Databases

Lectures 9-10*

Indexes. Tree-Structured Indexing. Hash-Based Indexing

*01.12.2021 - National Day

Sabina S. CS

Indexes

- motivating example
  - file of students records sorted by name
    - good file organization
      - retrieve students in alphabetical order
    - not a good file organization
      - retrieve students whose age is in a given range
      - retrieve students who live in Timișoara
- index
  - auxiliary data structure that speeds up operations which can't be efficiently carried out given the file's organization
  - enables the retrieval of the rids of records that meet a selection condition (e.g., the rids of records describing students who live in Timișoara)

Indexes

- *search key*
  - set of one or more attributes of the indexed file (different from the *key* that identifies records)
- an index speeds up queries with equality / range selection conditions on the search key
- *entries*
  - records in the index (e.g., <search key, rid>)
  - enable the retrieval of records with a given search key value

Indexes

- example
  - files with students records
  - index built on attribute *city*
  - entries: <city, rid>, where rid identifies a student record
  - such an index would speed up queries about students living in a given city:
    - find entries in the index with city *= 'Timișoara'*
    - follow rids from obtained entries to retrieve records describing students who live in Timișoara

Indexes

- an index can improve the efficiency of certain types of queries, not of all queries (analogy - when searching for a book at the library, index cards sorted on author name cannot be used to efficiently locate a book given its title)

- organization techniques (access methods) - examples
  - B+ trees
  - hash-based structures

- changing the data in the file => update the indexes associated with the file (e.g., inserting records, updating search key columns, updating columns that are not part of the key, but are included in the index)
- index size
  - as small as possible, as indexes are brought into main memory for searches

Indexes - Data Entries

- problems
  - what does a data entry contain?
  - how are the entries of an index organized?

- let k* be a data entry in an index; the data entry:
  - alternative 1

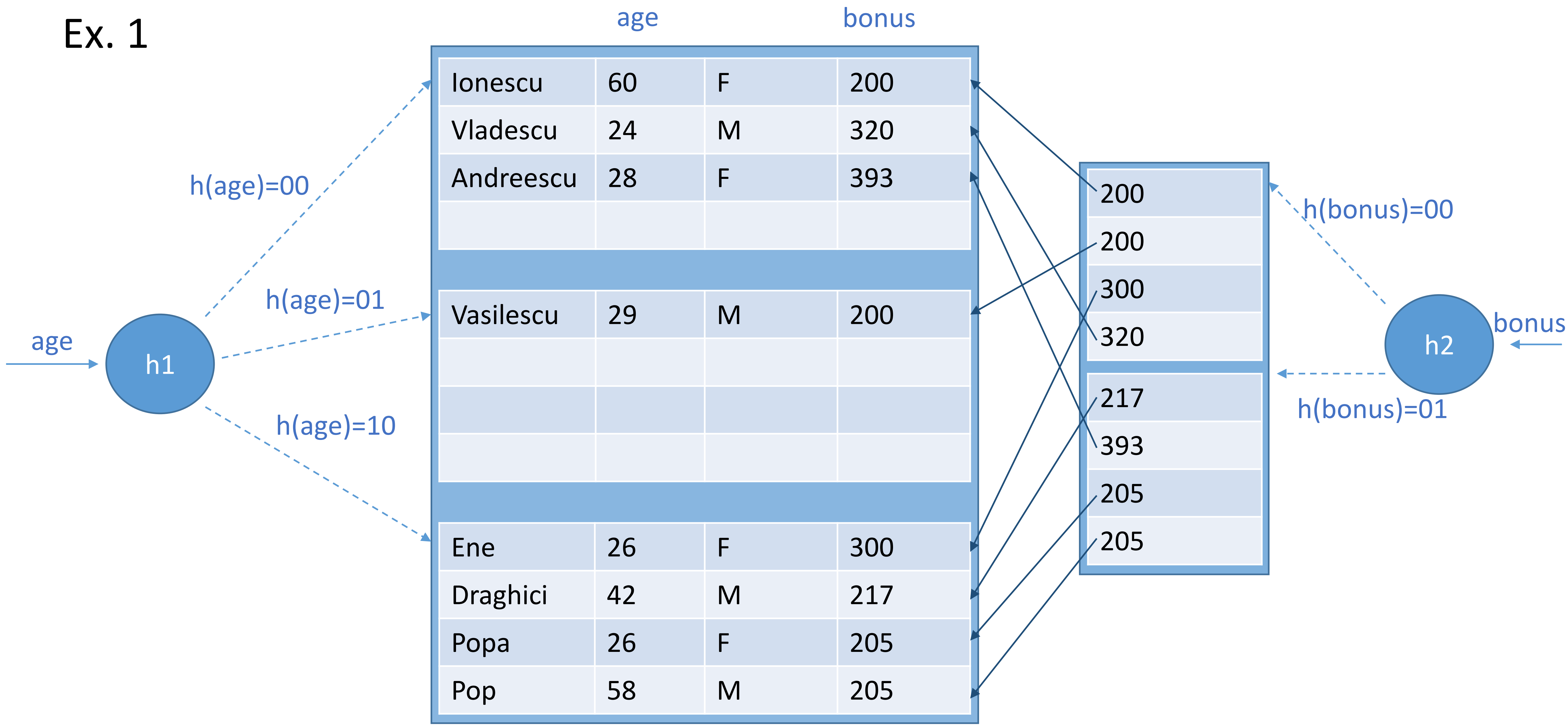
    is an actual data record with search key value = k
  - alternative 2
    - is a pair <k, rid> (rid – id of a data record with search key value = k)
  - alternative 3
    - is a pair <k, rid_list> (rid_list – list of ids of data records with search key value = k)

Indexes - Data Entries

- a1
  - the file of data records needn't be stored in addition to the index
  - the index is seen as a special file organization
  - at most 1 index / collection of records should use alternative a1 (to avoid redundancy)
- a2, a3
  - data entries point to corresponding data records
  - in general, the size of an entry is much smaller than the size of a data record
  - a3 is more compact than a2, but can contain variable-length records
  - can be used by several indexes on a collection of records
  - independent of the file organization

Indexes - Data Entries

Ex. 1



	age		bonus
Ionescu	60	F	200
Vladescu	24	M	320
Andreescu	28	F	393
Vasilescu	29	M	200
Ene	26	F	300
Draghici	42	M	217
Popa	26	F	205
Pop	58	M	205

200
200
300
320
217
393
205
205

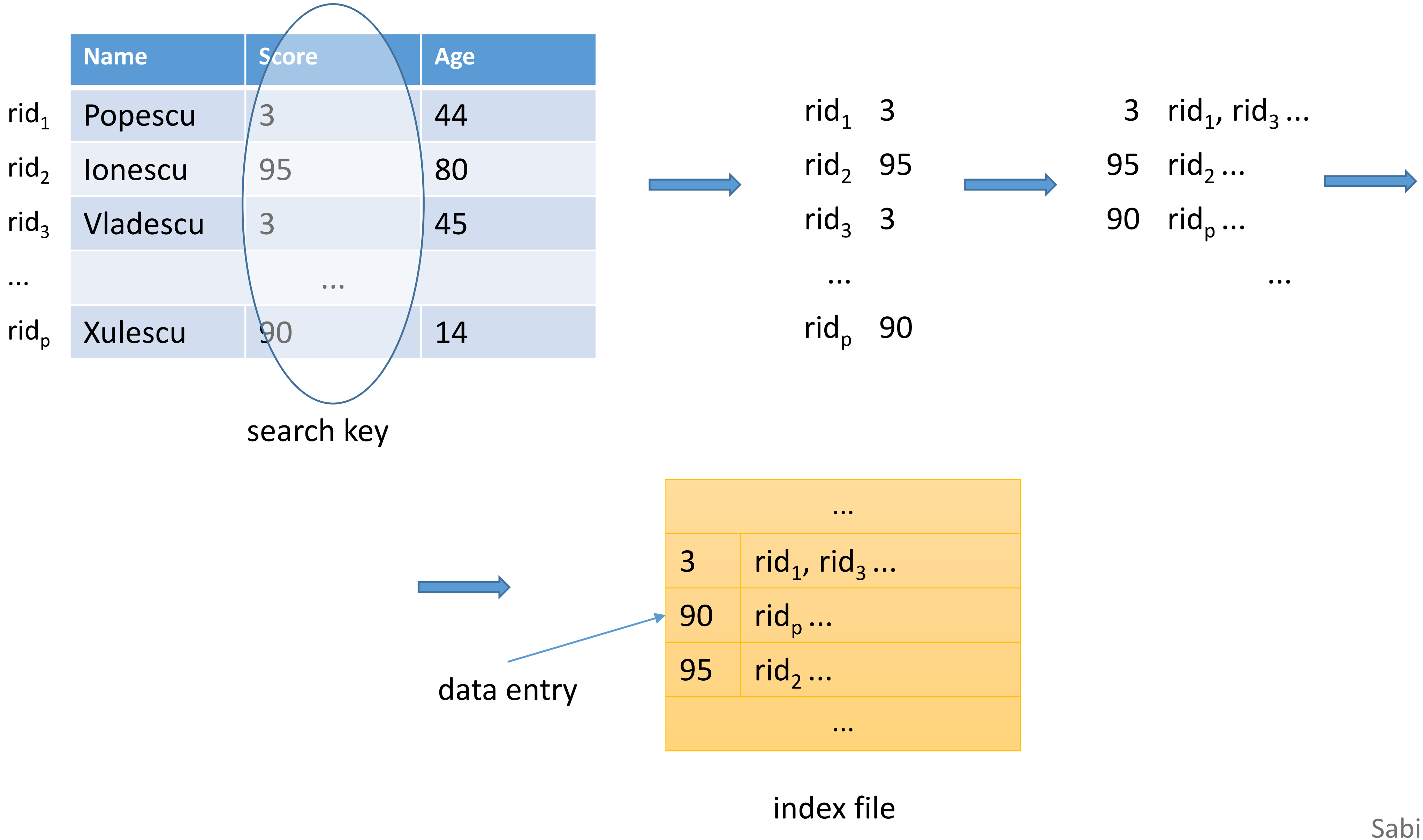
Indexes - Data Entries

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 …
	…

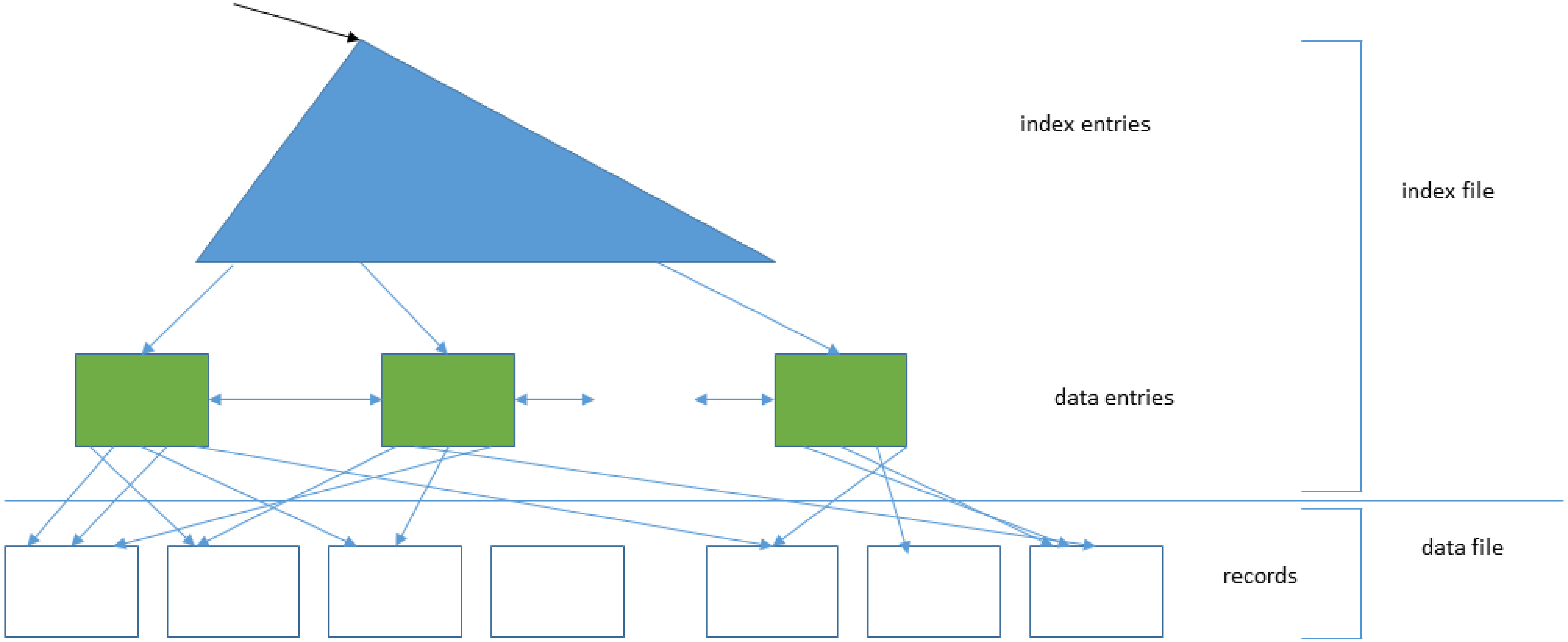
data entry

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

index file

Clustered / Unclustered Indexes

- clustered index: the order of the data records is close to / the same as the order of the data entries
- unclustered index: index that is not clustered



index entries guide the search for data entries

unclustered index, alternative 2

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
    - the number of I/O operations could be equal to the number of data entries that satisfy the query's condition

