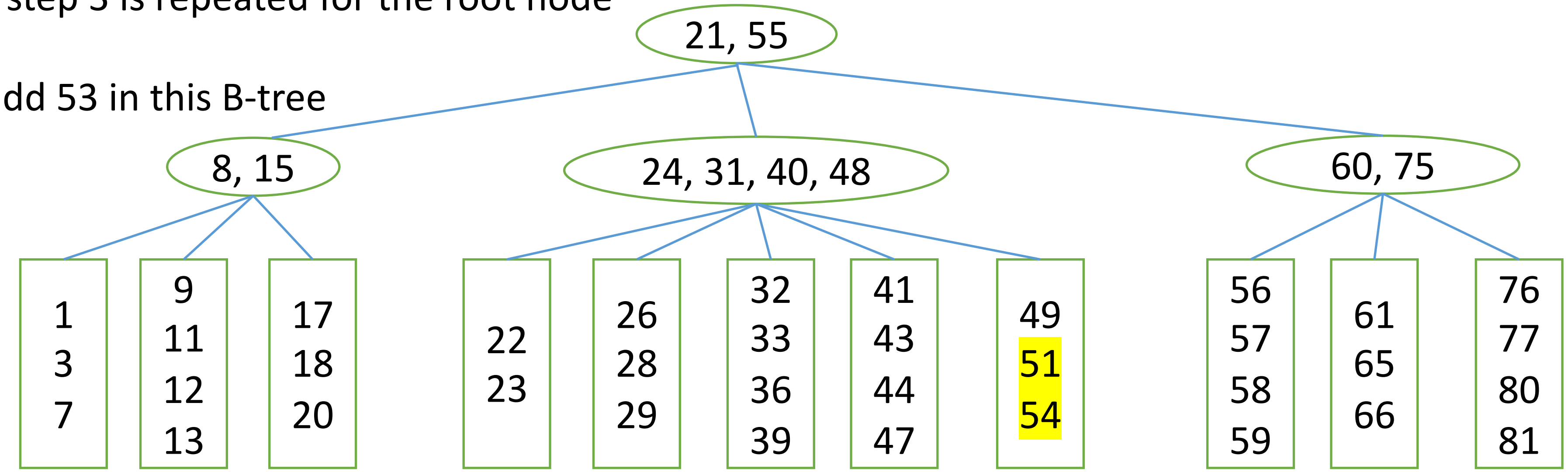
- search the node in which the key should be inserted
- insert the new key
- apply a balance procedure in case in which the maximum number of keys that can be stored is overstepped

B tree

B-tree of order m – Insert algorithm

1. Search the node to insert
2. Insert key
3. If the node is full (the dimension is overstepped)
    a. A new node is created and are moved there the keys that are greater than the middle value of the key
    b. The middle key is inserted in the parent node
    c. The right pointer of the key will reference the new node and the left pointer of the key will reference the old node that contains the smaller values
4. If the parent node is also full
    a. If the parent node is the root then is created a new root
    b. The step 3 is repeated for the root node

