Databases

Lecture 9

Indexes. Trees

Indexes

Indexes

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

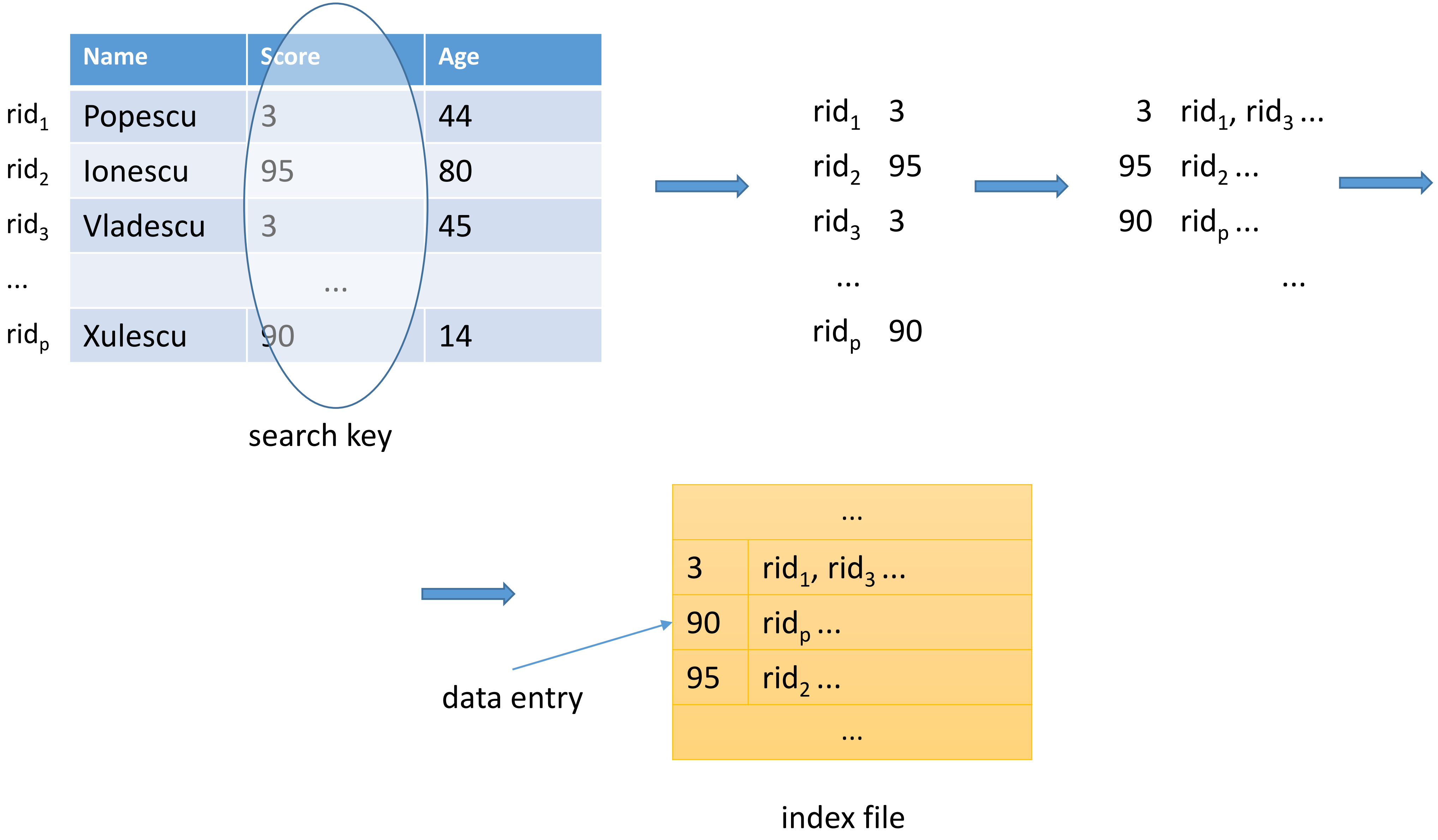
Indexes - Data Entries

- 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

Creating an index - example



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

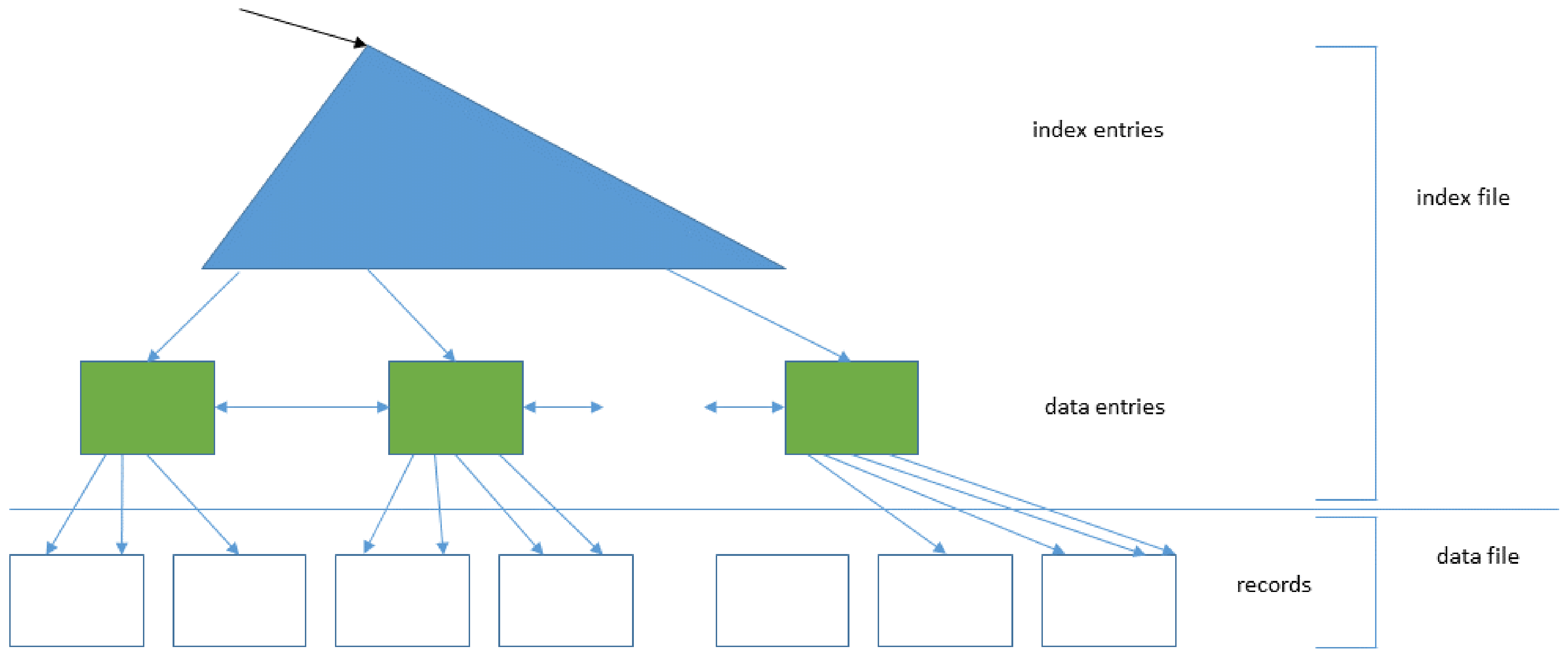
rid_1	3
rid_2	95
rid_3	3
…	
rid_p	90

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

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

Indexes - Properties

- clustered vs. unclustered
  - clustered index
    - the order of the data records is close to / the same as the order of the data entries
    - index entries guide the search for data entries