Primary / Secondary Indexes

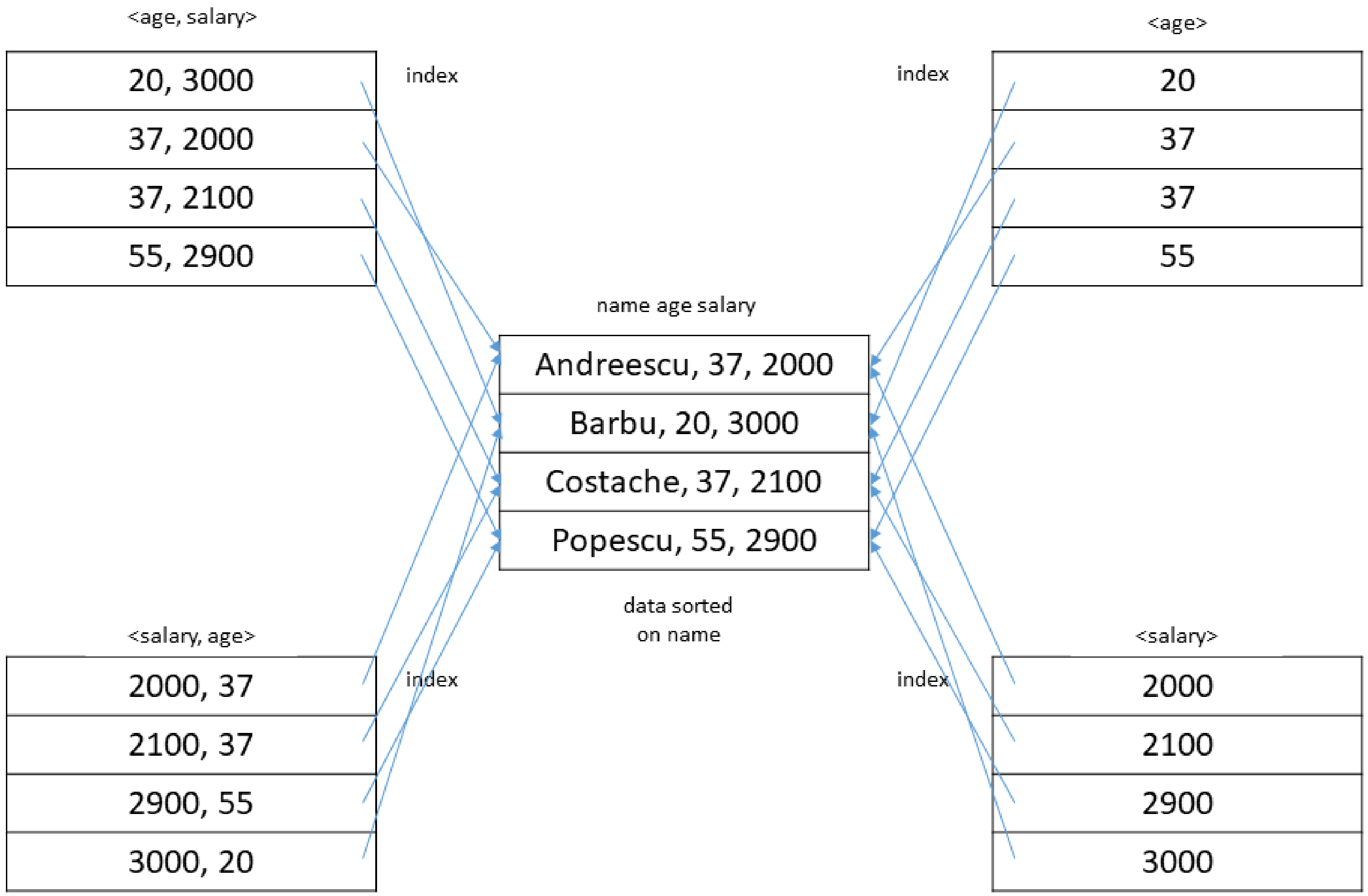
- primary index
  - the search key includes the primary key
- secondary index
  - index that is not primary
- unique index
  - the search key contains a candidate key

- 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



<age, salary>

index
20, 3000
37, 2000
37, 2100
55, 2900

<age>

index
20
37
37
55

name age salary

Andreescu, 37, 2000
Barbu, 20, 3000
Costache, 37, 2100
Popescu, 55, 2900

data sorted
on name

<salary, age>

index
2000, 37
2100, 37
2900, 55
3000, 20

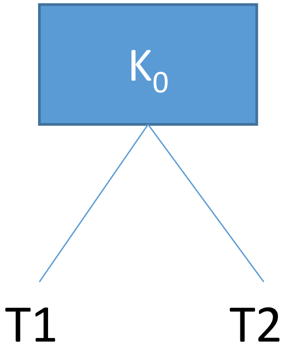
<salary>

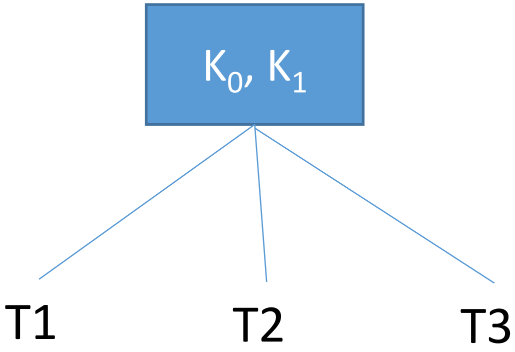
index
2000
2100
2900
3000

2-3 trees

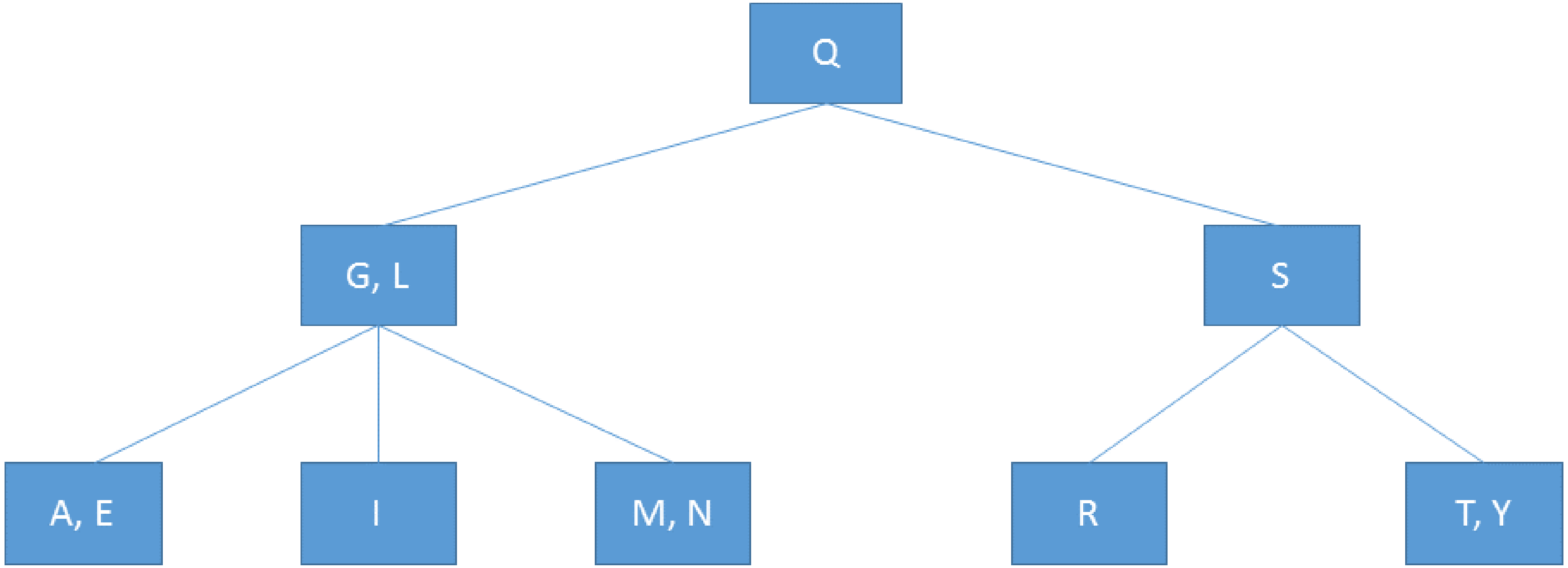
2-3 tree storing key values (collection of distinct values)

- 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)

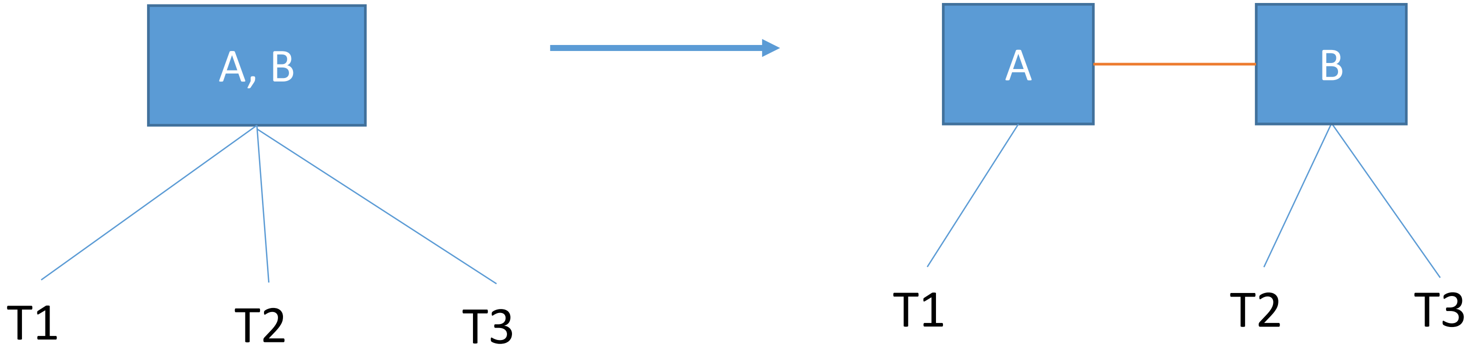


- <u>storing a 2-3 tree</u>
  - 2-3 tree index storing the values of a key
  - tree - key value + address of record (file / DB address of record with corresponding key value)

- 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)

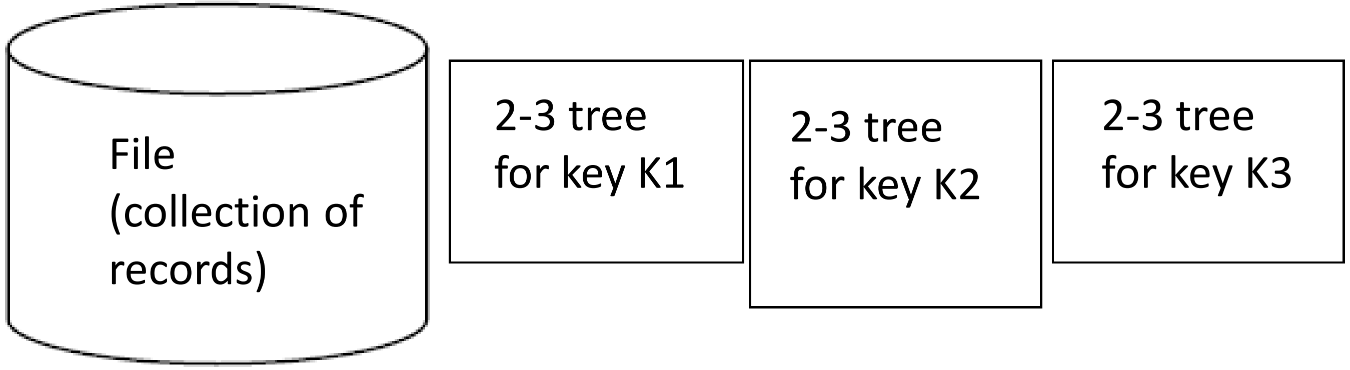
- IND - indicator that specifies the type of the link to the right (the 2 possible values can be seen in the previous figure)

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

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

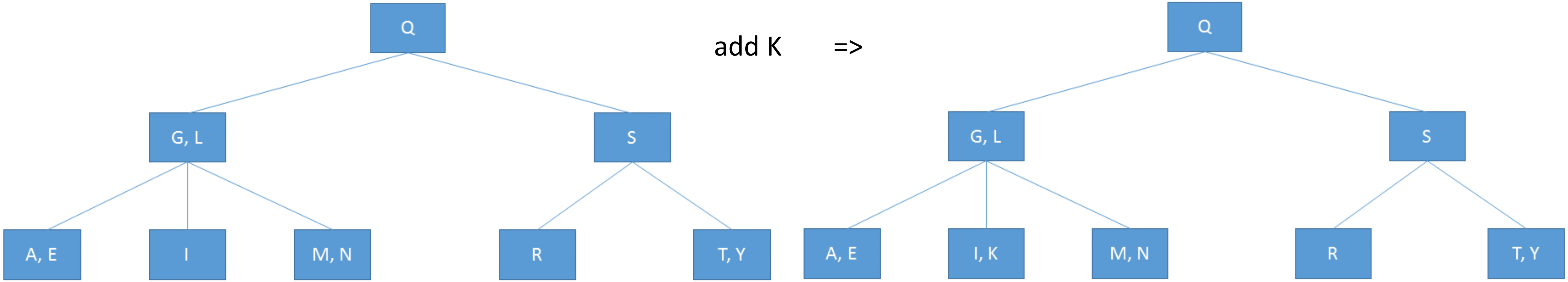
- NV – number of values in the node (1 or 2)
- K_1, K_2 – key values
- $ADDR_1$, $ADDR_2$ – the records' addresses (corresponding to K_1 and K_2)
- $Pointer_1$, $Pointer_2$, $Pointer_3$ – the 3 subtrees' addresses