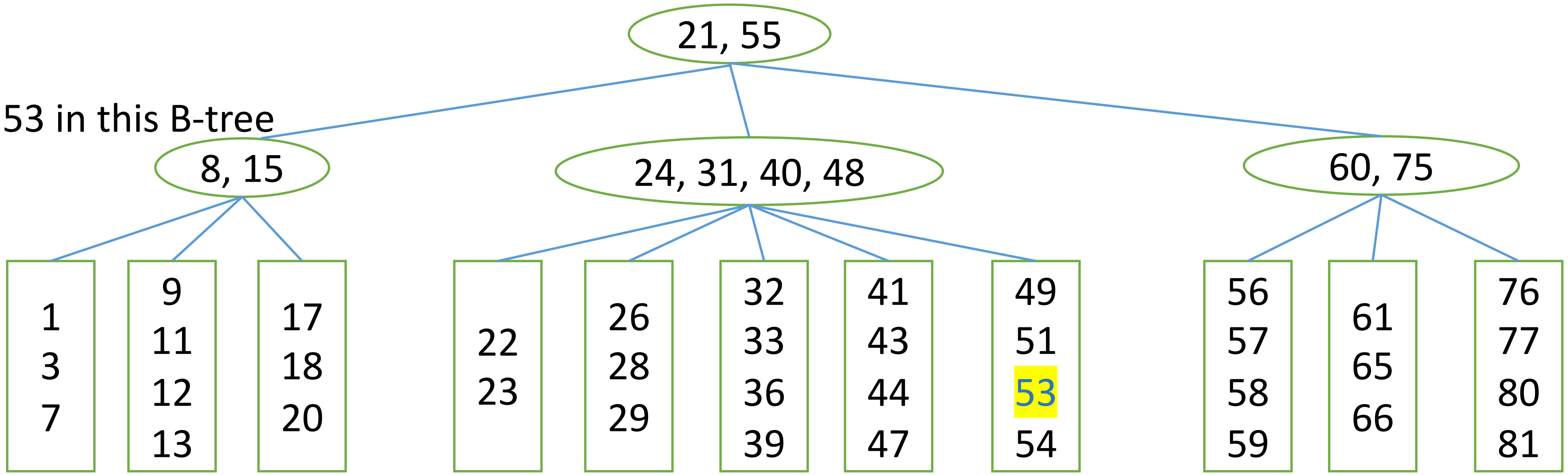
Example 1: add 53 in this B-tree



B tree

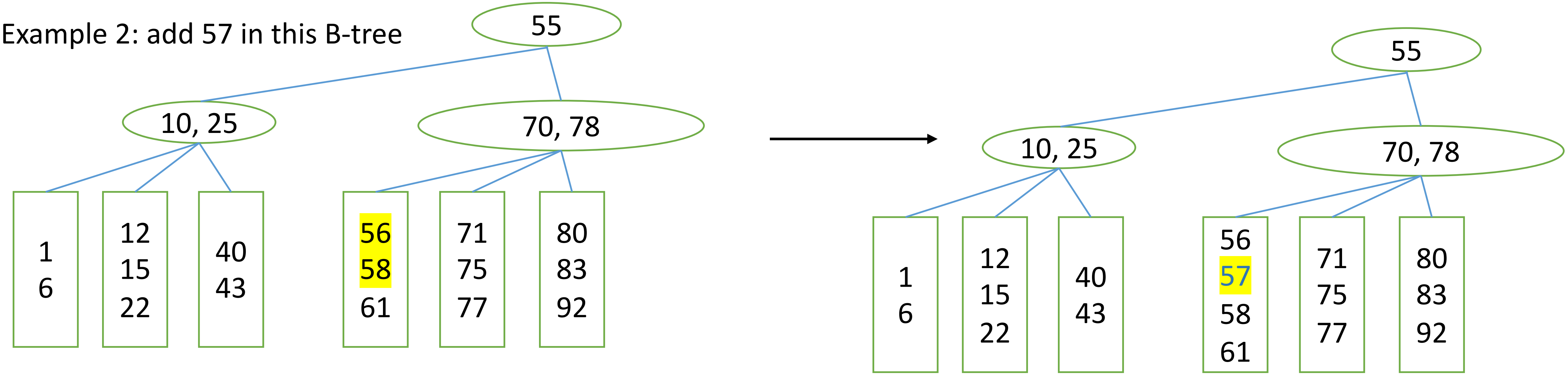
B-tree of order m

Example 1: added 53 in this B-tree



53 was inserted, because that node can store at most 4 values

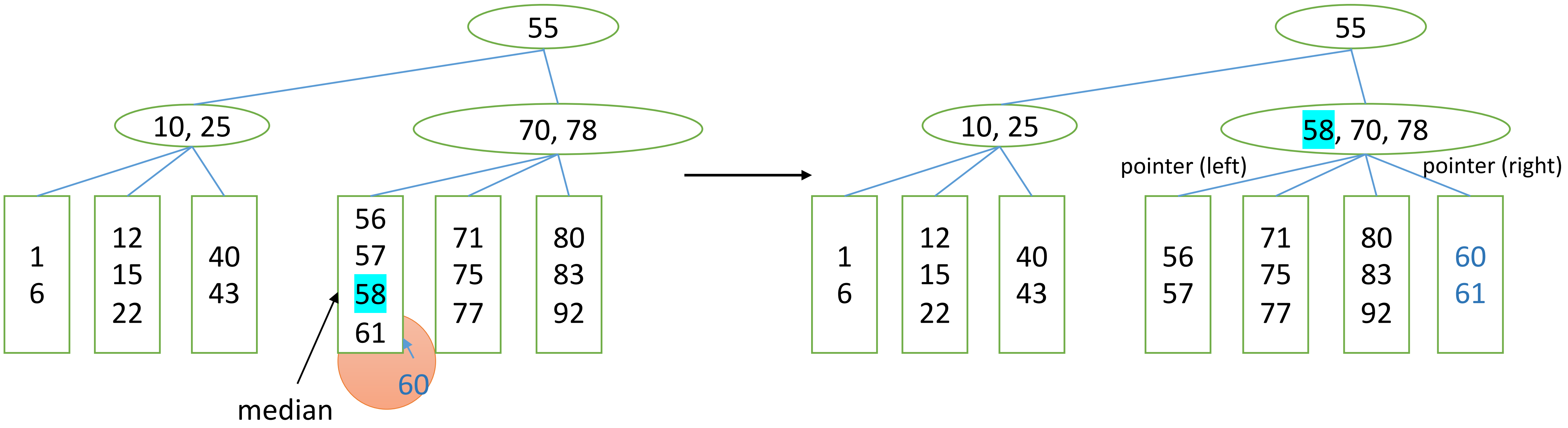
Example 2: add 57 in this B-tree



B tree

B-tree of order m

Example 2: add 60 in this B-tree

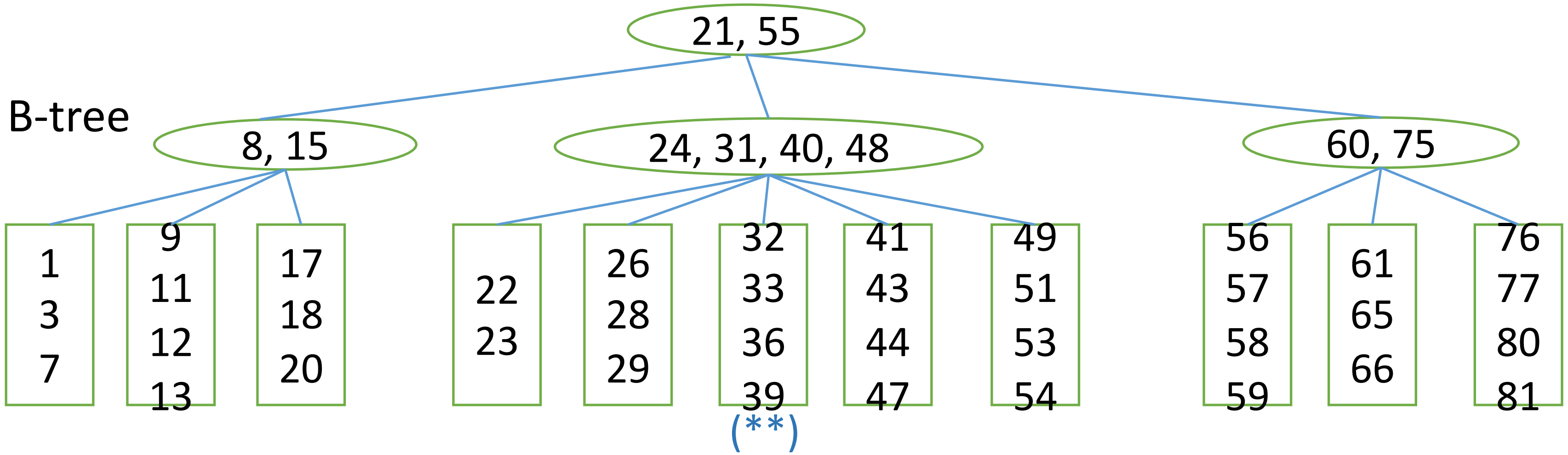


on point 3. if the terminal node already has m-1 values, the new value is attached to the node, the m values are sorted, the node is split into 2 nodes, and the middle value (median) is attached to the parent node; the parent is then analysed in a similar manner

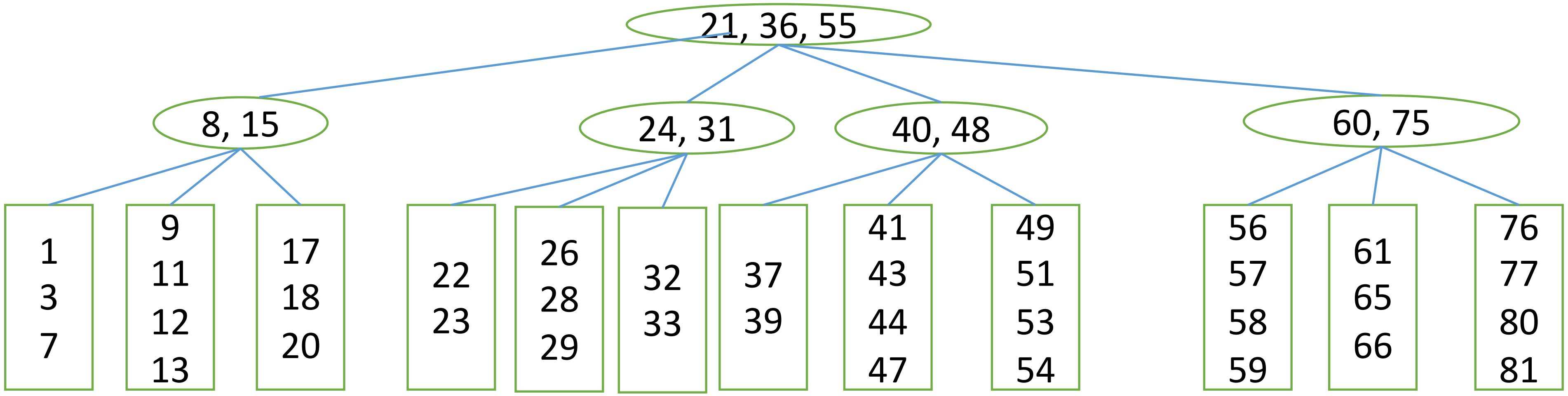
B tree

B-tree of order m

Example: add 37 in this B-tree



- the node marked with (**) should contain 32, 33, 36, 37, 39
- since the node's capacity is exceeded, it is split into nodes *32, 33*, and *37*, 39, and 36 is attached to the parent node (with values *24, 31, 40, 48*)
- in turn, the parent must be split into 2 nodes (values *24, 31*, and *40, 48*), and 36 is attached to its parent

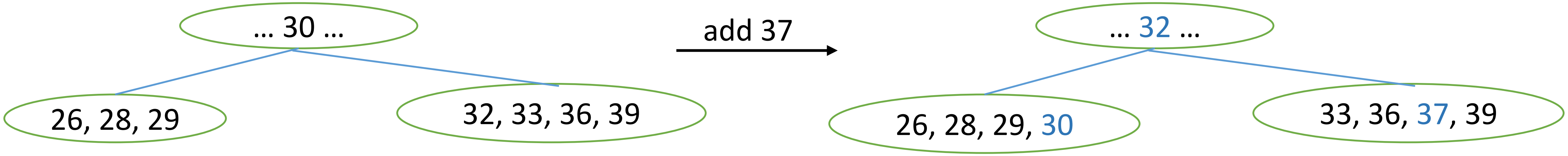


B tree

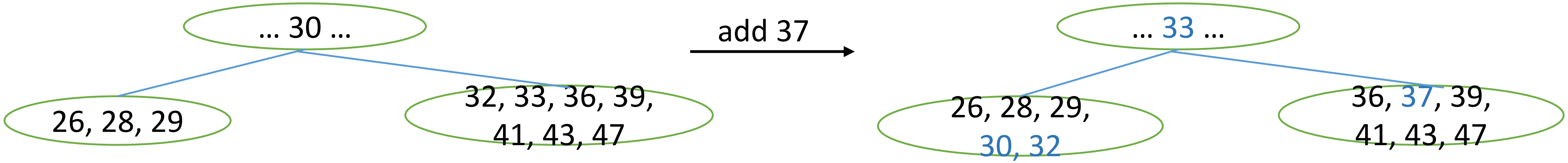
B-tree of order m - Optimization (insert)

- before performing a split - analyse whether one or more values can be transferred from the current node (with m-1 values) to a sibling node

Example: B-tree of order 5 (non-terminal node - between 2 and 4 values, i.e. between 3 and 5 subtrees):



Example: B-tree of order 8 (non-terminal node - between 3 and 7 values, i.e. between 4 and 8 subtrees):



B tree

B-tree of order m – Remove a value

- identify the node that will be removed
- if it is a non-terminal node, a value from the terminal nodes is transferred (instead)
- in case of undersize, a redistribution or a concatenation is perfect
- a node can have at most m subtrees (i.e. maximum m-1 values) and at least [m/2] subtrees (i.e. at least [m/2]-1 = [(m-1)/2] values)
- when eliminating a value from a node, an underflow can occur (the node can end up with less values than the required minimum)

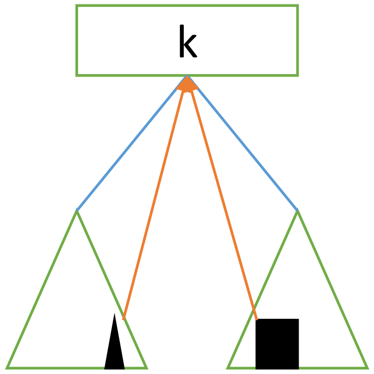
Eliminate value k:

1. search for k; if it does not exit, the algorithm ends
2. if k is found in a non-terminal node (see figure), k is replaced with a *neighbour value* from a terminal node (this value can be chosen between 2 values from the trees separated by k)
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

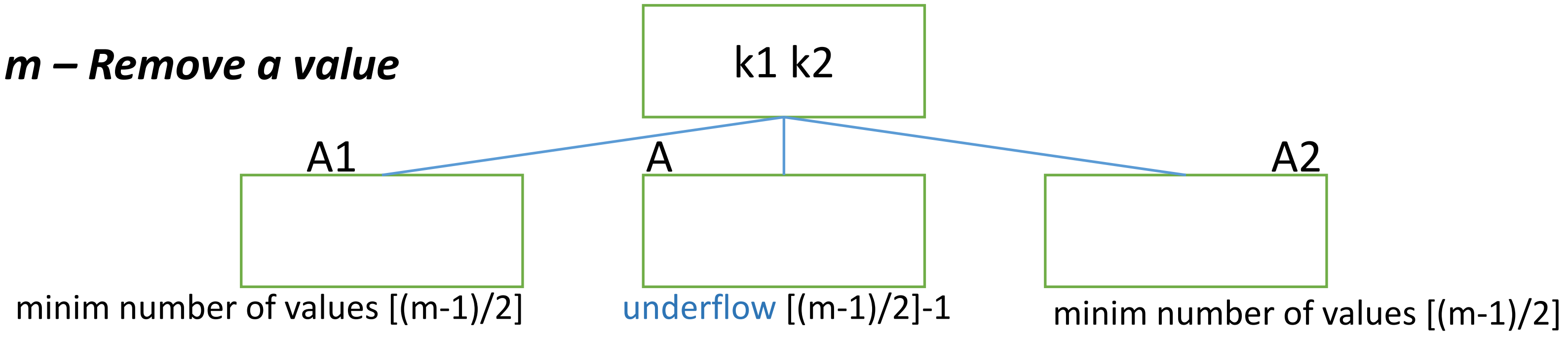
b. if the delete operation causes an underflow in the current node (A), but one of the sibling nodes (left / right - B) has at least 1 extra value, values are transferred between A and B via the parent node; the algorithm ends