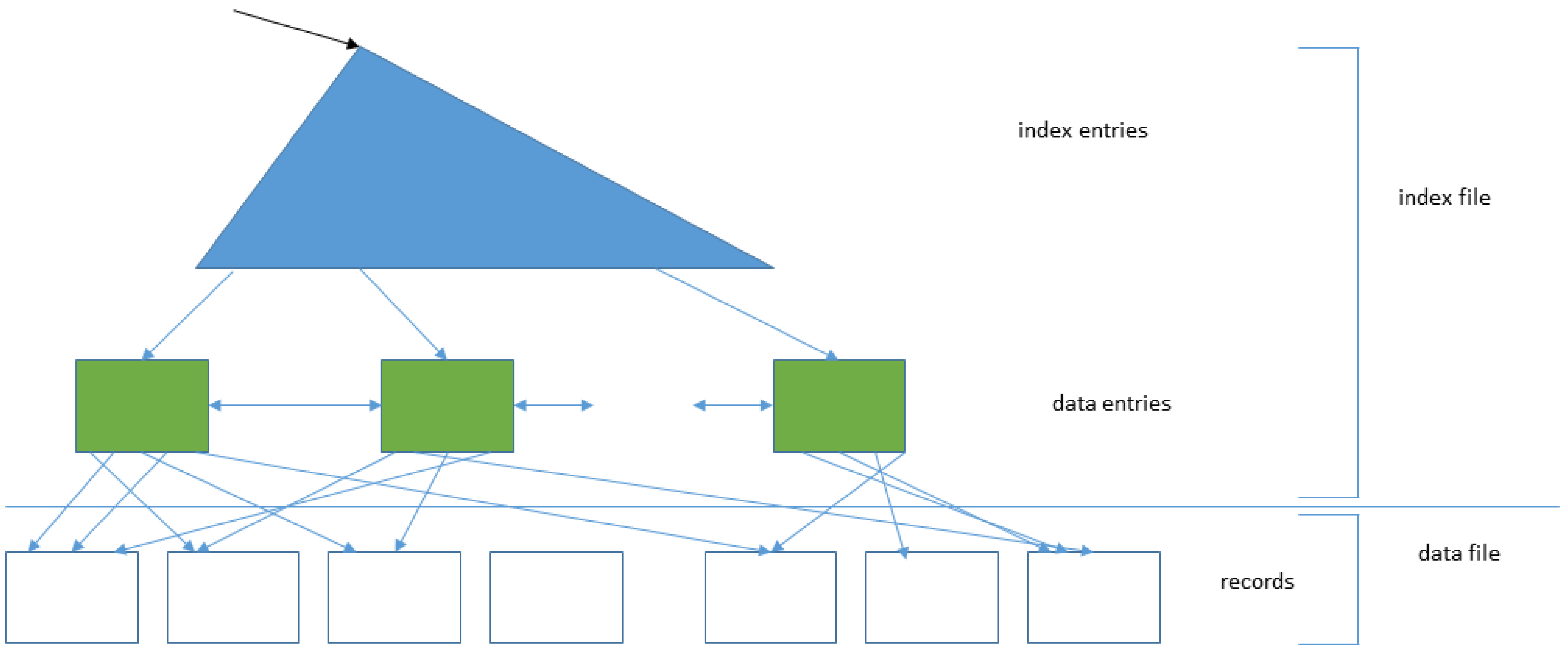


Indexes - Properties

- clustered vs. unclustered
  - a1 → clustered index
  - indexes using a2 / a3 are clustered only if the data records are ordered on the search key
  - at most 1 clustered index / data file
  - maintaining a clustered index
    - costly operation

Indexes - Properties

- clustered vs. unclustered
  - unclustered index
    - index that is not clustered



Indexes - Properties

- clustered vs. unclustered
  - unclustered index
    - several unclustered indexes could be built on a file

  - range search query
    - unclustered index usage cost
      - each data entry that meets the condition in a 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