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

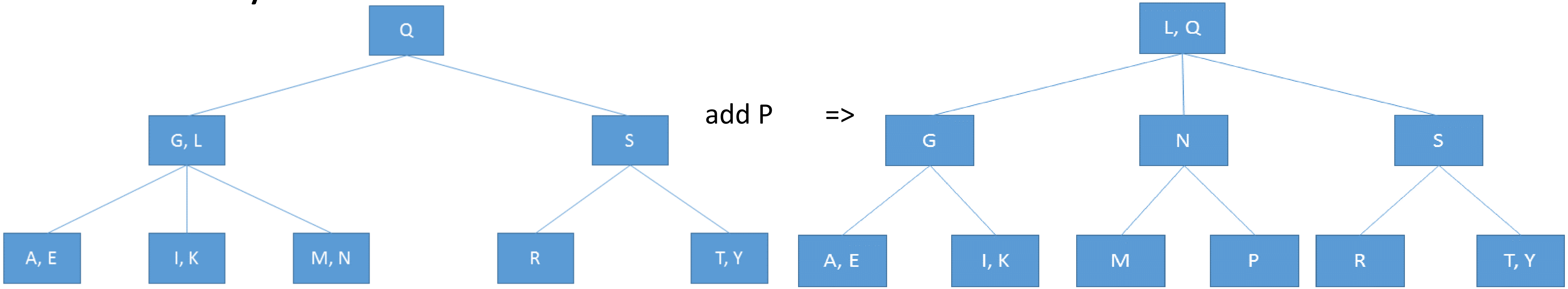


- operations in a 2-3 tree
  - searching for a record with key value K_0
  - inserting a record - description
  - removing a record - description
  - tree traversal (partial, total)

- <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



- 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

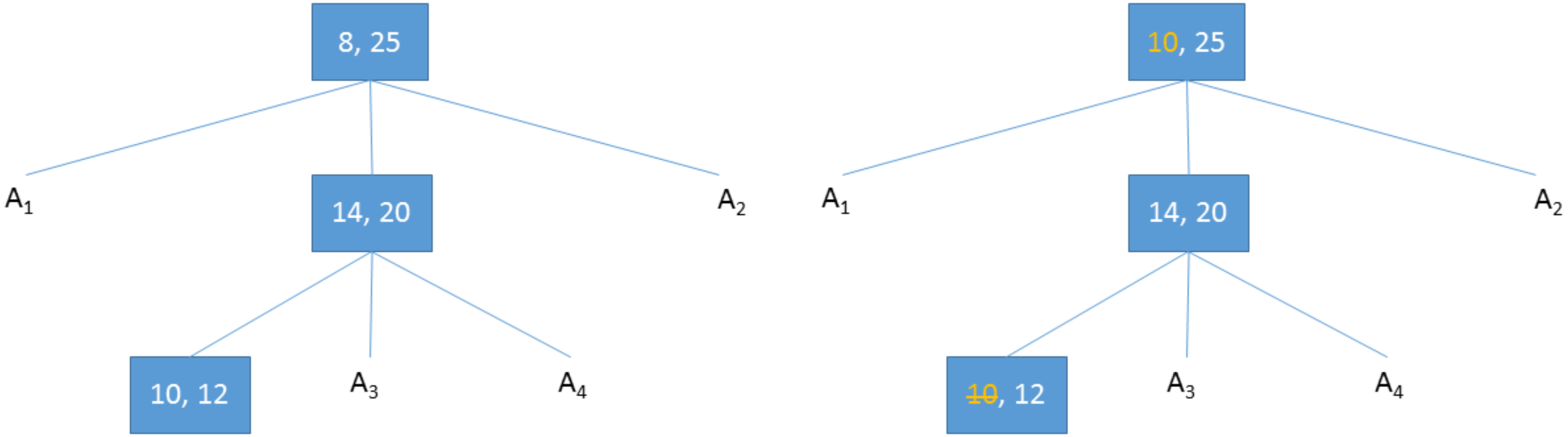


- delete a value K_0

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

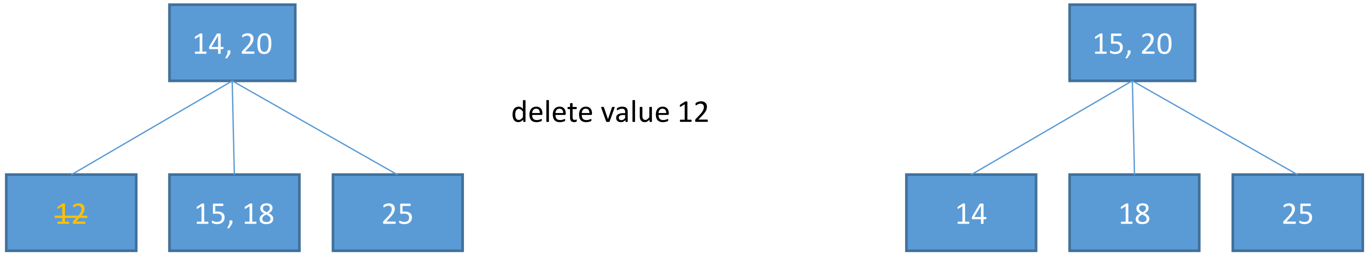
- e.g., remove 8:

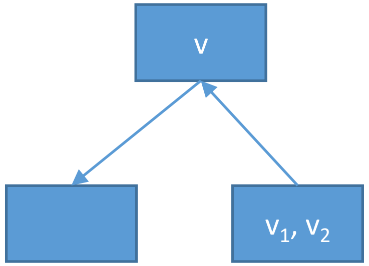


2. perform this step until case a / b occurs

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

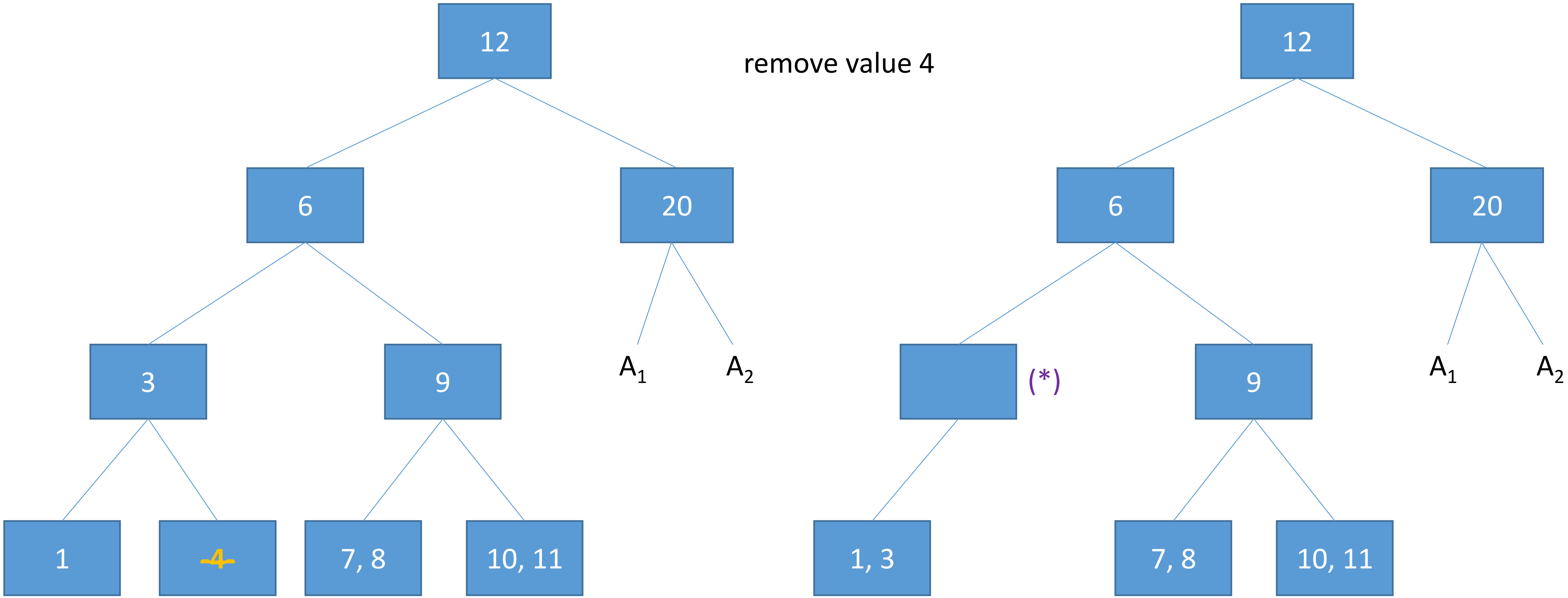




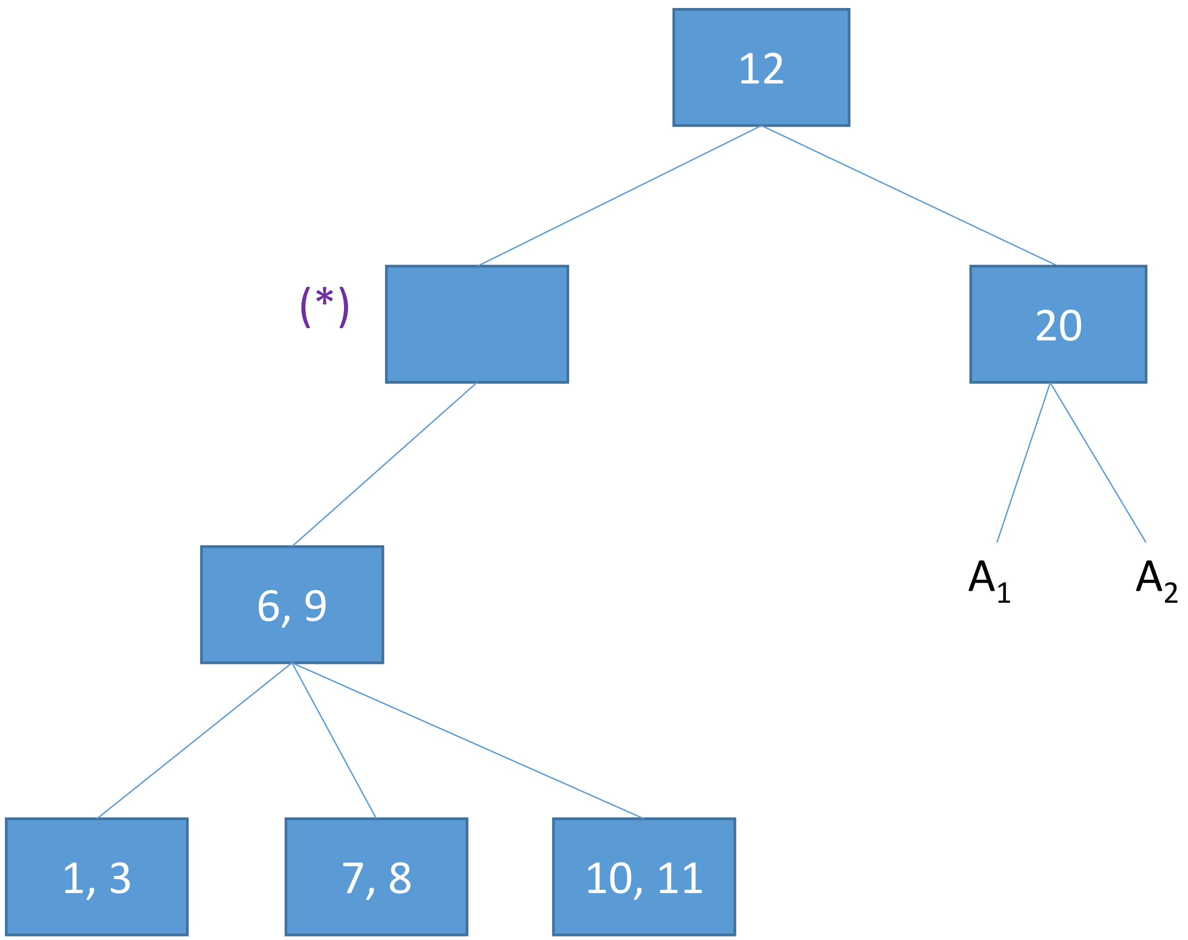
c. if the previous cases do not occur (current node has no values, sibling nodes have 1 value each), then the current node is merged with a sibling and a value from the parent node; case 2 is then analyzed for the parent

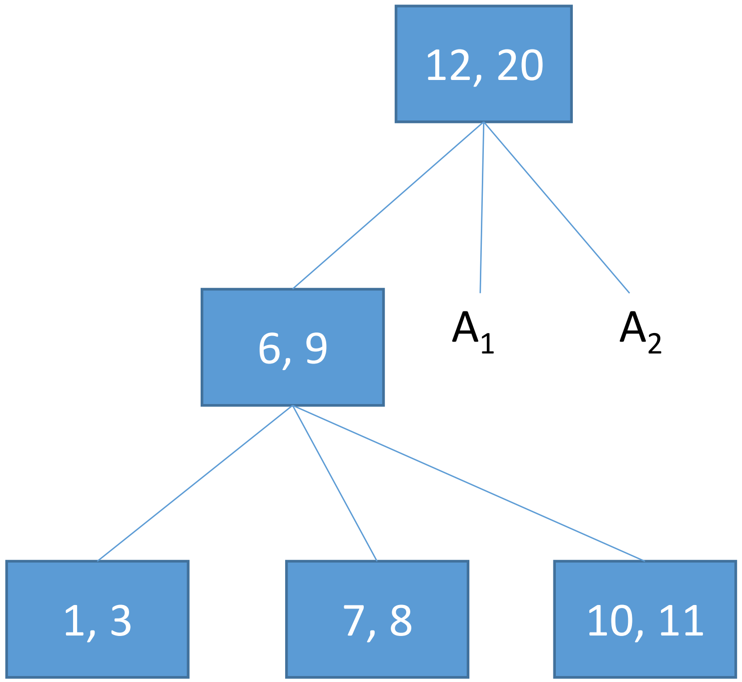
- if the root is reached and it has no values, it is eliminated and the current node becomes the root