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

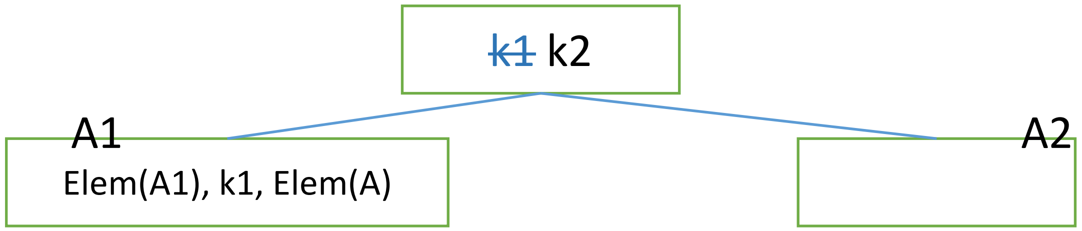


B tree

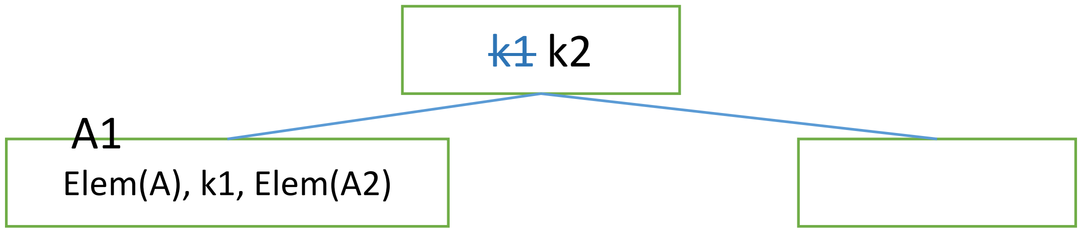
B-tree of order m – Remove a value



- 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 value K1 (separating A from A2); the node at address A2 is deallocated



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

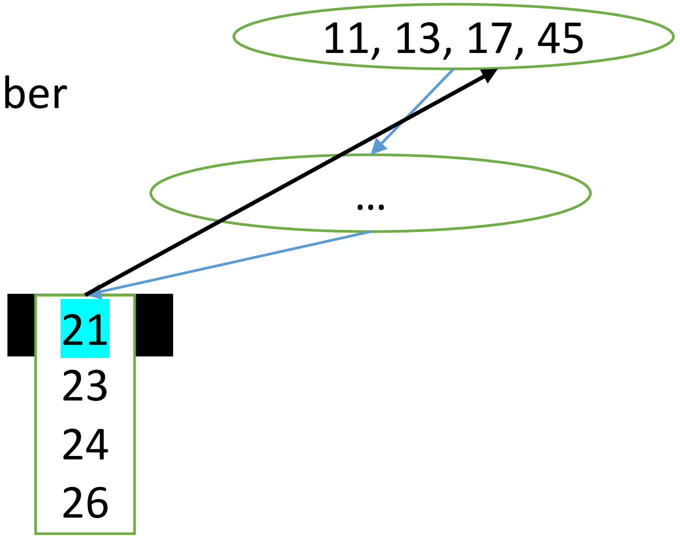
B tree

B-tree of order m – Remove algorithm

1. Search the value that should be removed

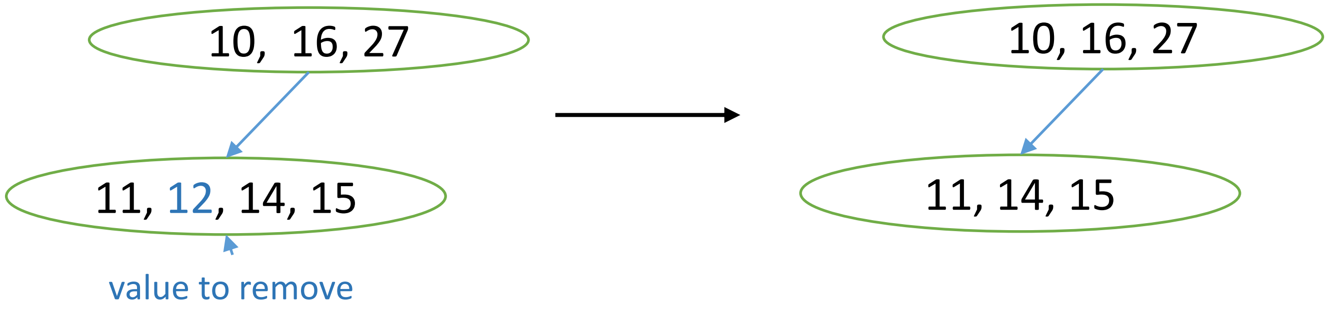
If it is in a non-terminal node, it is replaced by a neighbour greater value (i.e. the left value of the left terminal node of the right subtrees)

2. This step is repeated until until the cases a or b happens



a. If the node that contains the value that should be deleted is the root, or the number of the left values is the node >=[m/2]

- the value is removed
- the values and the pointers form the node are rearranged
- the algorithm is ending

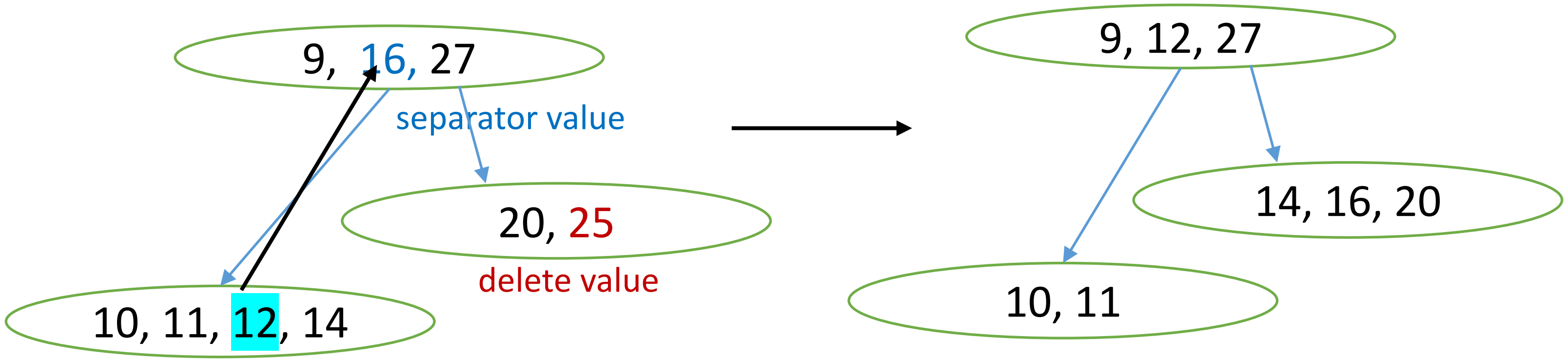


b. If the number of the values that are left in the node < [m/2], one of the neighbour nodes contains > [m/2] values, then **redistribution**

- the values from both nodes and the separator value from the parent are ordered
- the medium value is chosen and added to the parent node, and the other values are inserted in the left / right node
- algorithm ends