Indexes - Properties

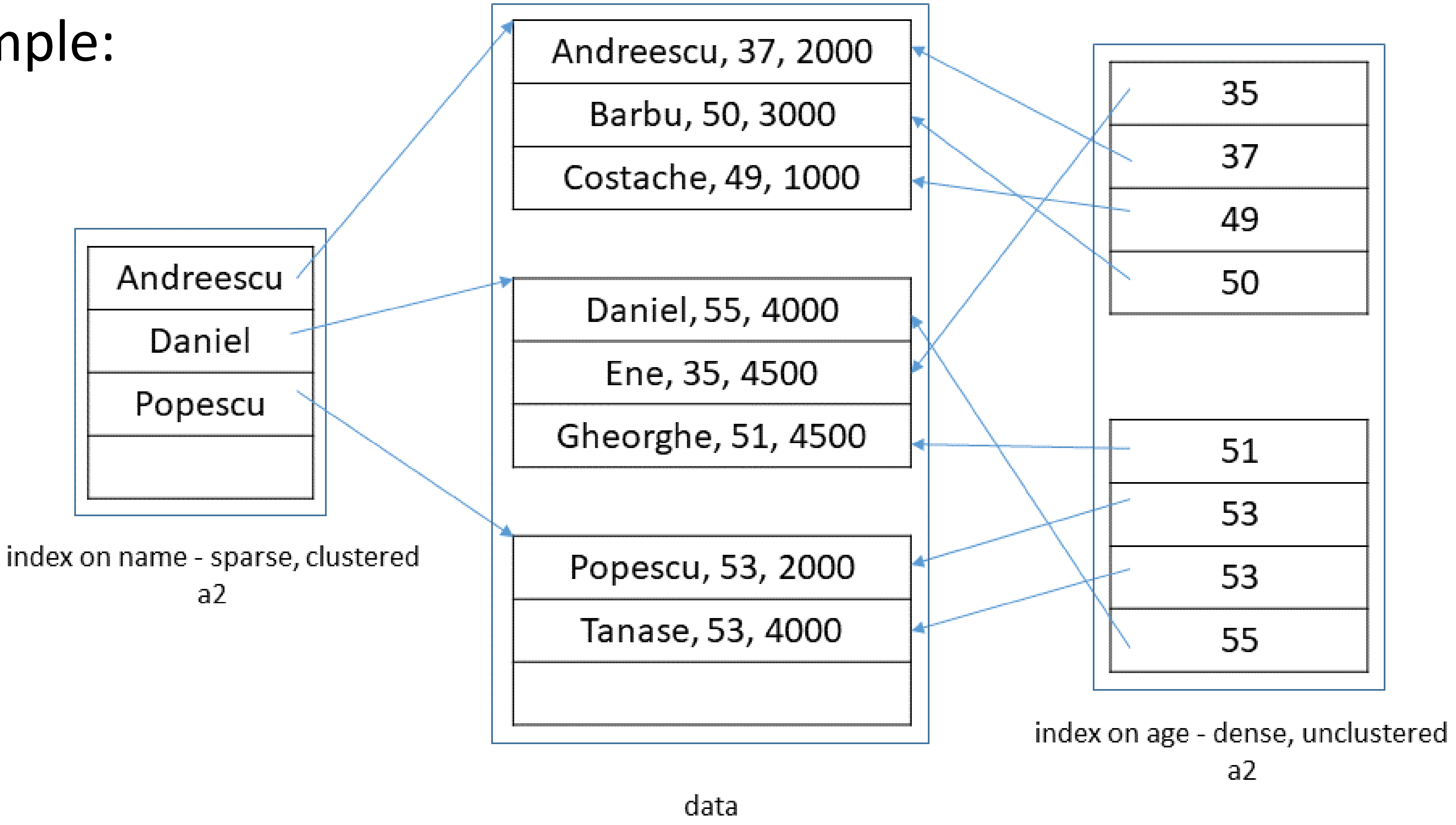
- primary and secondary
    - primary index
        - the search key includes the primary key
    - secondary index
        - index that is not primary
- unique index
    - the search key contains a candidate key
- duplicates
    - data entries with the same search key value
- primary indexes, unique indexes cannot contain duplicates
- secondary indexes can contain duplicates

Indexes - Properties

- sparse vs. dense
- dense index
  - at least one data entry for every search key value that appears in a data record
  - in the presence of duplicates and a2, multiple data entries can have the same search key value
- sparse index
  - one entry for each data page
  - more compact
- a1 => dense index
- sparse index => clustered