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



Sabina S. CS

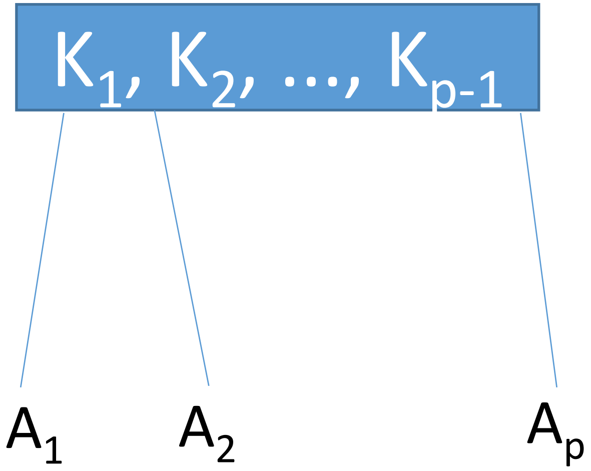

12
(*)
20
6, 9
A1
A2
1, 3
7, 8
10, 11


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

B-trees

- generalization of 2-3 trees
- B-tree of order m

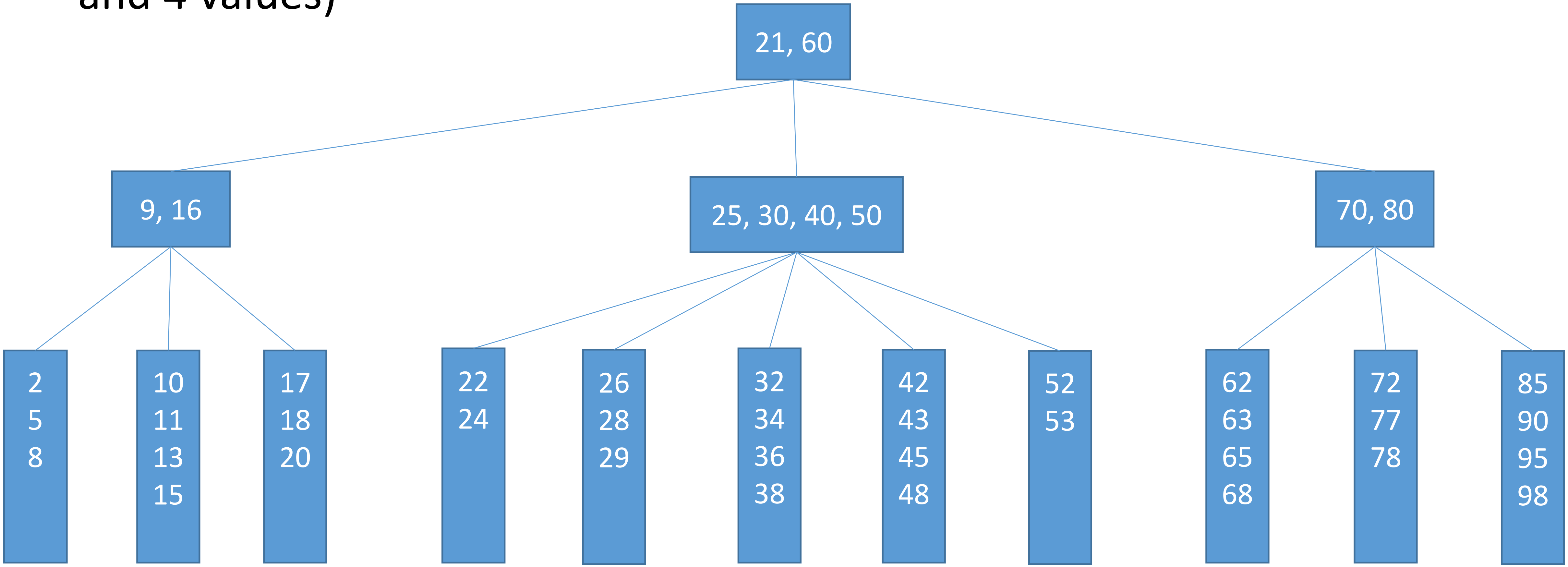
  1. if the root is not a terminal, it has at least 2 subtrees
  2. all terminal nodes – same level
  3. every non-terminal node – at most m subtrees
  4. a node with p subtrees has p-1 ordered values (ascending order): $K_1<K_2<...<K_{p-1}$
     - A_1: values less than K_1
     - A_i: values between K_{i-1} and K_i, i=2,...,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 such 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

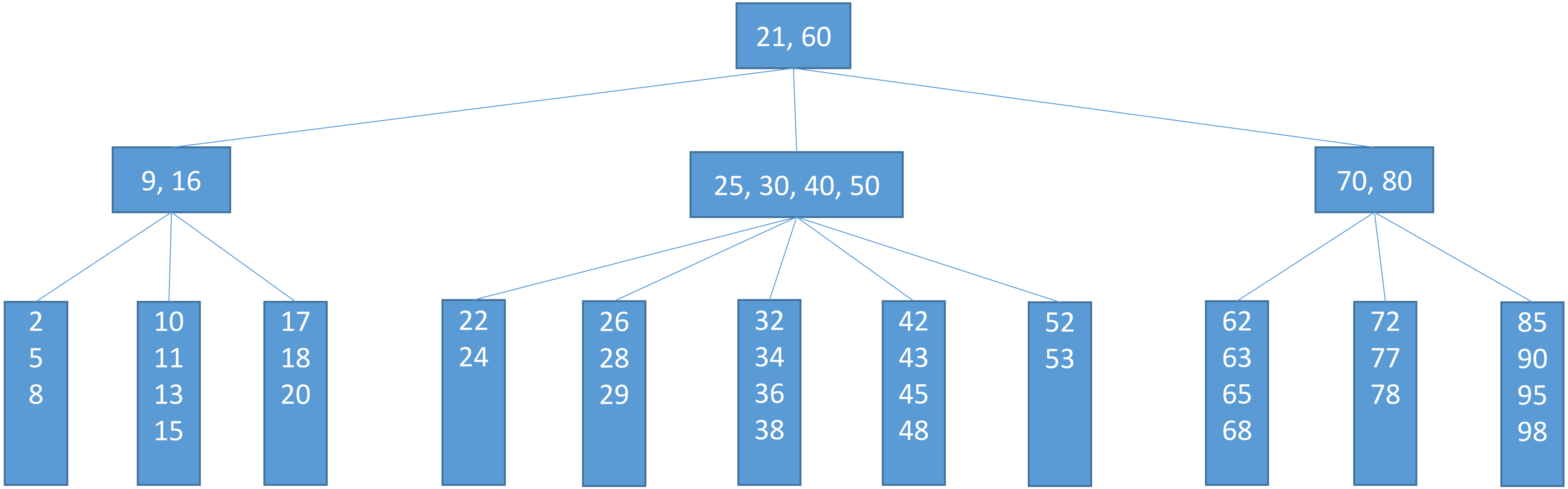
- B-tree of order m
    - useful operations in a B-tree
        - searching for a value
        - adding a value - description
        - removing a value- description
        - tree traversal (partial, total)

- 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)

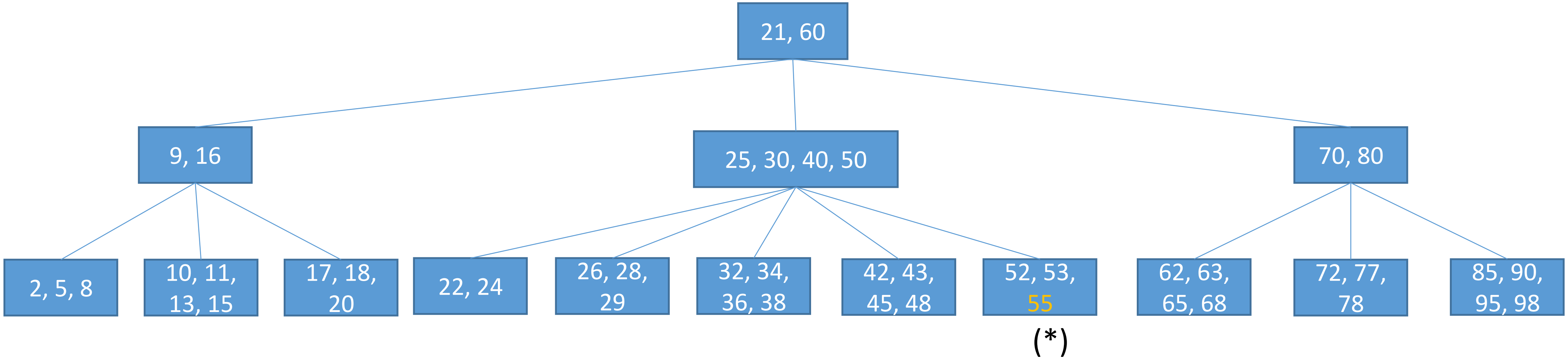
    - if the new value can be added, the search ends in a terminal node

    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:



Sabina S. CS

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

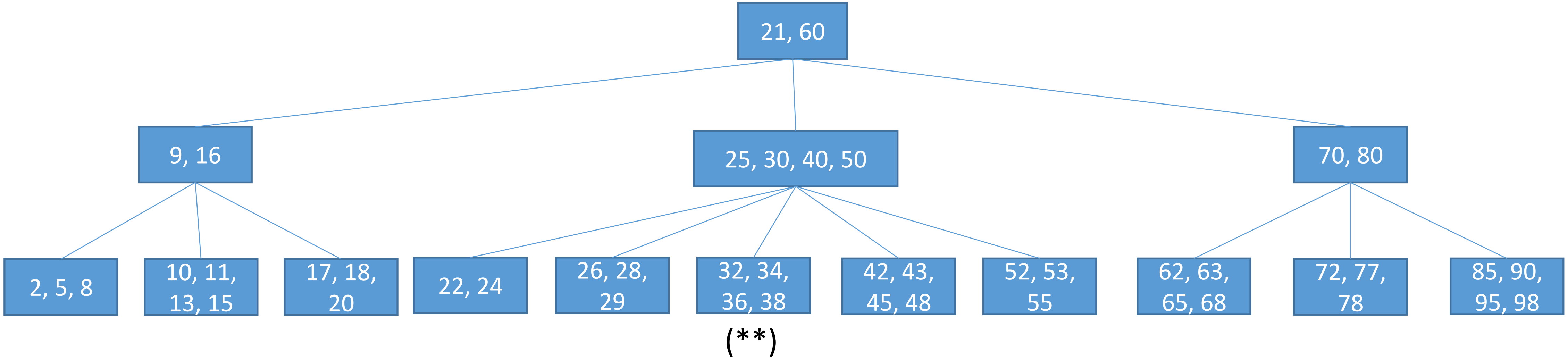


    - 55 belongs to the node marked with (*), which can store at most 4 values

- 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

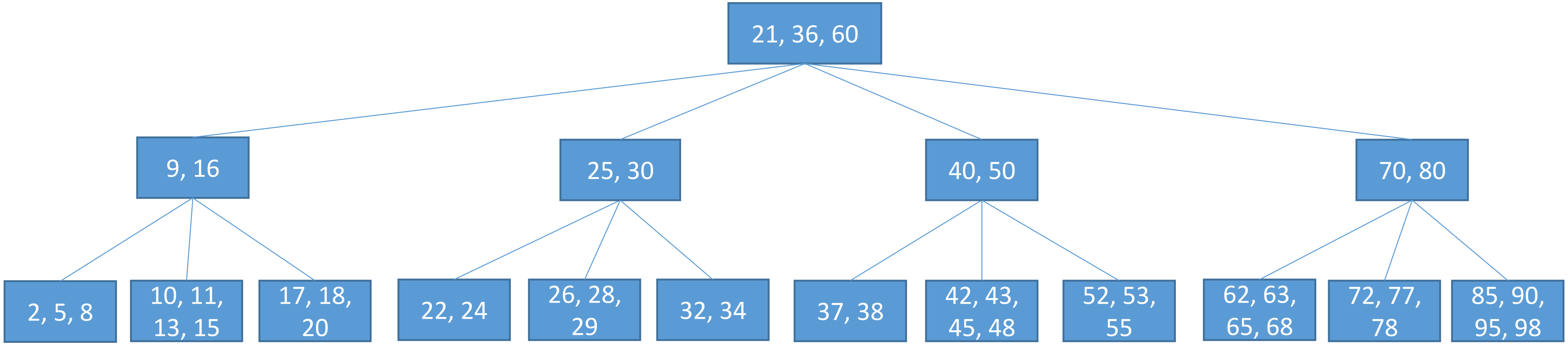
    - e.g., add 37 to the tree below



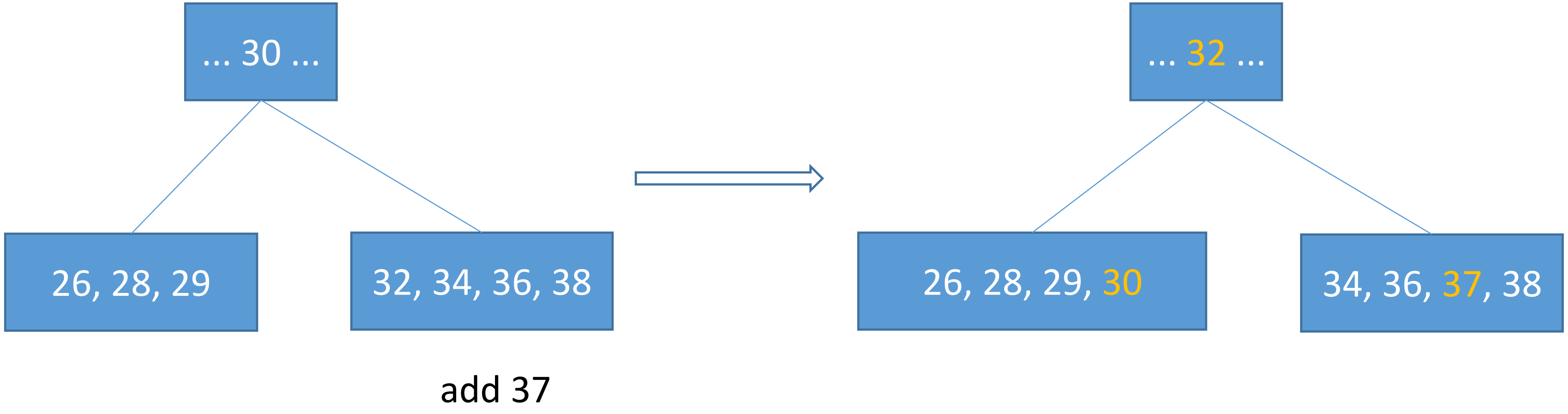
    - the node marked with (**) should contain values *32, 34, 36, 37, 38*