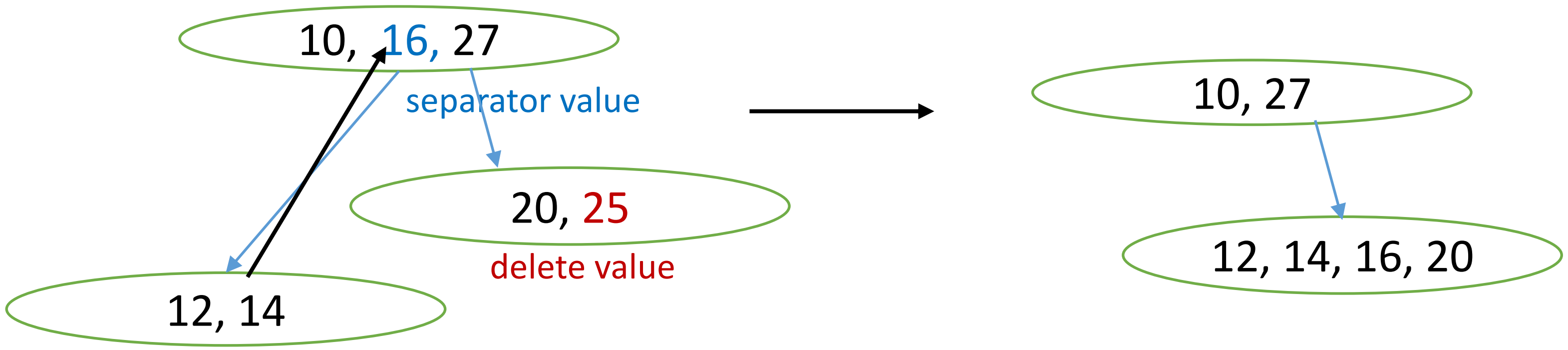
B tree

B-tree of order m – Remove algorithm



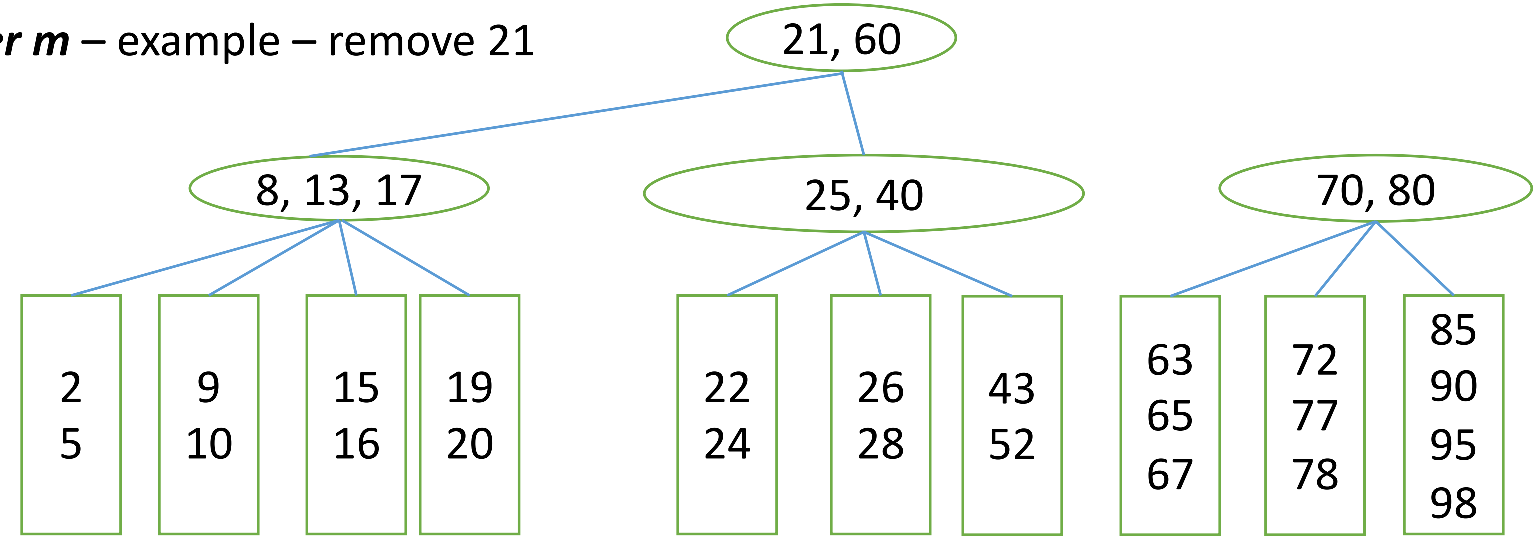
c. If the sum of the values from the node from where the value was removed and of the values of a neighbour is < m, a **concatenation** is performed

- the values of both nodes and the separator value from the parent node are inserted in a single node
- the step 2 is repeated for the parent node (from which had been removed the separator value)
- if the parent node is the root and does not have values, then the current node becomes the root