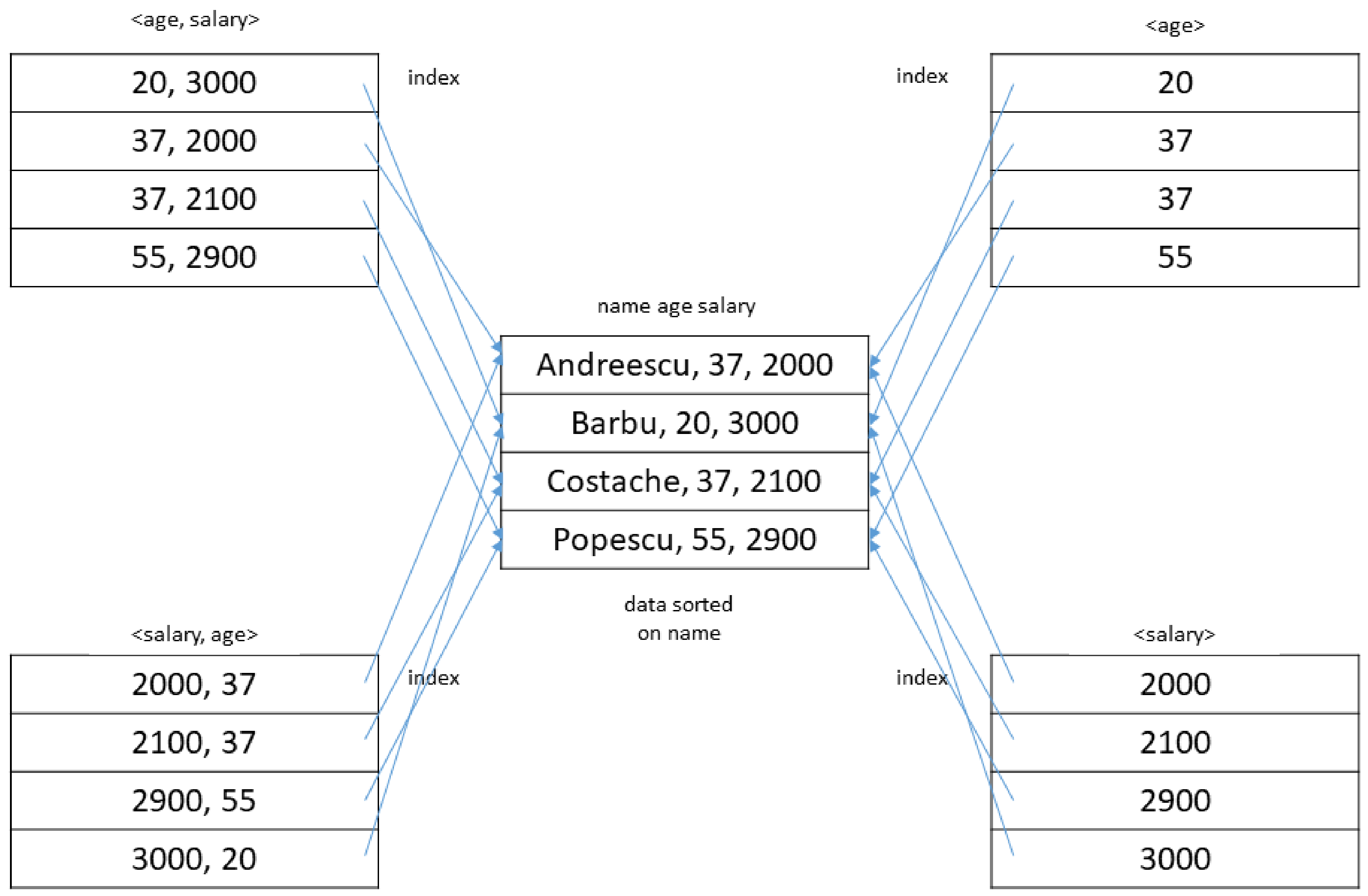
Indexes - Properties

- sparse vs. dense
  - example:



Indexes - Properties

- composite (concatenated) search key - search key that contains several fields
  - example:



* example

- the *Professors* relation sorted by age
- a page stores at most 3 records: the 1st tuple is in page 1 slot 1, the second - in page 1 slot 2, etc

prof_id	name	mail	age	impact_factor
4563	Popescu	pu@cs.edu	20	2.5
4570	Antonescu	au@cs.edu	25	3
4500	Ionescu	iu@by.edu	30	3.5
4555	Tanasescu	tu@cs.edu	65	3.1
4450	Tanasescu	tu@mc.edu	65	4

- dense index on *age* with a1 – the entire file
- dense index on *age* with a2 – data entries (20, <1, 1>), (25, <1, 2>), (30, <1, 3>), (65, <2, 1>), (65, <2, 2>)*

*recall that a record can be identified by <page id, slot number>

* example

prof_id	name	mail	age	impact_factor
4563	Popescu	pu@cs.edu	20	2.5
4570	Antonescu	au@cs.edu	25	3
4500	Ionescu	iu@by.edu	30	3.5
4555	Tanasescu	tu@cs.edu	65	3.1
4450	Tanasescu	tu@mc.edu	65	4

- dense index on *age* with a3 – entries (20, <1, 1>), (25, <1, 2>), (30, <1, 3>), (65, <2, 1>, <2, 2>)
- sparse index on *age* with a1 – cannot be built (by definition)
- sparse index on *age* with a2 – (20, <1, 1>), (65, <2, 1>)
- sparse index on *age* with a3 – (20, <1, 1>), (65, <2, 1>, <2, 2>)