->

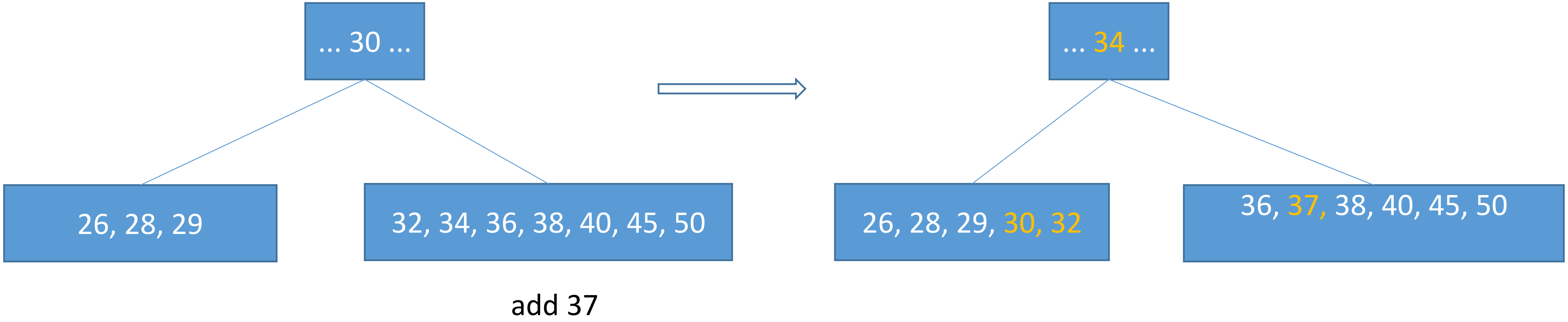
- B-tree of order m
  - adding a new value
    - since the node's capacity is exceeded, it is split into nodes *32, 34*, and *37, 38*, and 36 is attached to the parent node (with values *25, 30, 40, 50*)
    - in turn, the parent must be split into 2 nodes (values *25, 30*, and *40, 50*), and 36 is attached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



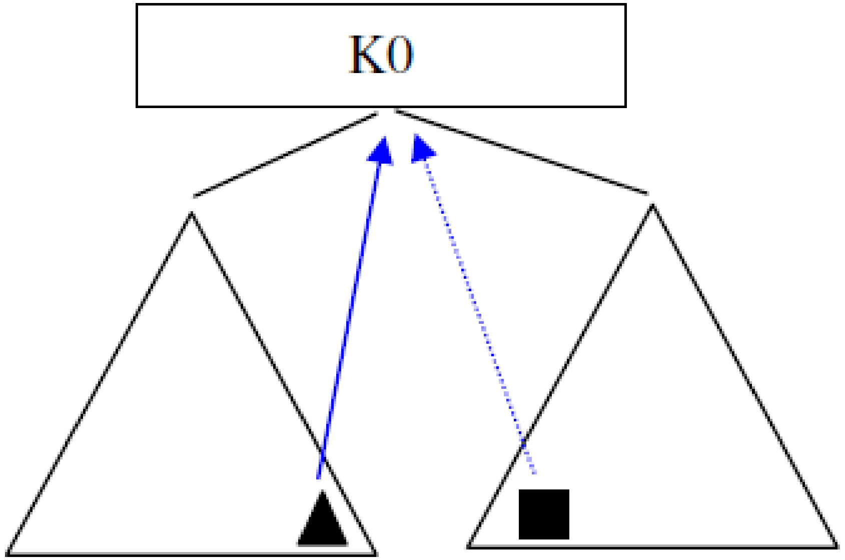
- B-tree of order m
  - adding a new value
    - optimizations
    - e.g., B-tree of order 8 (non-terminal node - between 3 and 7 values, i.e., between 4 and 8 subtrees):



- 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 $\left\lceil \frac{m}{2} \right\rceil - 1 = \left[\frac{m-1}{2} \right]$ 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)



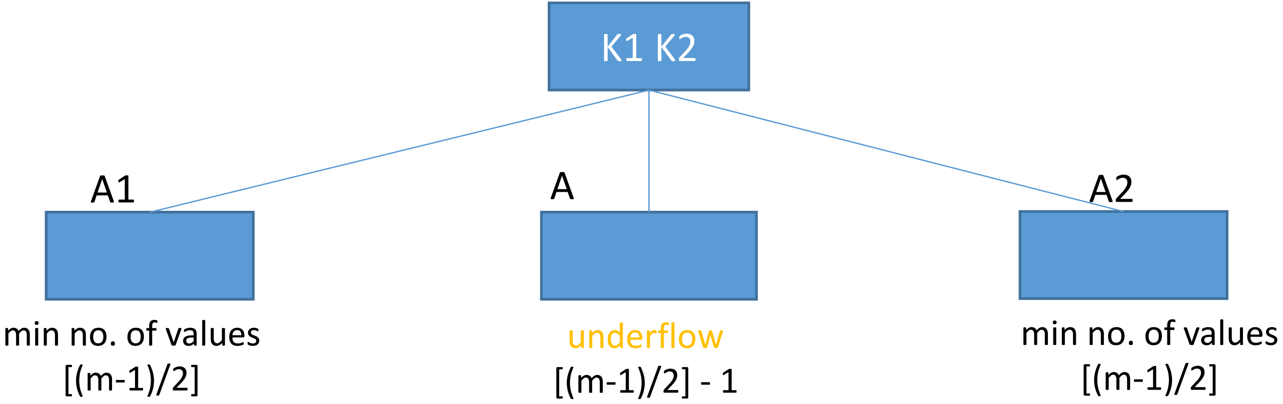
- B-tree of order m
    - removing a value

      3. perform this step until case a / b occurs

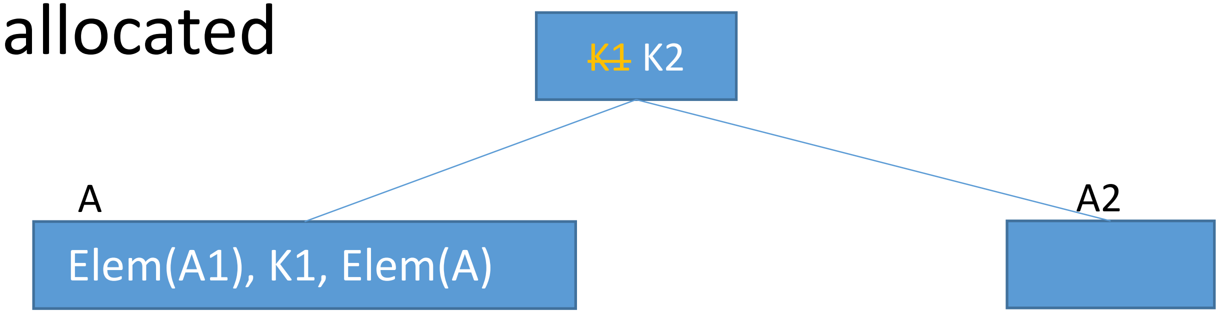
      a. if the current node (from which a value is removed) is the root or underflow doesn't occur, the value is eliminated; the algorithm ends

      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

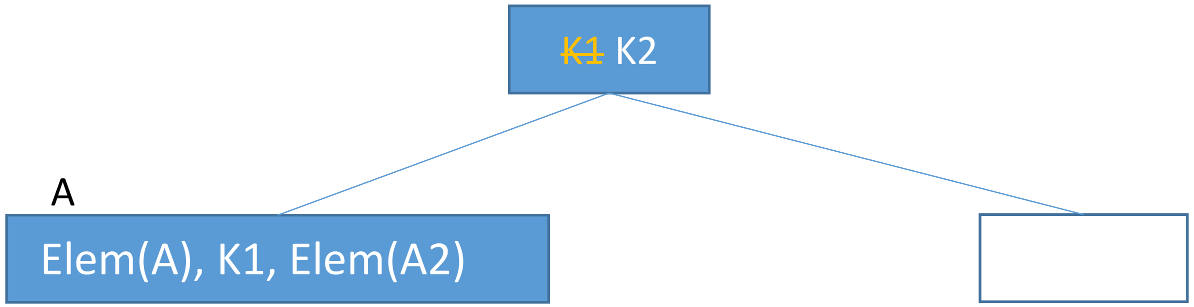
      c. if there is an underflow in A, and sibling nodes A1 and A2 have the minimum number of values, nodes must be concatenated:



- 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: $2+(m-1)*(10+10)+m*10=1024-10 \Rightarrow m=34$
- 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

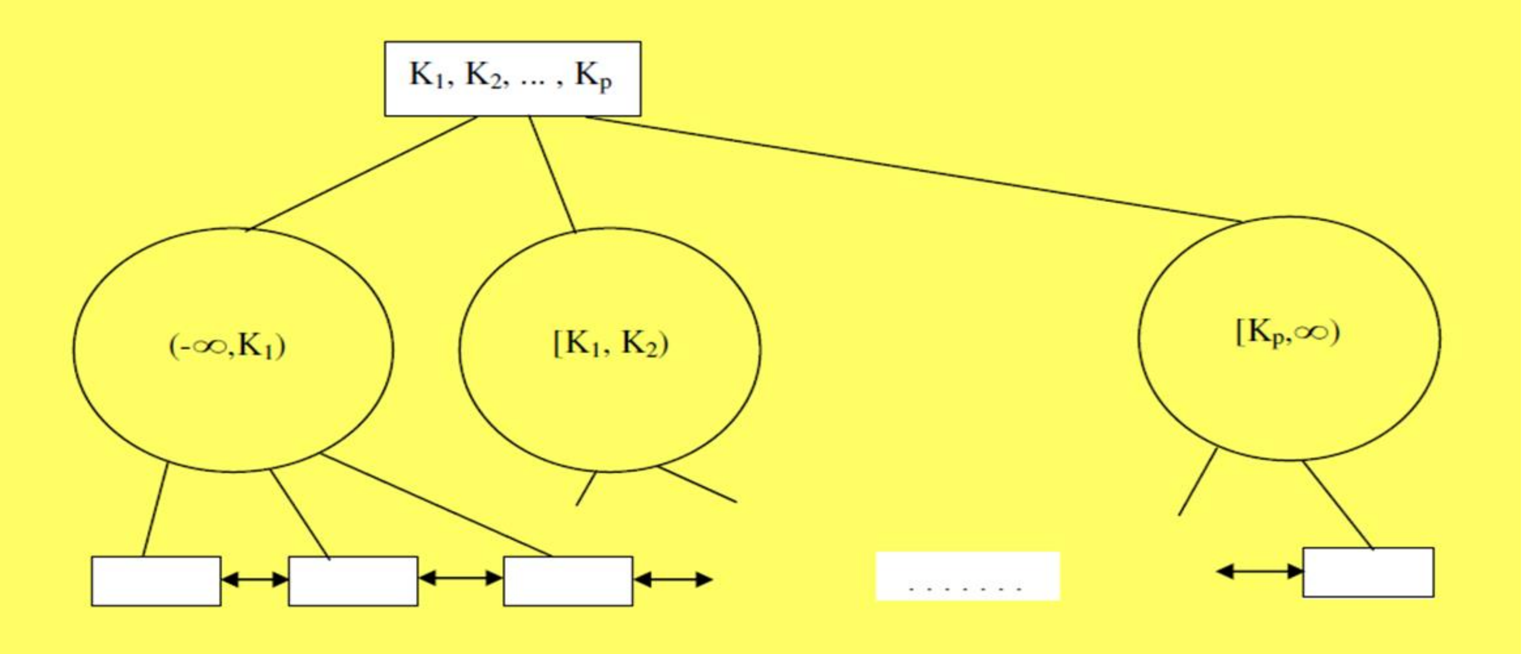
=> at most 4 levels in the tree

- 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+ trees

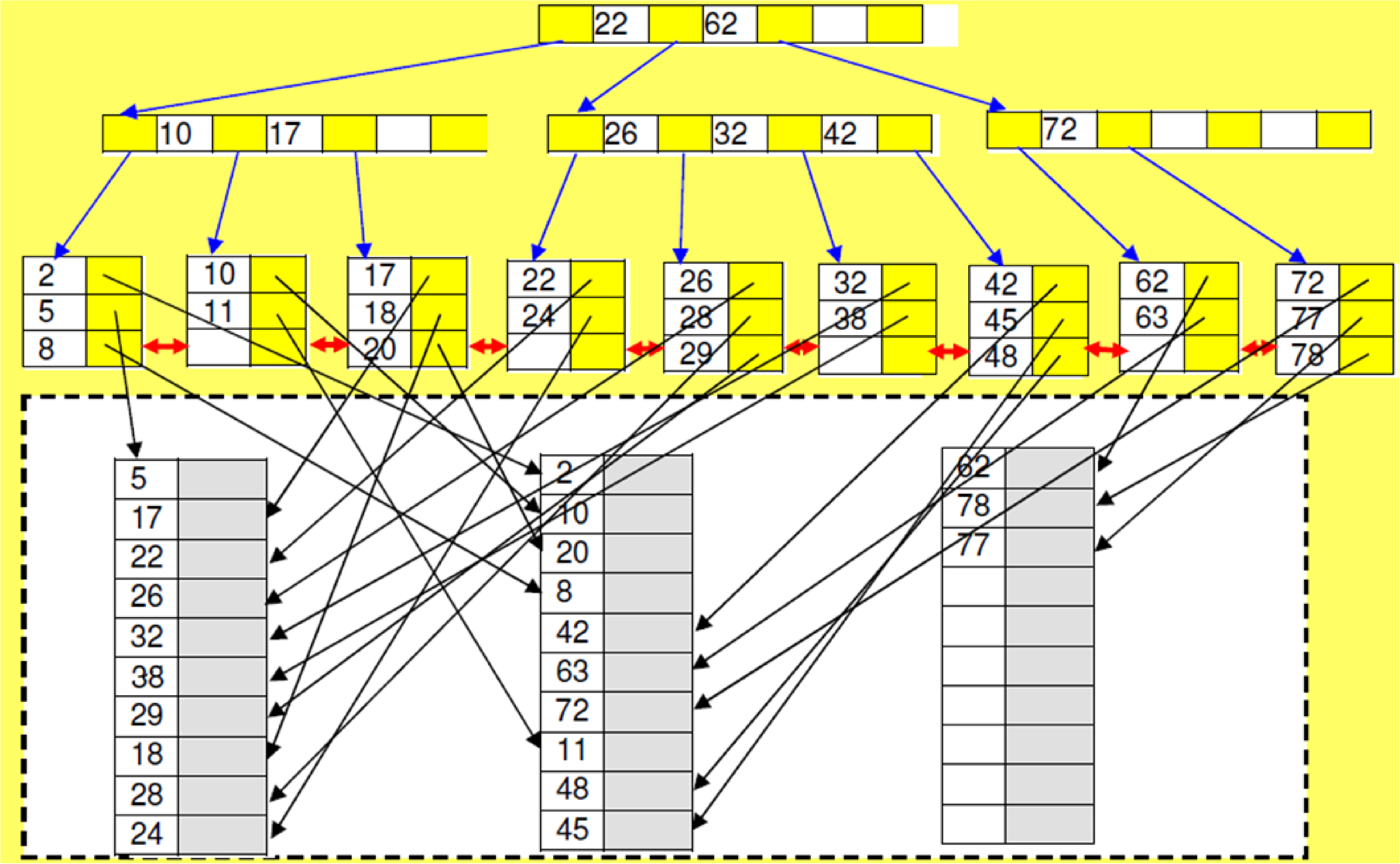
B+ tree

- B-tree variant
- last level contains all 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