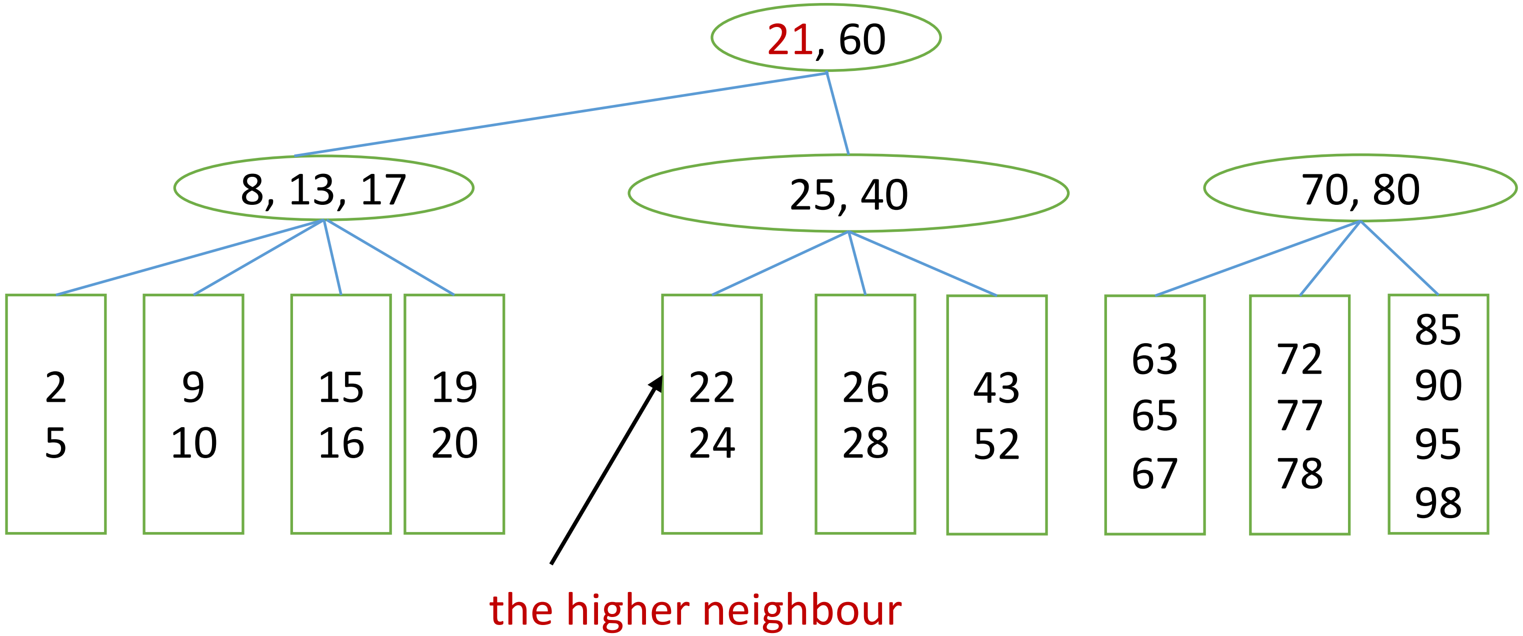


B tree

B-tree of order m – example – remove 21

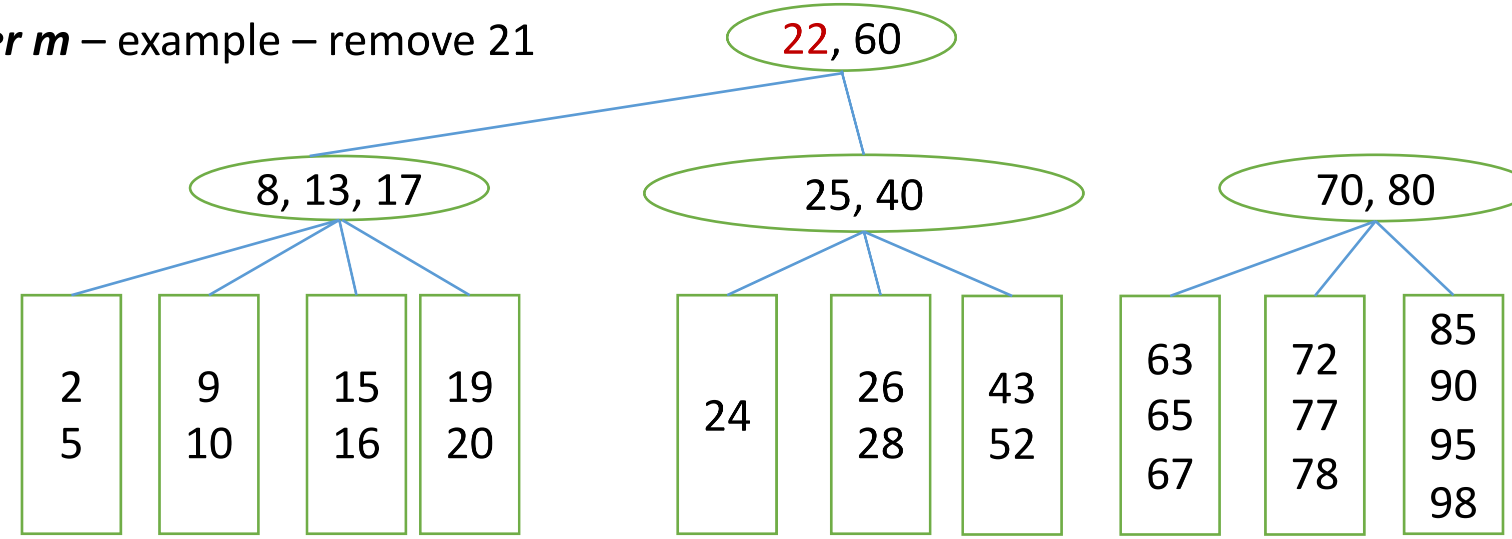


the higher neighbour - 22

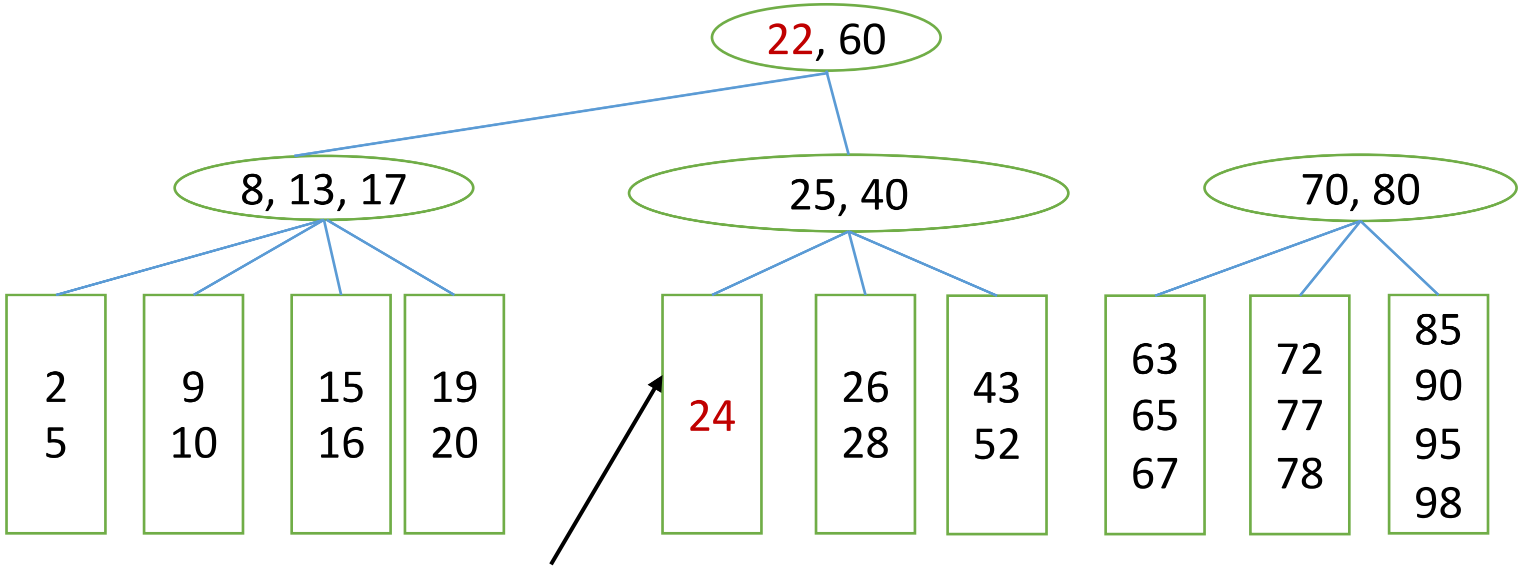


B tree

B-tree of order m – example – remove 21



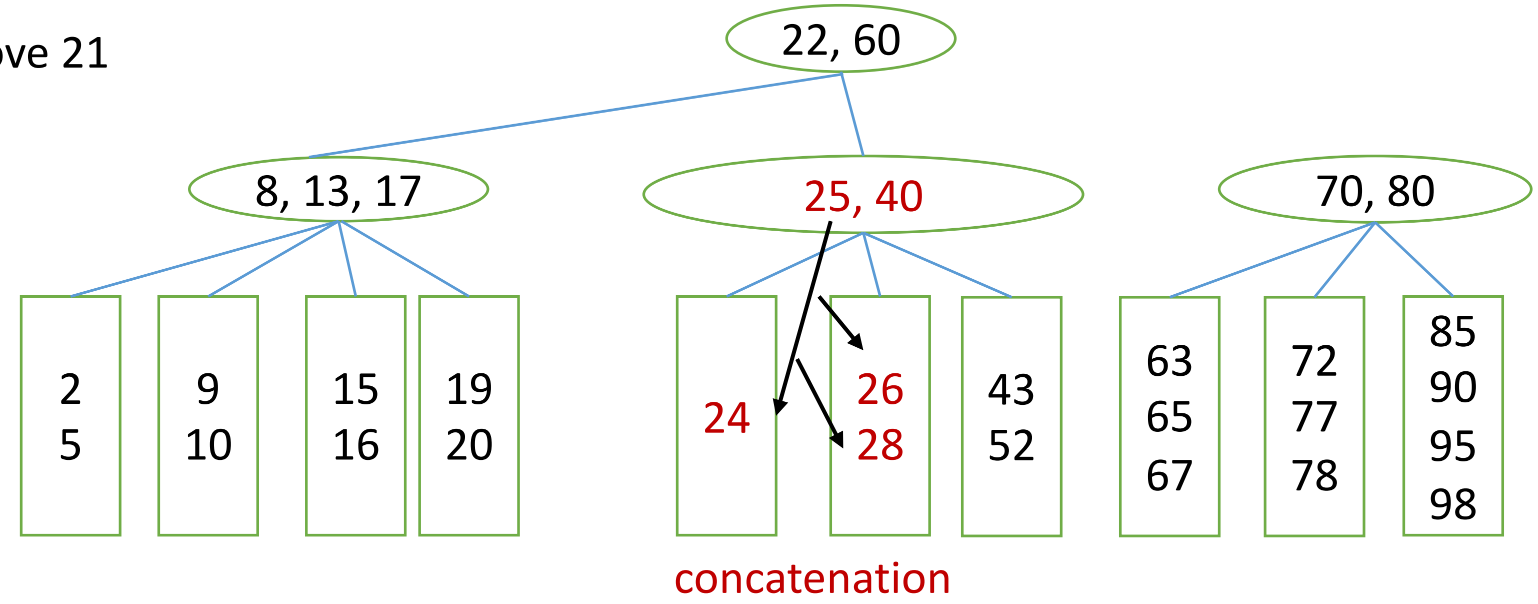
subdimension



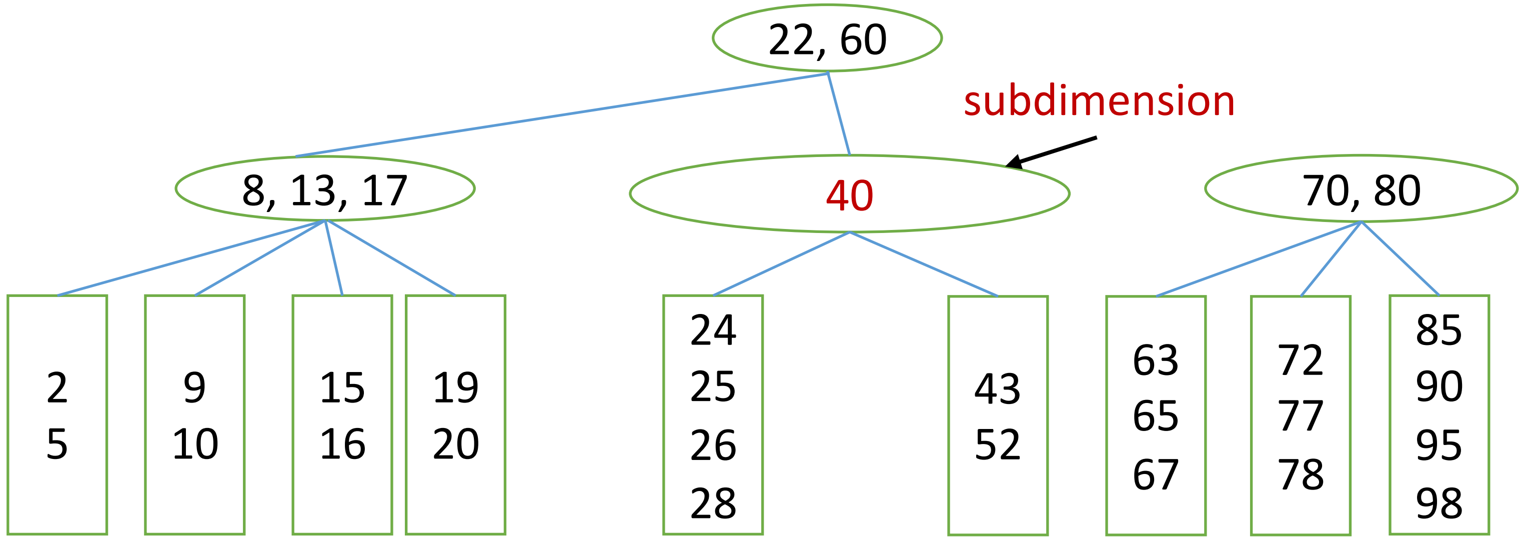
subdimension

B tree

B-tree of order m – example – remove 21

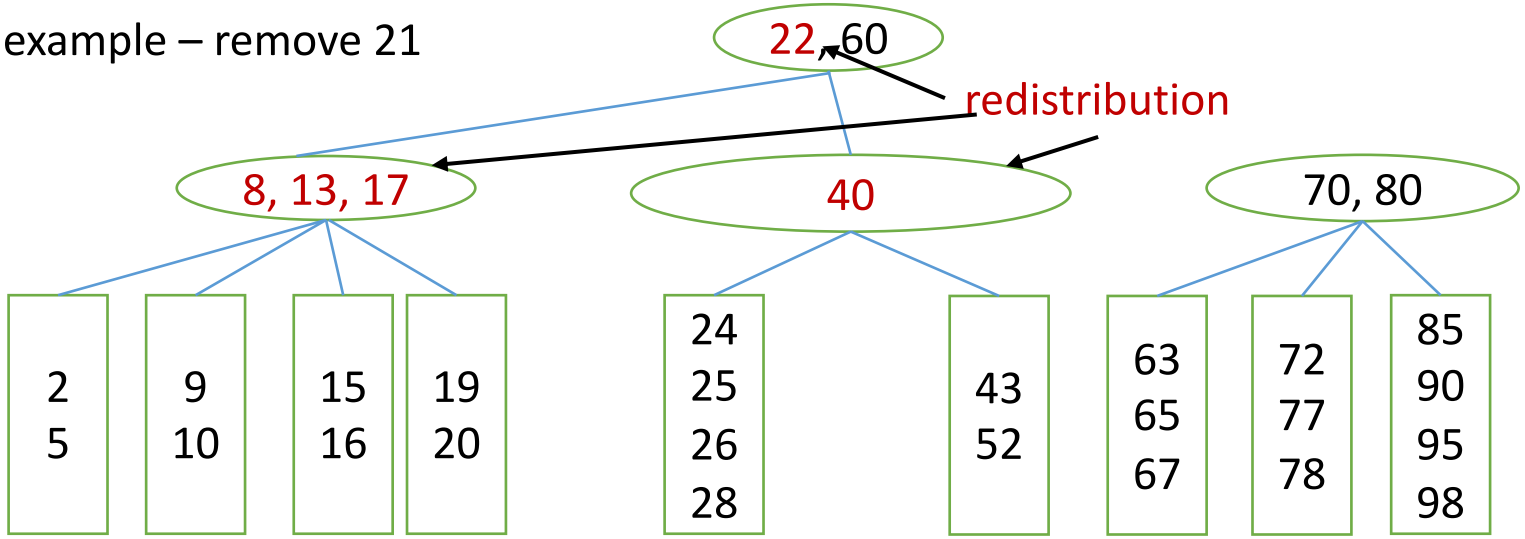


subdimension

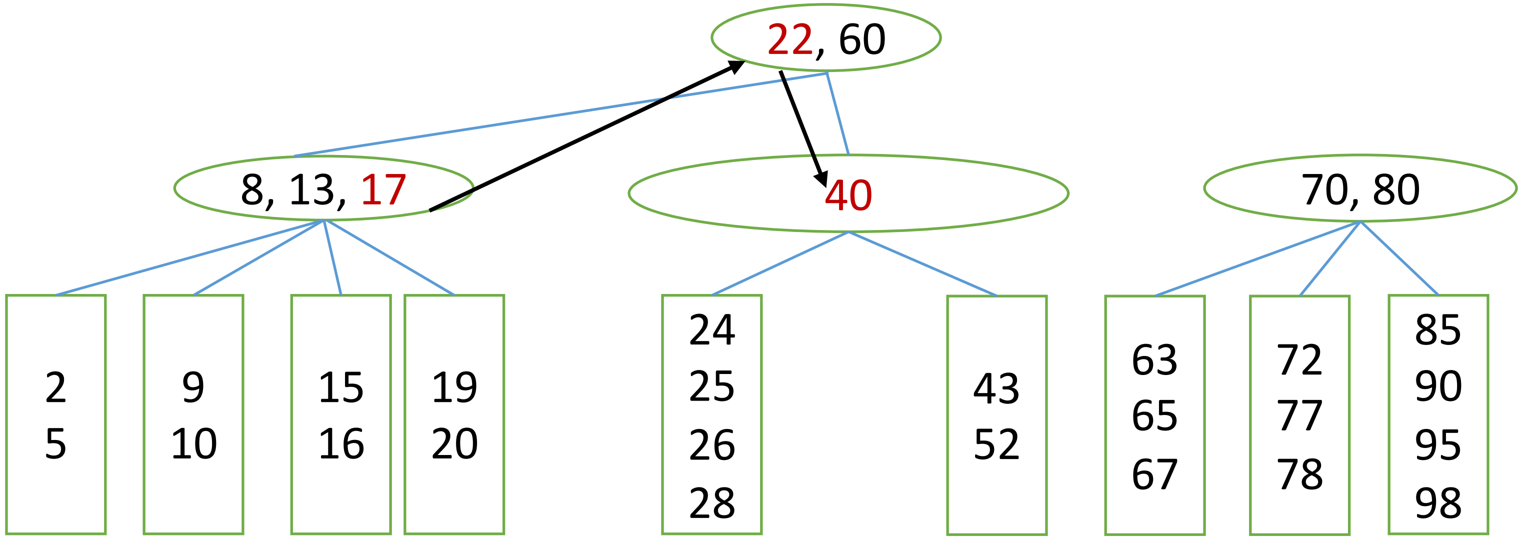


B tree

B-tree of order m – example – remove 21

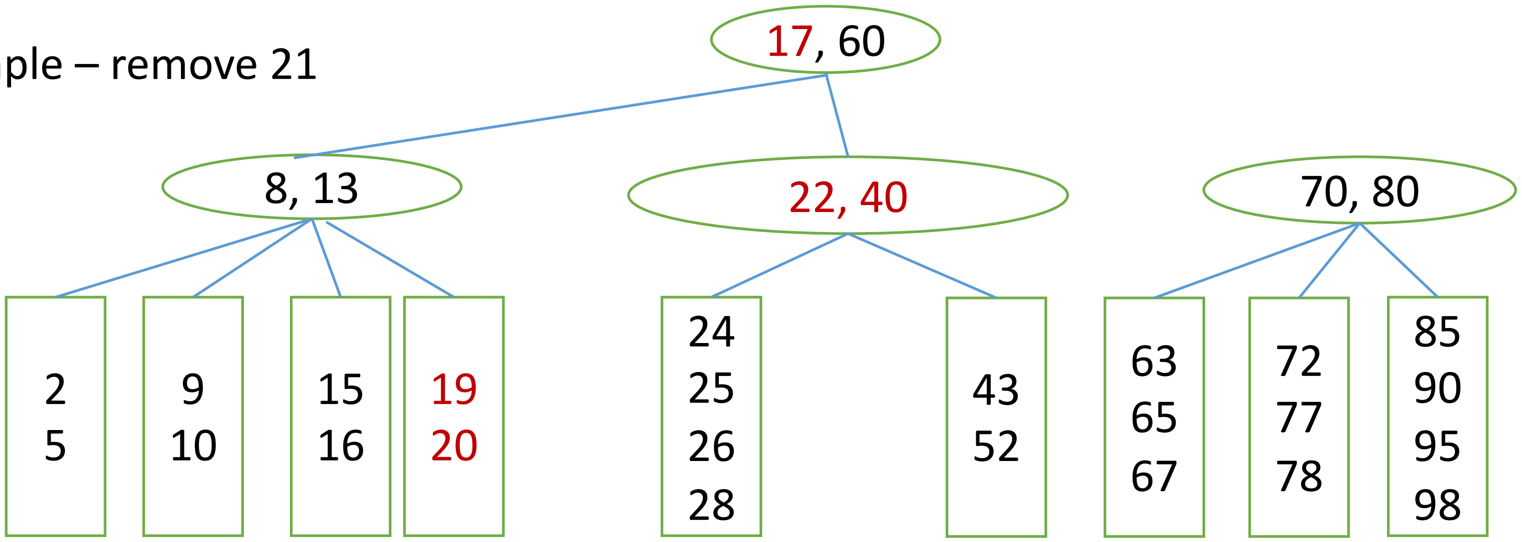


subdimension

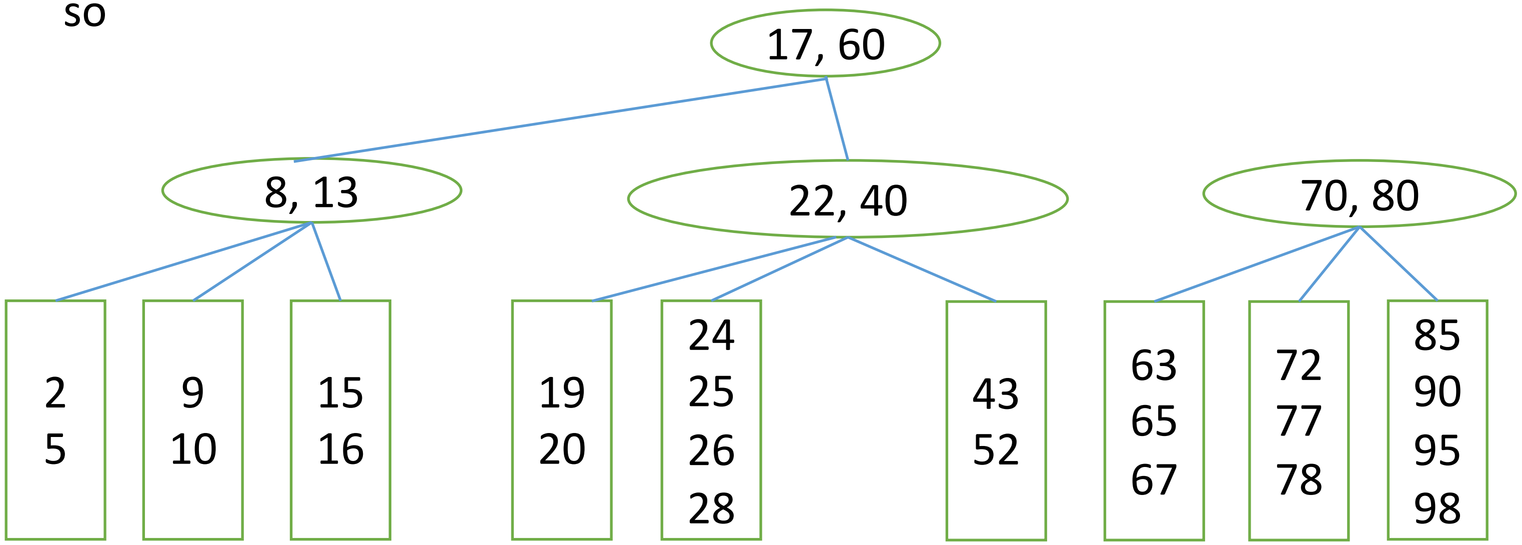


B tree

B-tree of order m – example – remove 21



so



B tree

B-tree of order m - A block stores a node from a B-tree

Example:

- 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

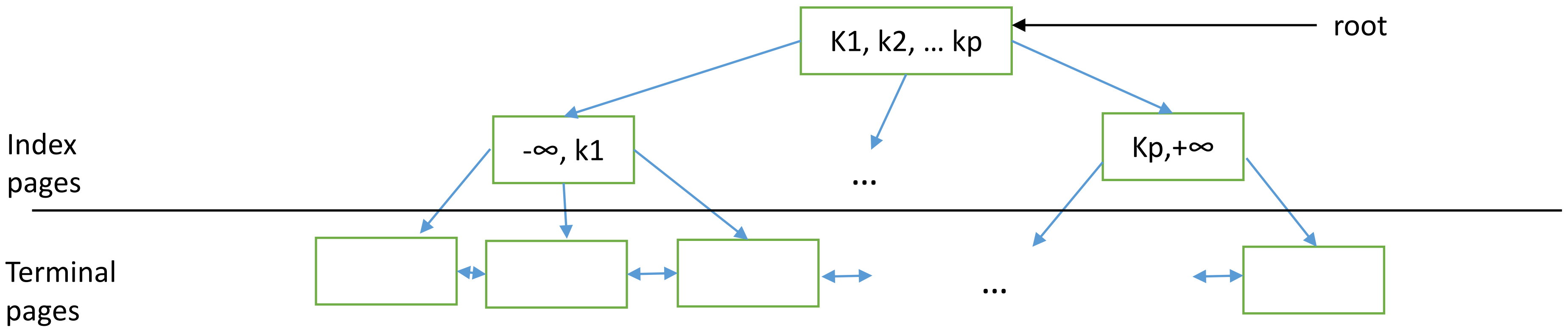
- if the size of a block is 2048b and the other values are unchanged, then the order of the tree is m = 68, i.e., a node can have between 33 and 67 values
- the maximum number of required blocks (from the file that stores the B-tree) when searching for a value - the maximum number of levels in the tree; 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