- storing a B+ tree
    - B-tree methods
- operations (algorithms)
    - B-tree

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 alternative 3 is used (<k, rid_list>) => variable-length entries (in the presence of duplicates), even if attributes are of fixed length

B+ tree - in practice

* prefix key compression

- larger key size => less index entries fit on a page, i.e., less children / index page => larger B+ tree height
- keys in index entries - just direct the search => often, they can be compressed
- adjacent index entries with search key values: *Meteiut*, *Mircqkjt*, *Morqwkj*
- compress key values: *Me, Mi,* etc
- what if the subtree also contains *Micfgjh*? => need to store *Mir* (instead of *Mi*)
- it's not enough to analyze neighbor index entries *Meteiut* and *Morqwkj*; the largest key value in *Mircqkjt*'s left subtree and the smallest key value in its right subtree must also be examined
- inserts / deletes - modified correspondingly

B+ tree - in practice

- values found in practice
  - order – 200
  - fill factor (node) – 67%
  - fan-out – 133
  - capacity
    - height 4: 133^4 = 312,900,721
    - height 3: 133^3 = 2,352,637
- top levels can often be kept in the BP
  - 1st level – 1 page (8KB)
  - 2nd level – 133 pages (approx. 1MB)
  - 3rd level – 133^2 =17689 pages (approx. 133 MB)

B+ tree - benefits

- balanced index => uniform search time
- rarely more than 3-5 levels, the top levels can be kept in main memory => only a few I/O operations are needed to search for a record
- widely used in DBMSs
- ideal for range selections, good for equality selections as well

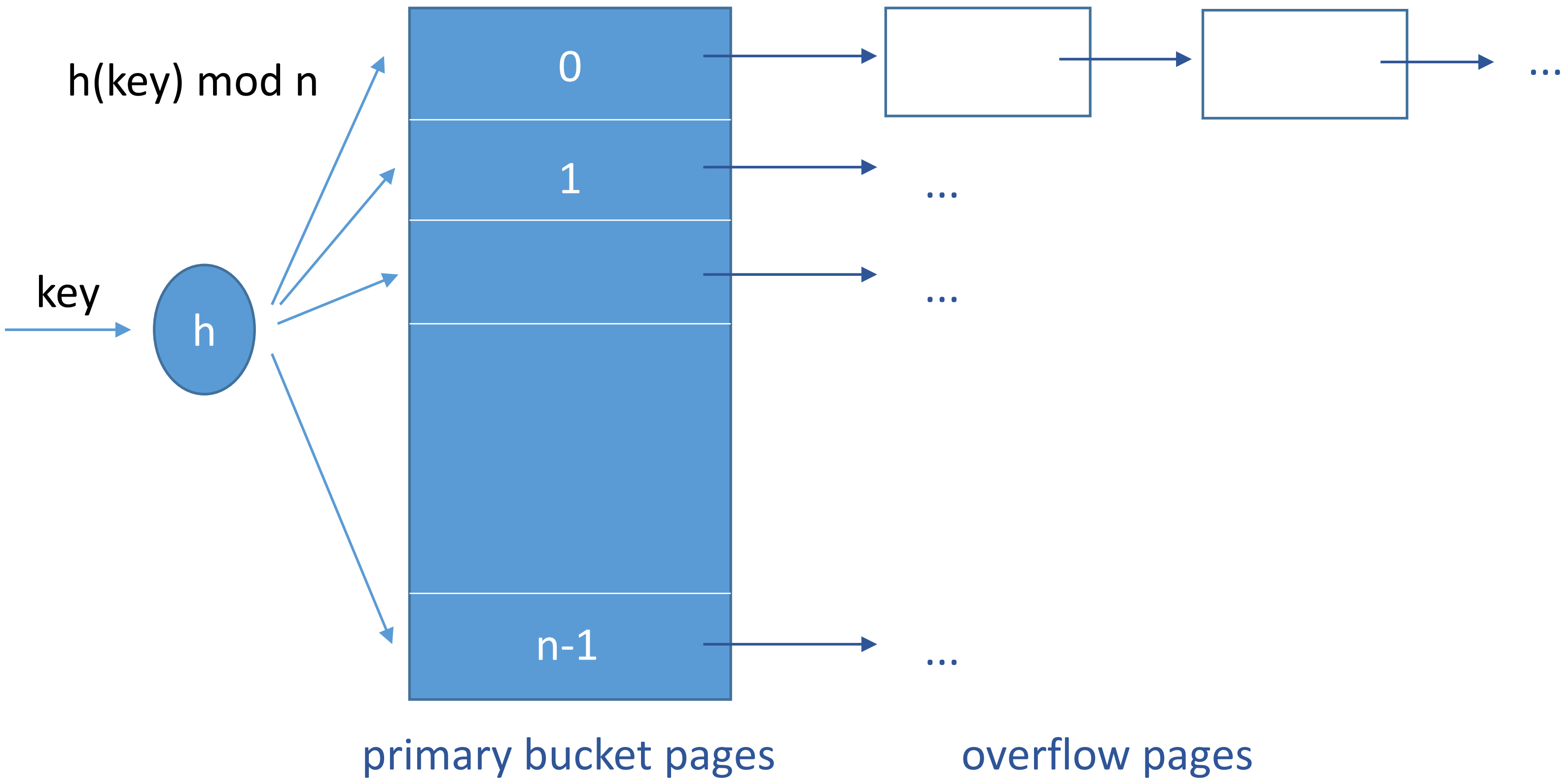
Hash-Based Indexing

- hashing function
    - maps search key values into a range of bucket numbers
- hashed file
    - search key (field(s) 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
        - example: file hashed on *EmployeeName*
            - Find employee *Popescu.*
- ideal for equality selections

Static Hashing

* static hashing

- buckets 0 to n-1
- bucket
  - one primary page
  - possibly extra overflow pages
- data entries in buckets
  - a1/a2/a3



* static hashing

- search for a data entry
    - apply hashing function to identify the bucket
    - search the bucket
    - possible optimization
        - entries sorted by search key

* static hashing

- 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

* static hashing

- 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
    - good function in practice
        - *h(val) = a*val + b*
        - *h(val) mod n* to identify bucket, for buckets numbered 0..n-1

* static hashing

- number of buckets known when the file is created
- ideally
  - search: 1 I/O
  - insert / delete: 2 I/Os
- file grows a lot => overflow chains; long chains can significantly affect performance
  - tackle overflow chains
    - initially, pages - 80% full
    - create a new file with more buckets
- file shrinks => wasted space

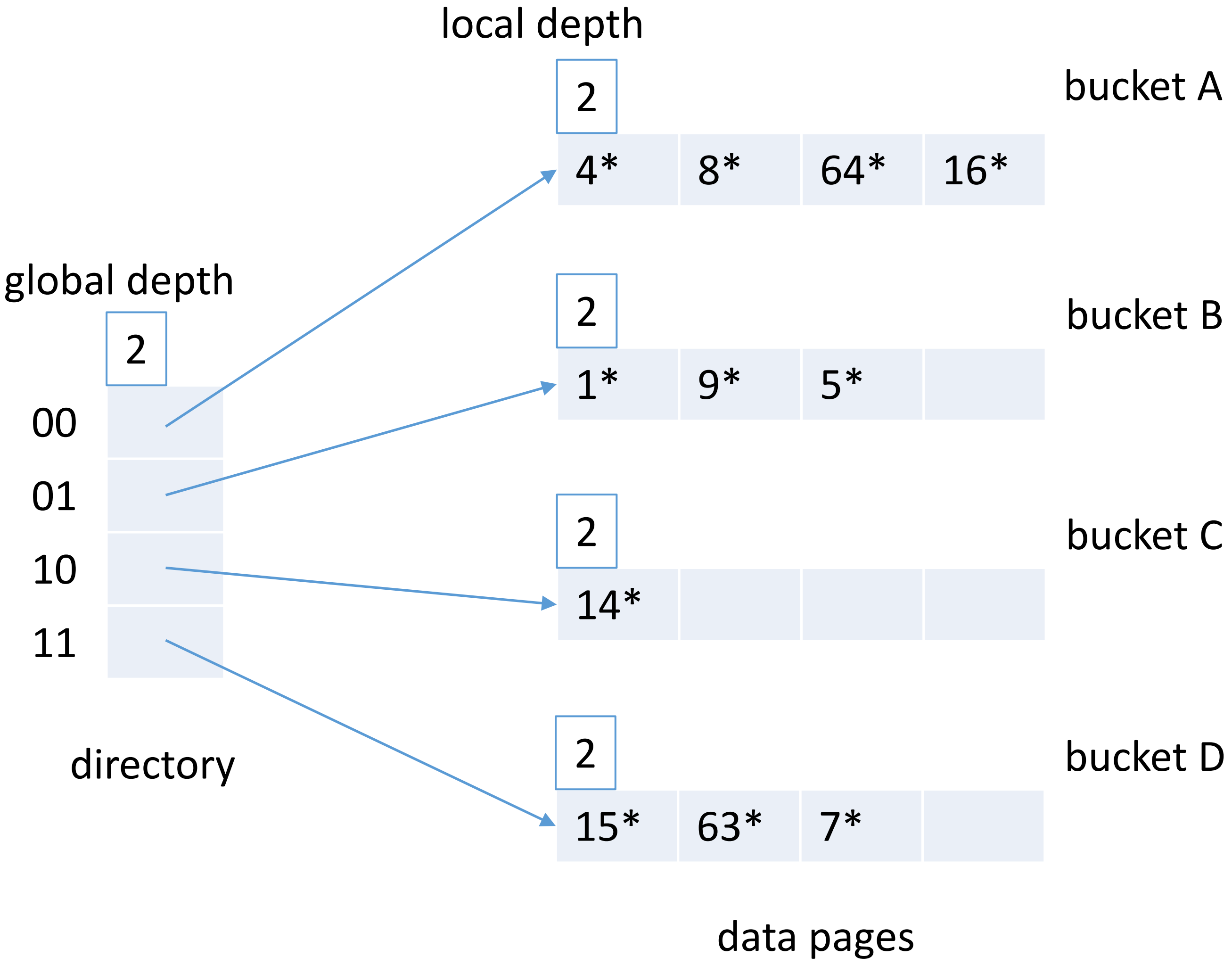
* static hashing

- main problem
  - fixed number of buckets
- solutions
  - periodic rehash
  - dynamic hashing

Extendible Hashing

* extendible hashing

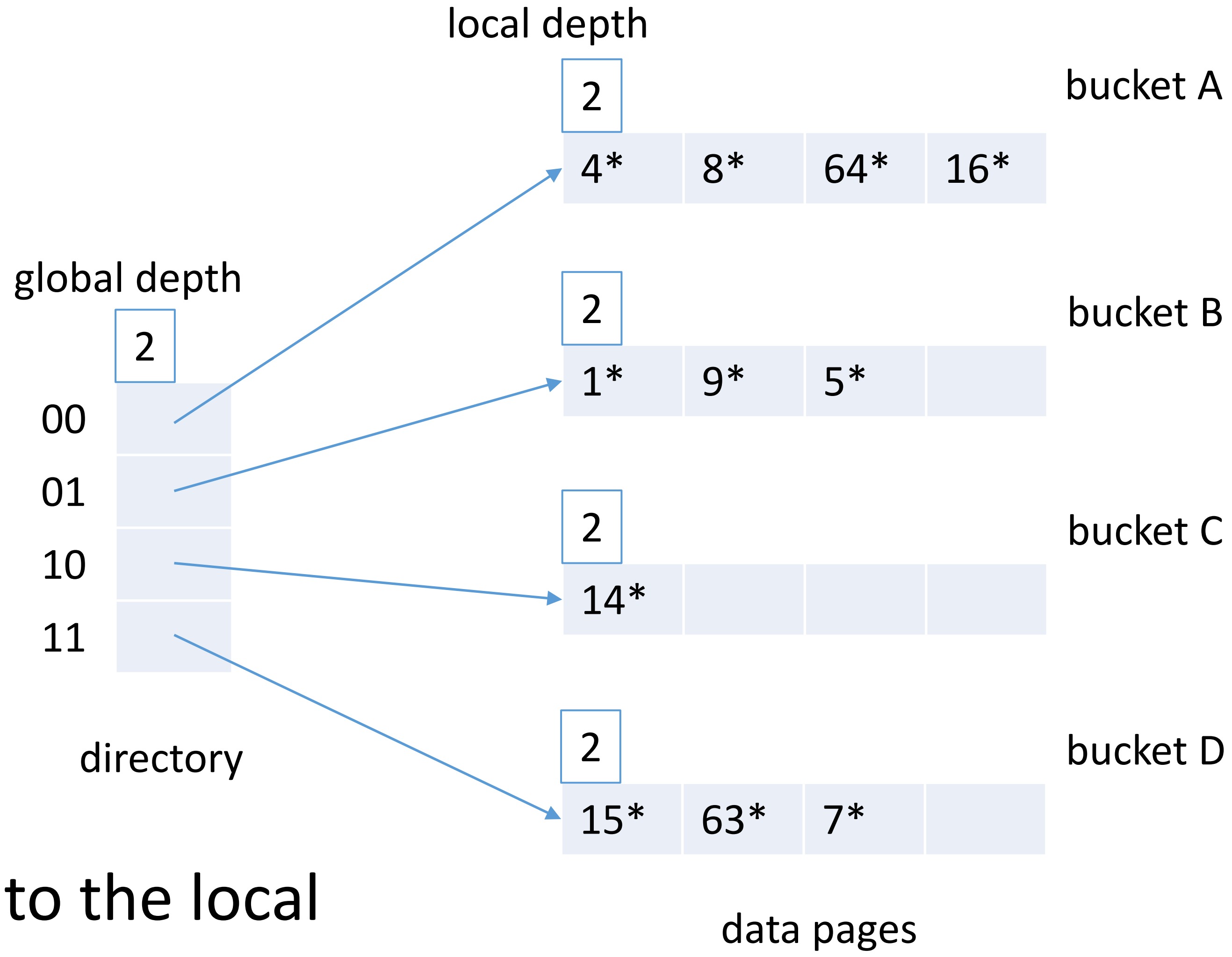
- dynamic hashing technique
- directory of pointers to buckets
- double the size of the number of buckets
  - double the directory
  - split overflowing bucket