=> 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

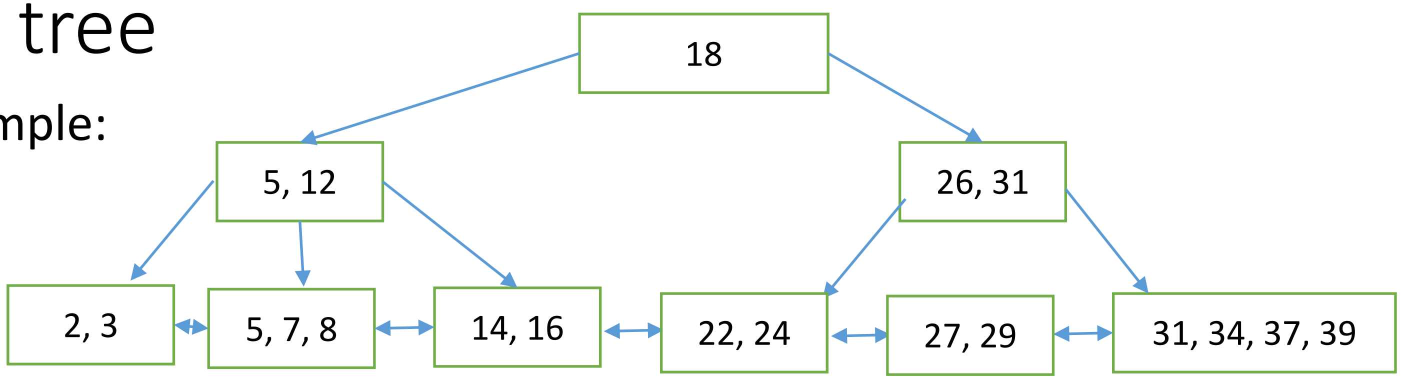
B+ tree

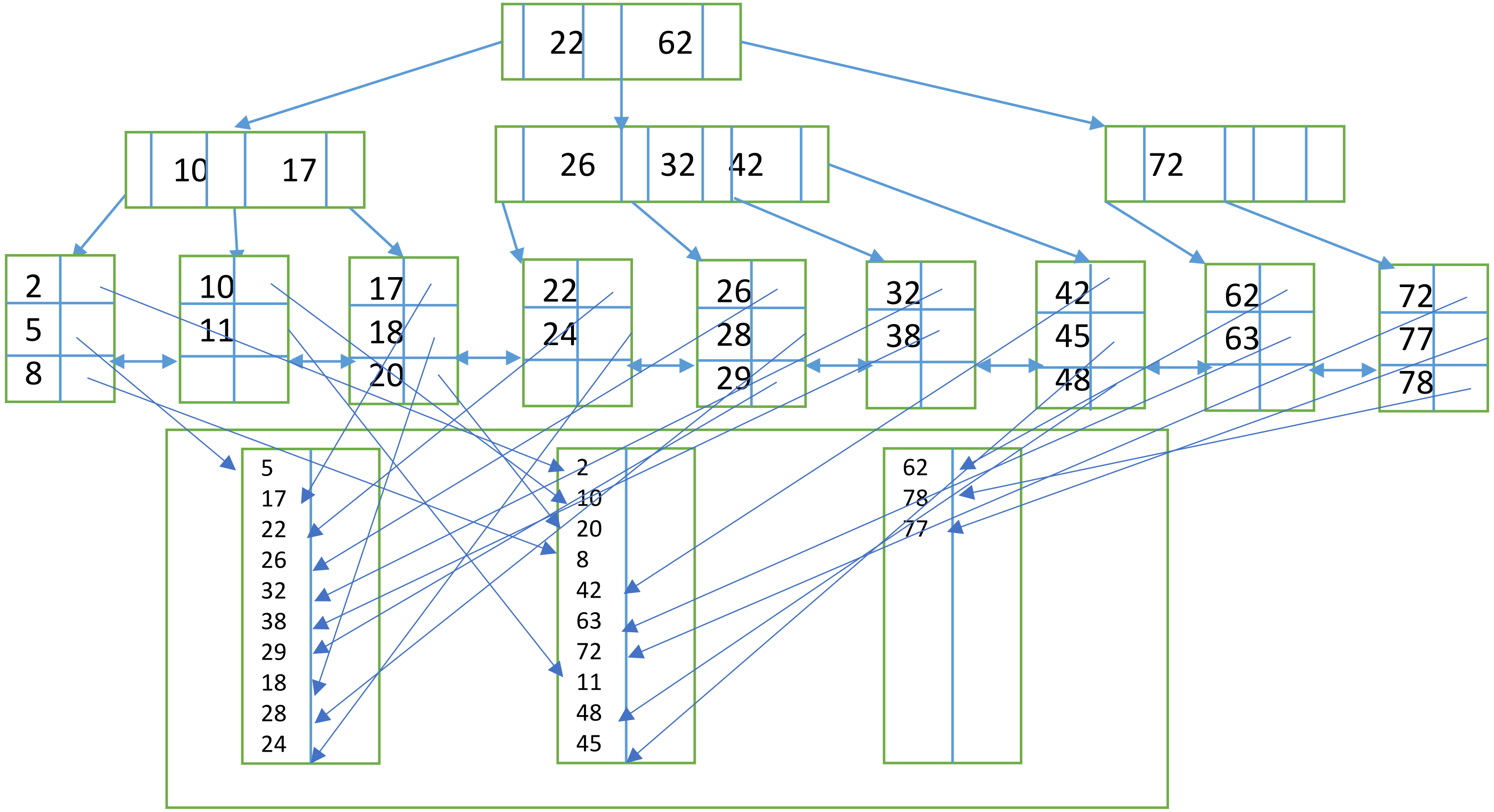
- A combination between B-trees and ISAM
  - The search start from the root and it is redirected through comparisons to a terminal node
  - In a B+ tree all the pointes to the records from the tables are only on the level of the terminal nodes
- A B+ tree can have less levels – with a higher capacity to store the search key – than the correspondingly B tree
- B-tree variant
- Last level contains all the values (key values and the records' addresses)
- Some key values can also appear in non-terminal nodes, without the records' addresses; their purpose is to separate values from terminal nodes (guide the search)
- Terminal nodes are maintained in a doubly linked list (data can be easily scanned)



B+ tree

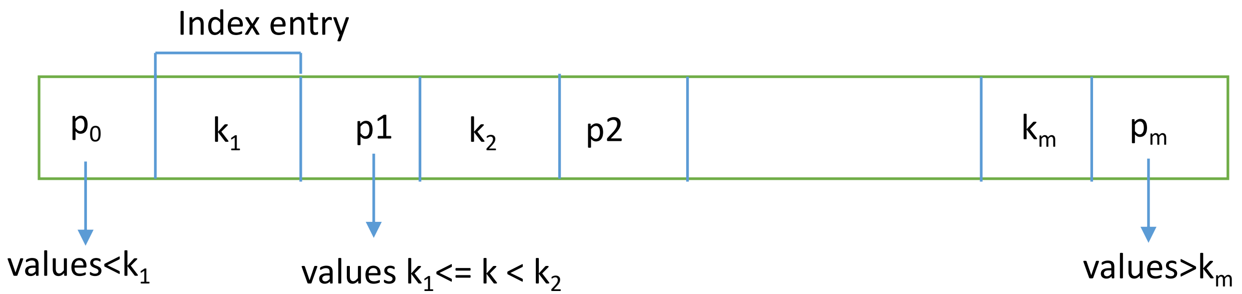
Example:



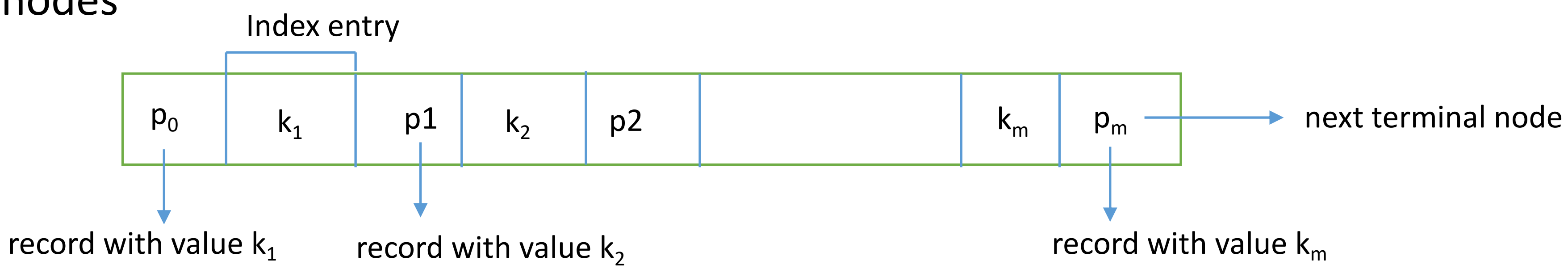


B+ tree

Internal nodes



Terminal nodes



B+ tree in practice:

- concept of *order* - relaxed, replaced by a physical space criterion (for instance, nodes should be at least half-full)
- terminal / non-terminal nodes - different numbers of entries; usually, inner nodes can store more entries than terminal ones
- variable-length search key => variable-length entries => variable number of entries / page
- if Option 3 is used (<k, rid_list>) => variable-length entries (in the presence of duplicates), even if attributes are of fixed length

B+ tree

B+ tree in practice:

- Typic order: 200. Cover factor: 67%. Medium: number_of_values / number_of_indexed_pages = 133
- Typical capacities
  - Hight 4: 133^4=312900700 records
  - Hight 3: 133^3=2352637 records
- In general can be memorized in the buffer from the internal memory
  - Level 1: 1 page = 8Kbytes
  - Level 2: 133 pages = 1 Mbyte
  - Level 3: 17689 pages = 133 Mbytes

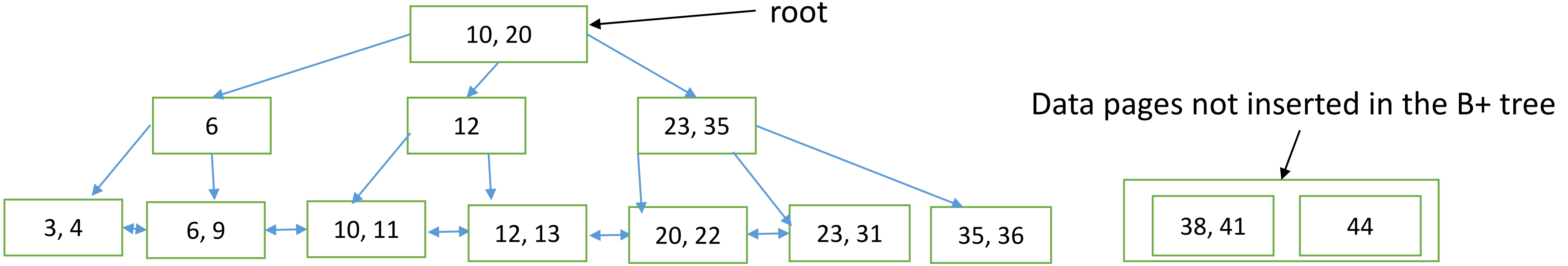
Conclusions B+ trees:

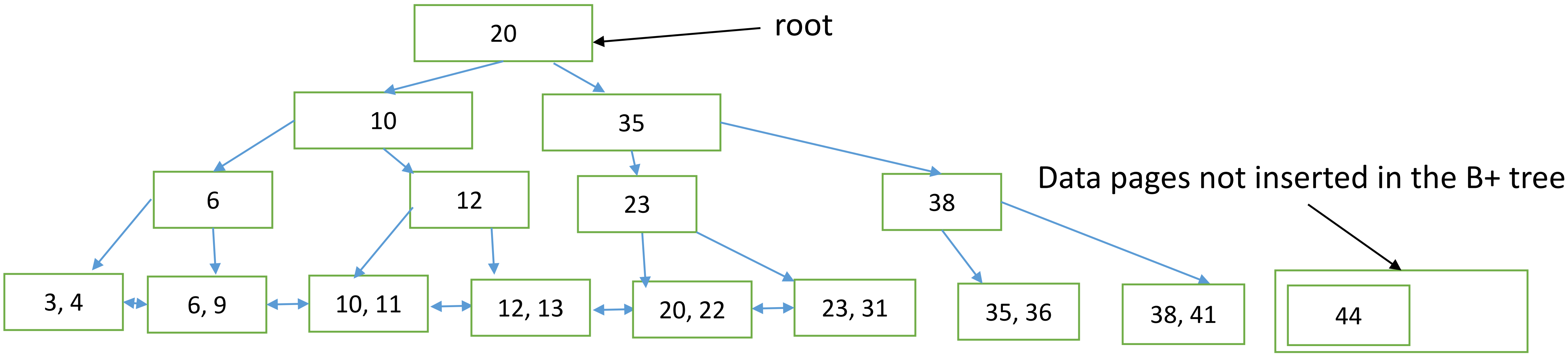
- The indexes remain balanced as long as the time is uniform
- Seldom are more than 3-5 levels (the first levels are kept in RAM such that a search will need only 2-3 I/O
- In general the nodes occupy 67% (so, it is used with 50% more space than it is necessarily)
- One of the most used model for the index structure in DBMS and also an optimized one
- B+ tree can be used for the clustered indexes, rare indexes (if the table is sort) but also for the unclustered indexes, dense indexes

B+ tree

Bulk Loading

- The entries from the index are always inserted in the rightest page up to the last level. When the page becomes full, it is split into 2 page.
- More fast than the repeated inserts





B+ tree

Bulk Loading

- Option 1: multiple inserts: slow, the terminal node are sequential stored
- Option 2: Bulk Loading
  - The index can be used concurrently
  - Less I/O during the constructions
  - The terminal nodes are stored sequentially (and linked)
  - Can be controlled the degree filling of a page

B+ tree prefix (key compress)

- Increase of the number of the values stored in a node
- The index values are used only to direct the traffic of the comparations, so, can be compressed
  - Example: if there are adjunct entries into the index the following values for the search key *Dan Moore, David John, Katty Hulk* can be abbreviated *David Jon* with *Dav*, and correspondingly the others
- Insert / delete can be modified correspondingly

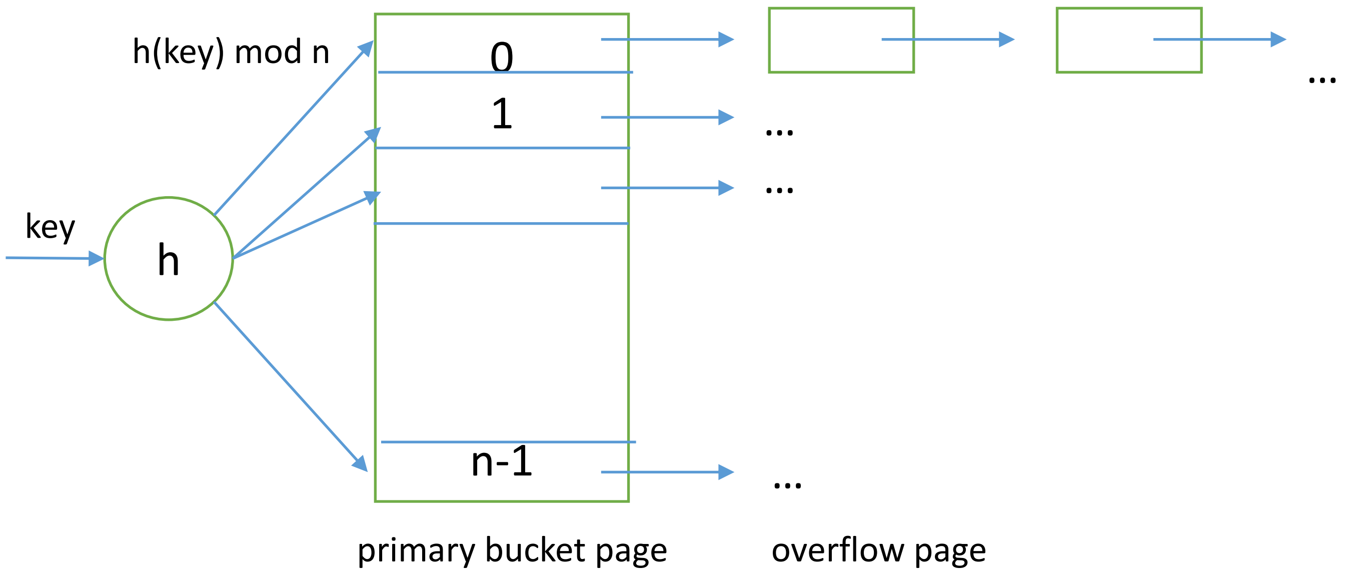
Hash-Base Indexing

Hashing function
- maps search key values into a range of bucket numbers

Hashed file
- search key (fields of the file)
- records grouped into *buckets*
- determine record r's bucket - apply hash function to search key
- quick location of records with given search key value (e.g. a file hashed on StudentName. Display the student *Popescu*.)

Ideal for equality selections

Static hashing
- Bucket 0 to n-1
- Bucket:
    - one primary page
    - possibly extra overflow pages
- Data entries in buckets: a1/a2/a3



Hash-Base Indexing

Static hashing

- search for a data entry
  - apply hashing function to identify the bucket
  - search the bucket
  - possible optimization: entries sorted by search key
- add a data entry
  - apply hashing function to identify the bucket
  - add the entry to the bucket
  - if there is no space in the bucket:
    - allocate an overflow page
    - add the data entry to the page
    - add the overflow page to the bucket's overflow chain
- delete a data entry
  - apply hashing function to identify the bucket
  - search the bucket to locate the data entry
  - remove the entry from the bucket
  - if the data entry is the last one on its overflow page:
    - remove the overflow page from its overflow chain
    - add the page to a free pages list

Static hashing

- good hashing function
  - few empty buckets
  - few records in the same bucket
  - i.e. key values are uniformly distributed over the set of buckets

References:

- C.J. Date, *An Introduction to Database Systems (8th Edition)*, Addison-Wesley, 2003.
- H. Garcia-Molina, J. Ullman, J. Widom, *Database Systems: The Complete Book, Prentice Hall Press*, 2008.
- G. Hansen, J. Hansen, *Database Management And Design (2nd Edition),* Prentice Hall, 1996.
- R. Ramakrishnan, J. Gehrke, *Database Management Systems*, McGraw- Hill, 2007. http://pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed.html
- R. Ramakrishnan, J. Gehrke, *Database Management Systems (2nd Edition),* McGraw-Hill, 2000.
- A. Silberschatz, H. Korth, S. Sudarshan, *Database System Concepts*, McGraw-Hill, 2010. http://codex.cs.yale.edu/avi/db-book/
- L. Țâmbulea, *Curs Baze de date*, Facultatea de Matematică și Informatică, UBB, 2013-2014.
- J. Ullman, J. Widom, *A First Course in Database Systems*, http://infolab.stanford.edu/~ullman/fcdb.html