- directory: array of 4 elements
- directory element: pointer to bucket
- entry r with key value K
- h(K) = $(\ldots a_2a_1a_0)_2$
- nr = a_1a_0, i.e., last 2 bits in $(\ldots a_2a_1a_0)_2$, nr between 0 and 3
- directory[nr]: pointer to desired bucket

* extendible hashing

- global depth *gd* of hashed file
  - number of bits at the end of hashed value interpreted as an offset into the directory
  - kept in the header
  - depends on the size of the directory
    - 4 buckets => gd = 2
    - 8 buckets => gd = 3
- initially, the global depth is equal to the local depth of every bucket



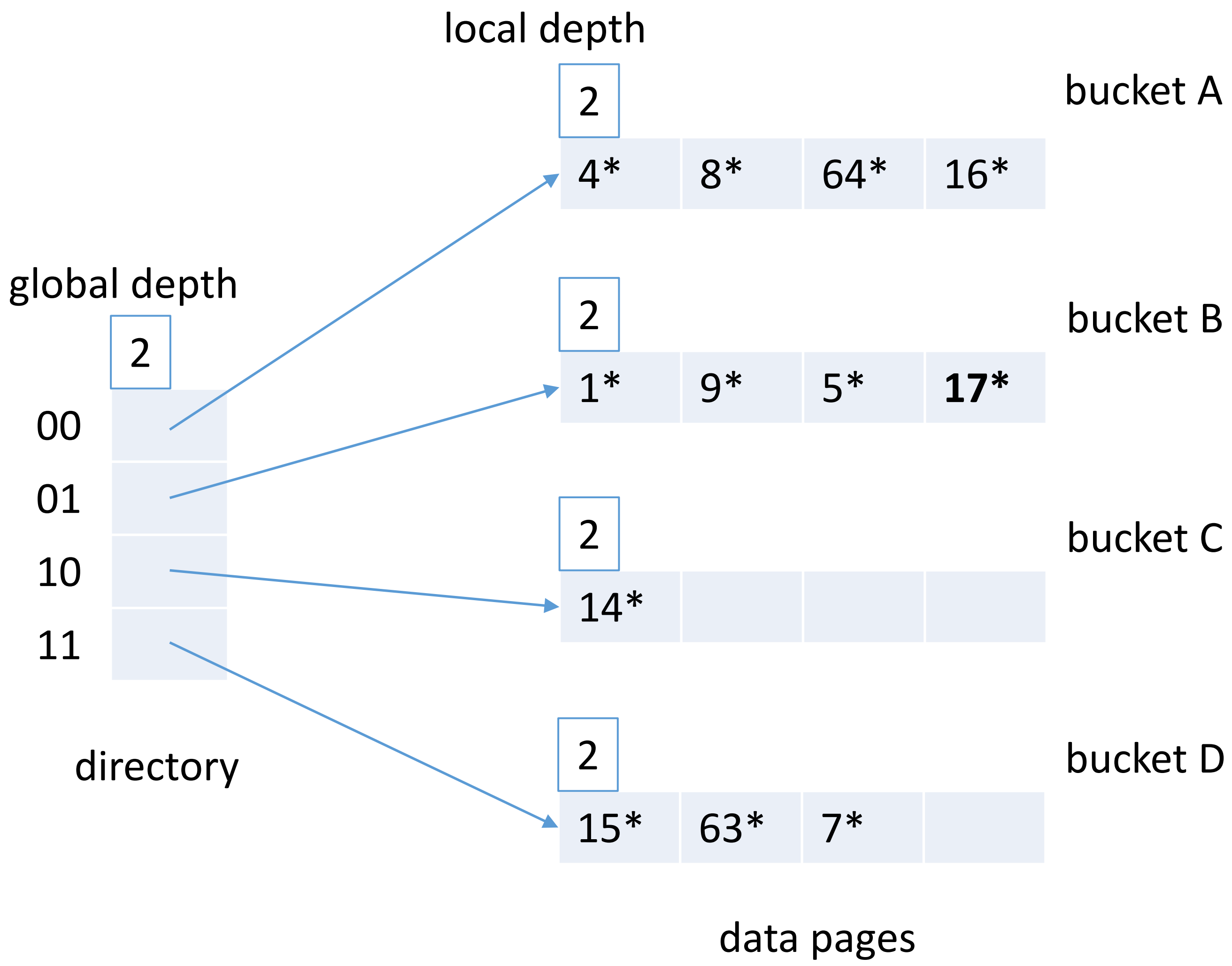
* extendible hashing

- insert entry
    - find bucket

  a. bucket has free space => the new value can be added

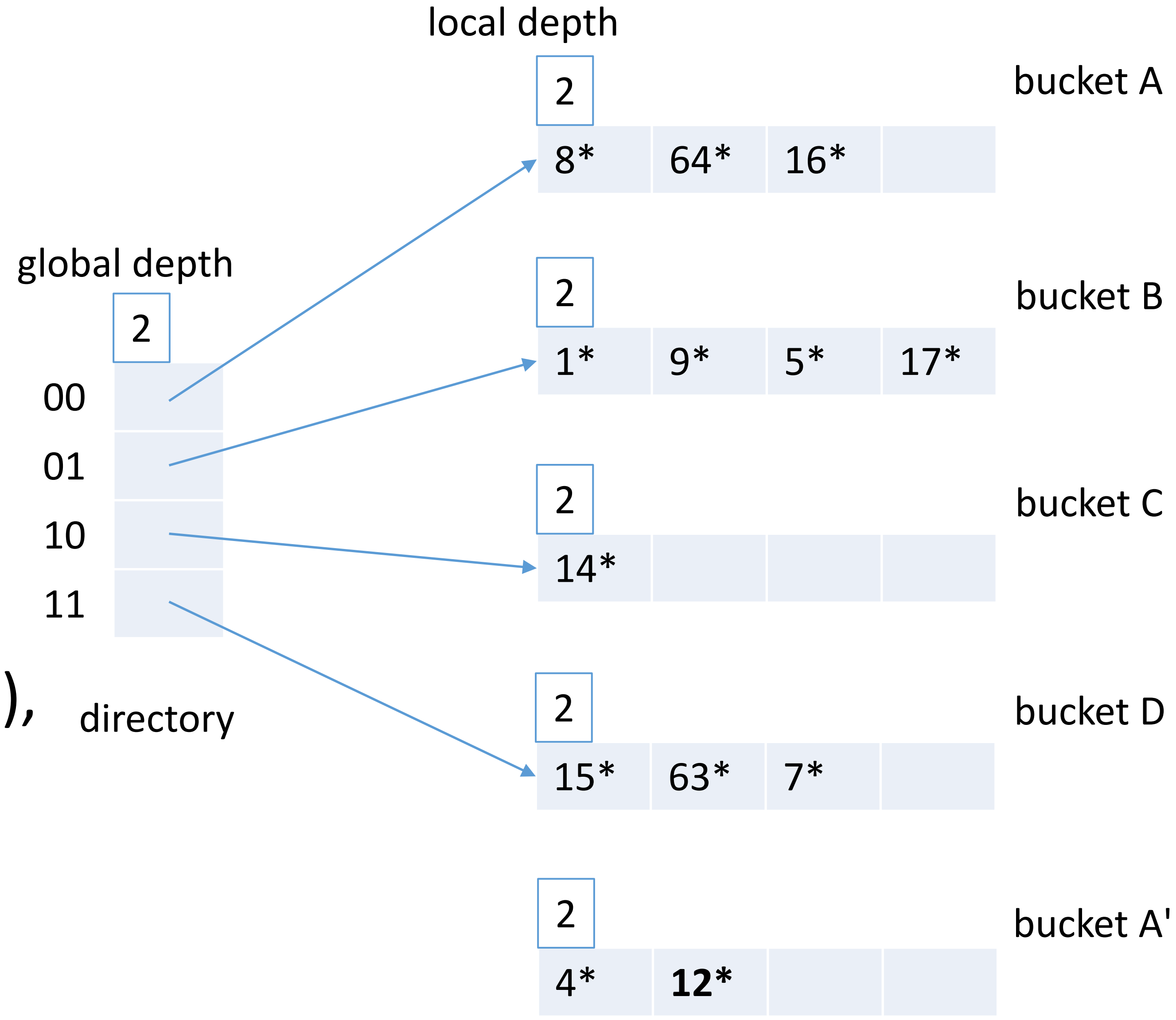
    - example: add data entry with hash value 17 to bucket B

obs. data entry with hash value 17 is denoted as 17*



* extendible hashing

- insert entry
  - b. bucket is full
    - example: add entry 12*, bucket A full
    - split bucket A
      - allocate new bucket A'
      - redistribute entries across A & A' (the split image of A), by taking into account the last 3 bits of h(K)

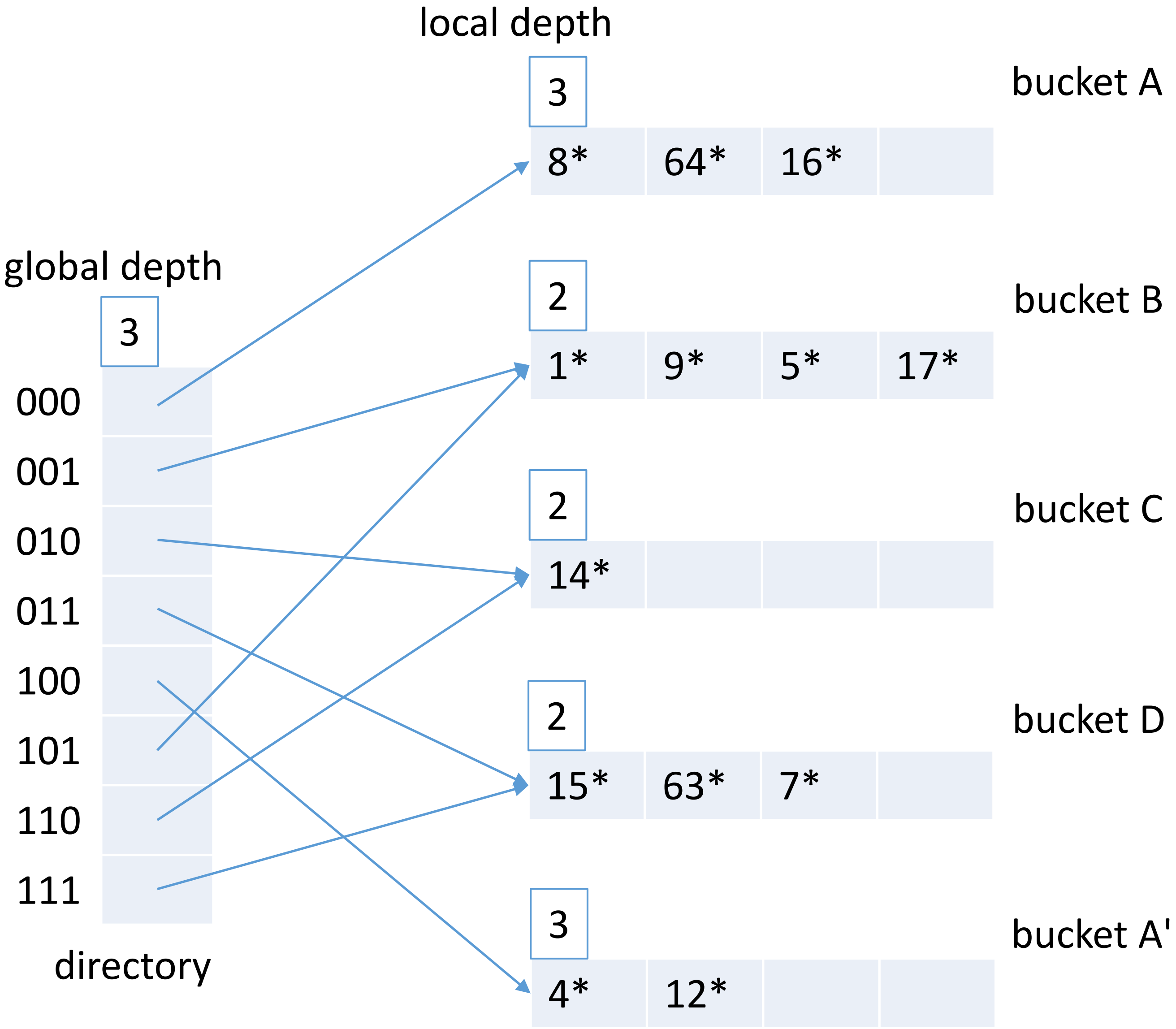


* extendible hashing

- insert entry

  b. bucket is full

  - if gd = local depth of bucket being split => double the directory, gd++
  - 3 bits are needed to discriminate between A & A', but the directory has only enough space to store numbers that can be represented on 2 bits, so it is doubled
  - increment local depth of bucket: LD(A) = 3
  - assign new local depth to bucket's split image: LD(A') = 3



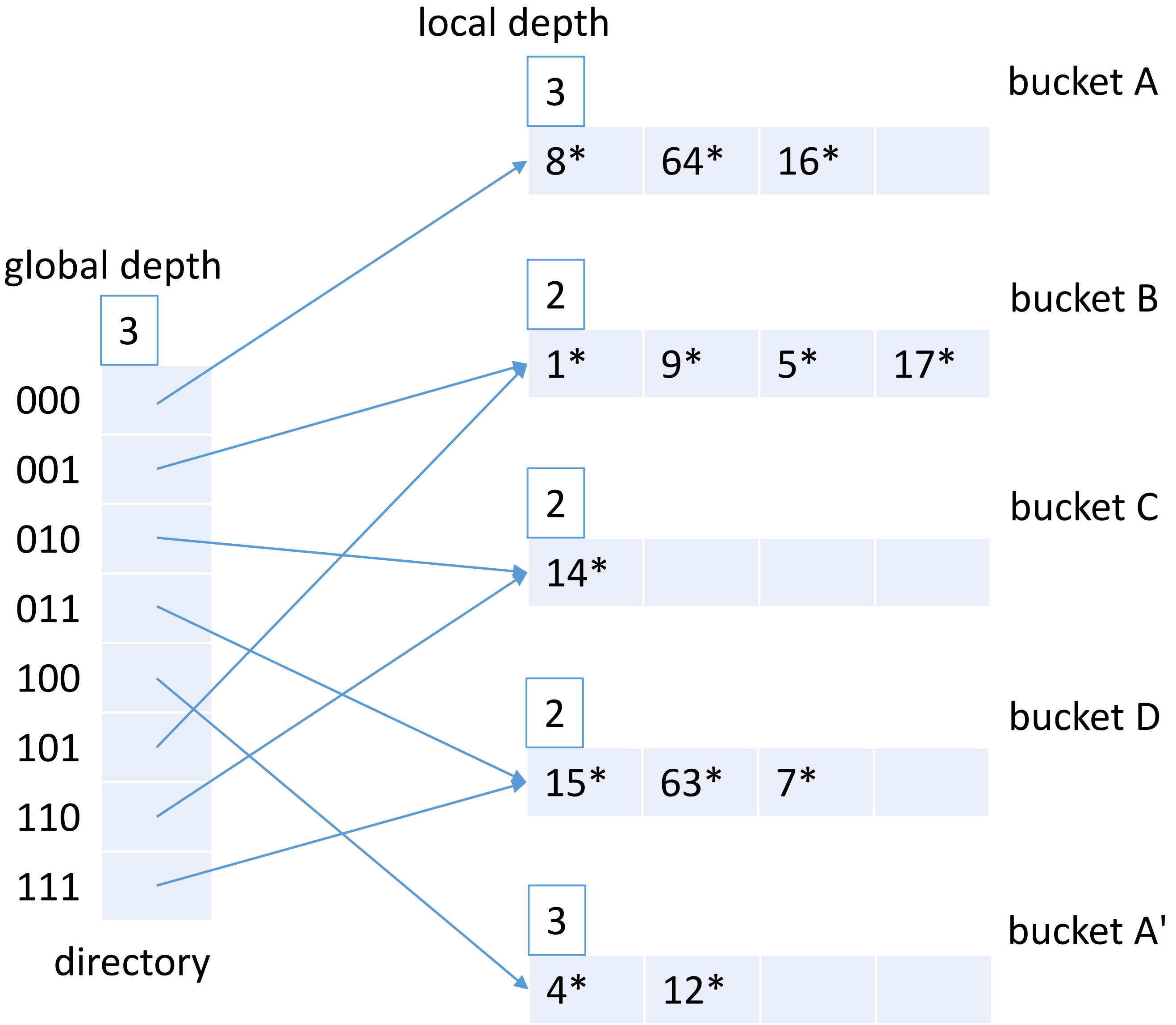
Sabina S. CS

* extendible hashing

- insert entry

  b. bucket is full

  - *corresponding elements*
    - 000, 100
    - 001, 101
    - 010, 110
    - 011, 111
    - point to the same bucket, except for 000 and 100, which point to A and split image A', respectively



local depth

global depth

3

000
001
010
011
100
101
110
111

directory

bucket A: 3 | 8* 64* 16*

bucket B: 2 | 1* 9* 5* 17*

bucket C: 2 | 14*

bucket D: 2 | 15* 63* 7*

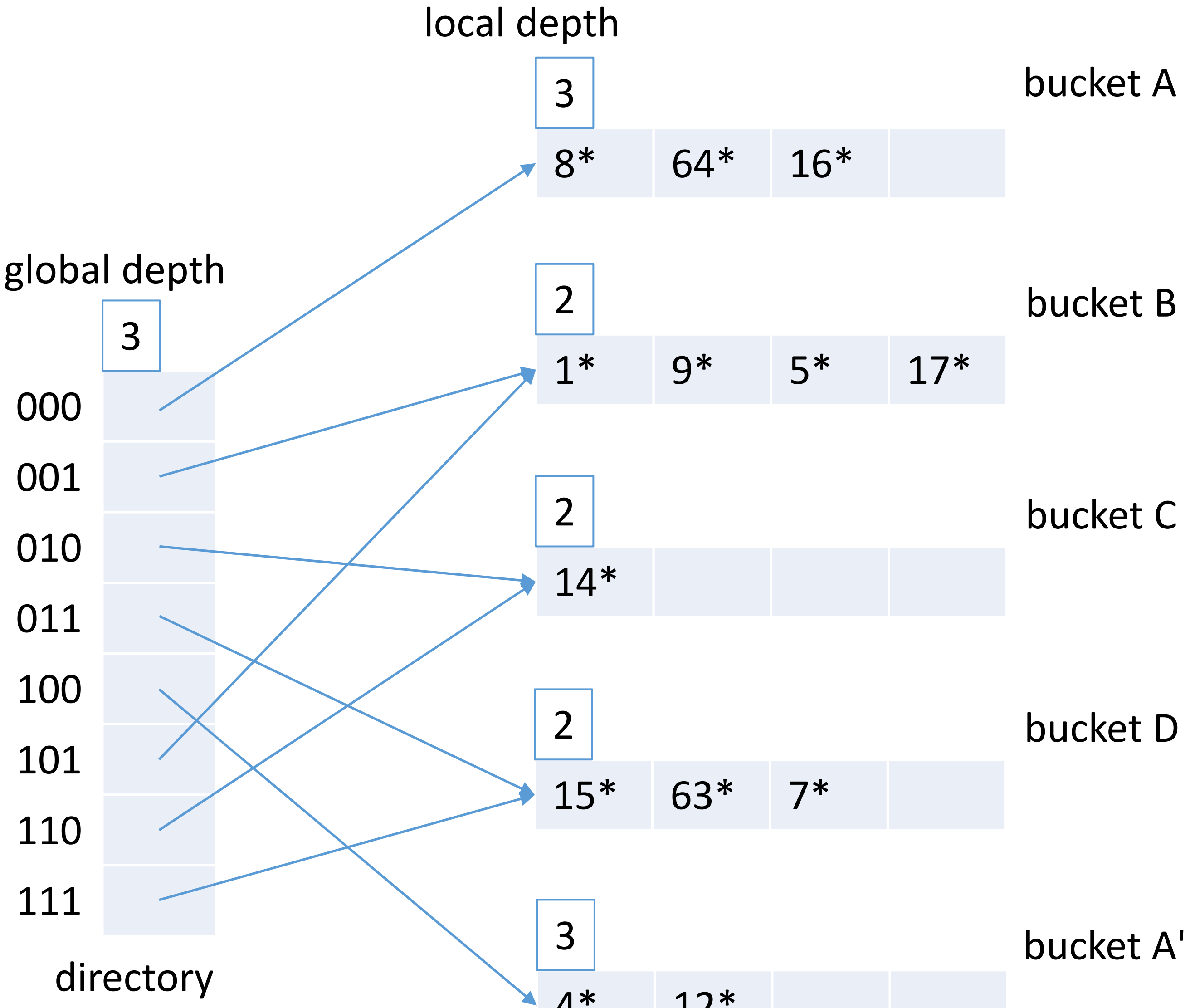
bucket A': 3 | 4* 12*

* extendible hashing

- insert entry

  b. bucket is full

  - example: add 21*
  - it belongs to bucket B, which is already full, but its local depth is 2 and gd = 3



->

Sabina S. CS

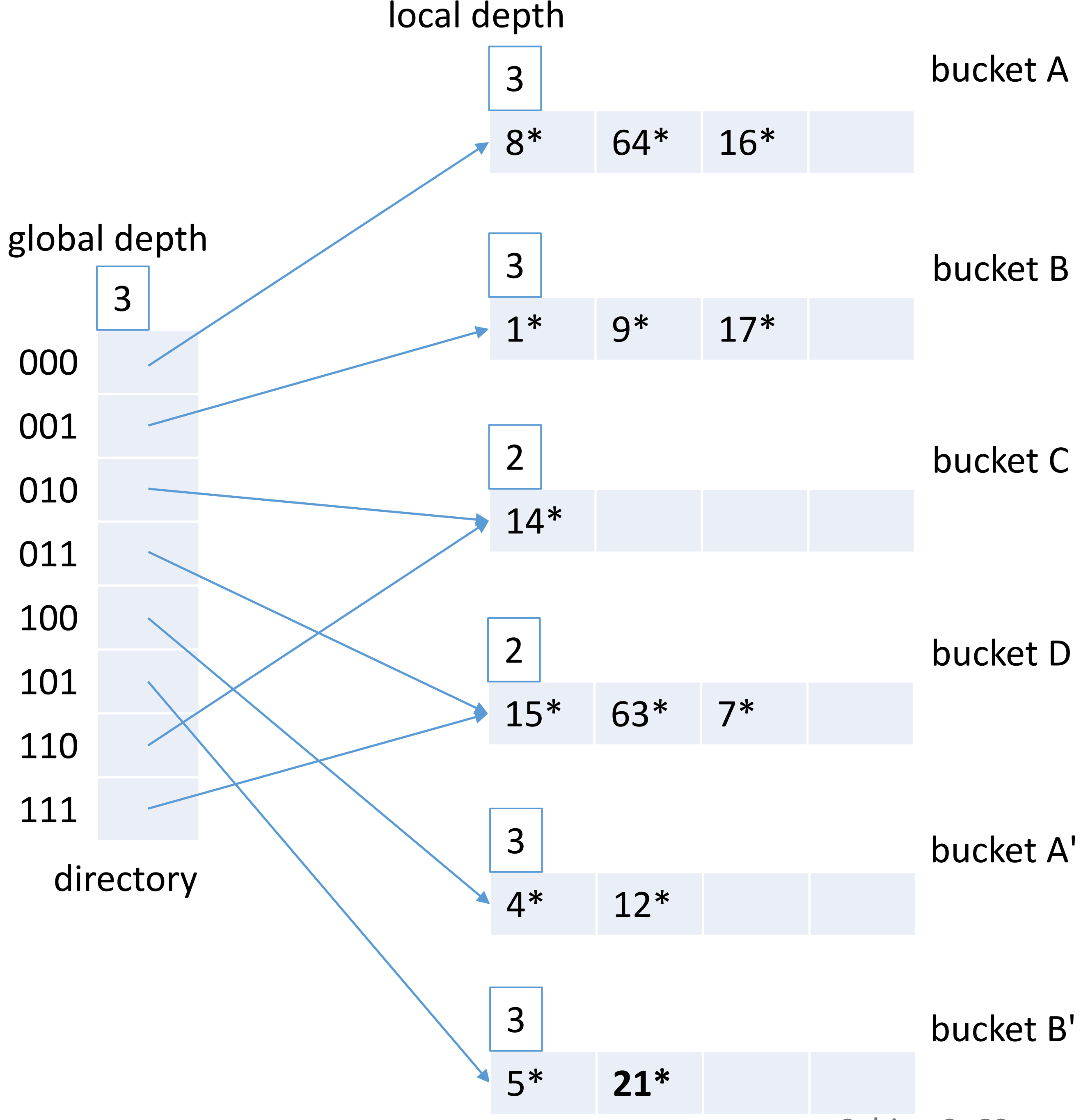
* extendible hashing

- insert entry

  b. bucket is full

  - example: add 21*
  - it belongs to bucket B, which is already full, but its local depth is 2 and gd = 3

  => split B, redistribute entries, increase local depth for B and its split image; directory isn't doubled, gd doesn't change



Sabina S. CS

* extendible hashing

- search for entry with key value K_0
  - compute $h(K_0)$
  - take last gd bits to identify directory element
  - search corresponding bucket
- delete entry
  - locate & remove entry
  - if bucket is empty:
    - merge bucket with its split image, decrement local depth
  - if every directory element points to the same bucket as its split image:
    - halve the directory
    - decrement global depth

* extendible hashing

- obs 1. 2^{gd-ld} elements point to a bucket Bk with local depth ld
  - if gd=ld and bucket Bk is split => double directory
- obs 2. manage collisions - overflow pages

- bucket split accompanied by directory doubling
  - allocate new bucket page nBk
  - write nBk and bucket being split
  - double directory array (which should be much smaller than file, since it has 1 page-id / element)
    - if using *least significant bits* (last gd bits) => efficient operation:
      - copy directory over
      - adjust split buckets' elements

* extendible hashing

- equality selection
- if directory fits in memory:

  => 1 I/O (as for Static Hashing with no overflow chains)
- otherwise
  - 2 I/Os

- example: 100 MB file, entry = 50 bytes => 2.000.000 entries
- page size = 8 KB => approx. 160 entries / bucket

=> need 2.000.000 / 160 = 12.500 directory elements

References

- [Ra00] 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