* example

prof_id	name	mail	age	impact_factor
4563	Popescu	pu@cs.edu	20	2.5
4570	Antonescu	au@cs.edu	25	3
4500	Ionescu	iu@by.edu	30	3.5
4555	Tanasescu	tu@cs.edu	65	3.1
4450	Tanasescu	tu@mc.edu	65	4

- dense index on *impact_factor* with a2 – (2.5, <1, 1>), (3, <1, 2>), (3.1, <2, 1>), (3.5, <1, 3>), (4, <2, 2>)
- dense index on *impact_factor* with a3 – (2.5, <1, 1>), (3, <1, 2>), (3.1, <2, 1>), (3.5, <1, 3>), (4, <2, 2>)
- sparse index on *impact_factor* with a1 – cannot be built (by definition)
- sparse index on *impact_factor* with a2 or a3 – the values in the search key are not ordered

Data Collection Stored in a Binary Tree

- binary search algorithm on ordered collections
  - very fast; reduced search time (+)
  - maintaining the records' sort order - costly (especially for dynamic collections with many INSERT, UPDATE, DELETE operations) (-)
- solution
  - store the collection using a *binary tree*
  - record - stored in a node
  - node with key value *v*
    - left subtree
      - records with key values < *v*
    - right subtree
      - records with key values > *v*

- memory structure for a binary tree node
    - record's values stored in K, INF

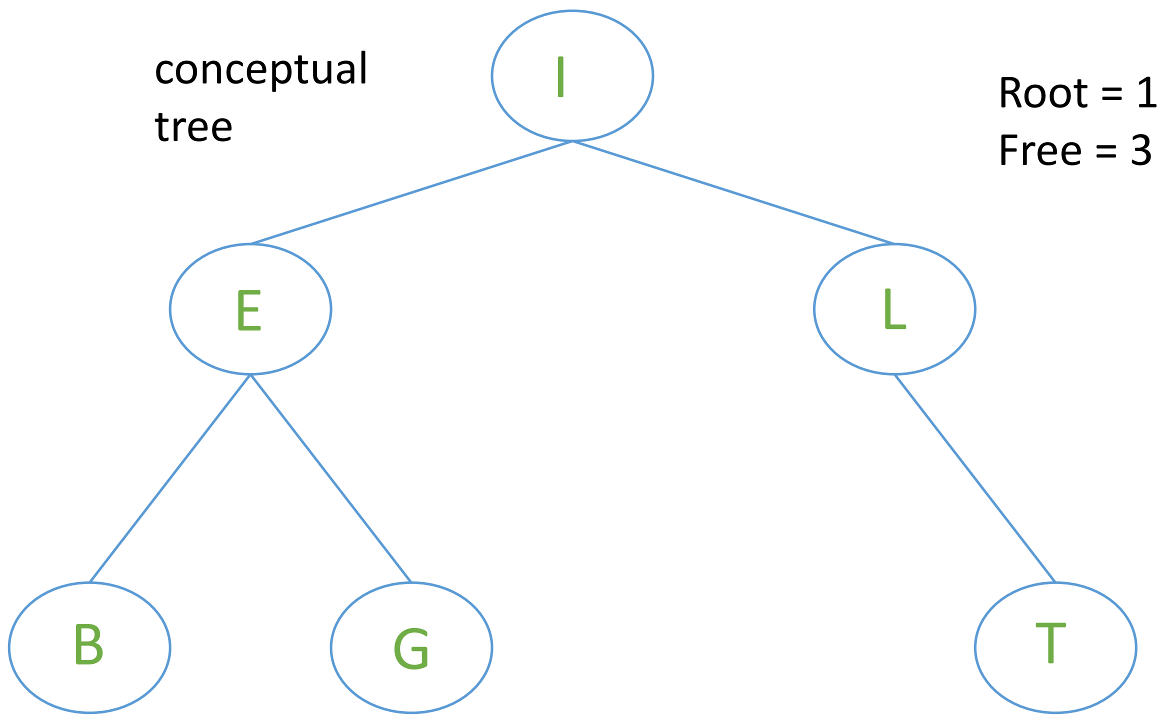
r

PointerL	**PointerR**	**K**	**INF**

- memory structure for a binary tree file
    - collection of nodes
    - pointer to the root
    - list of empty nodes
        - linked by PointerL

* Example

- Root - pointer to the root
- Free - pointer to the head of the empty nodes list



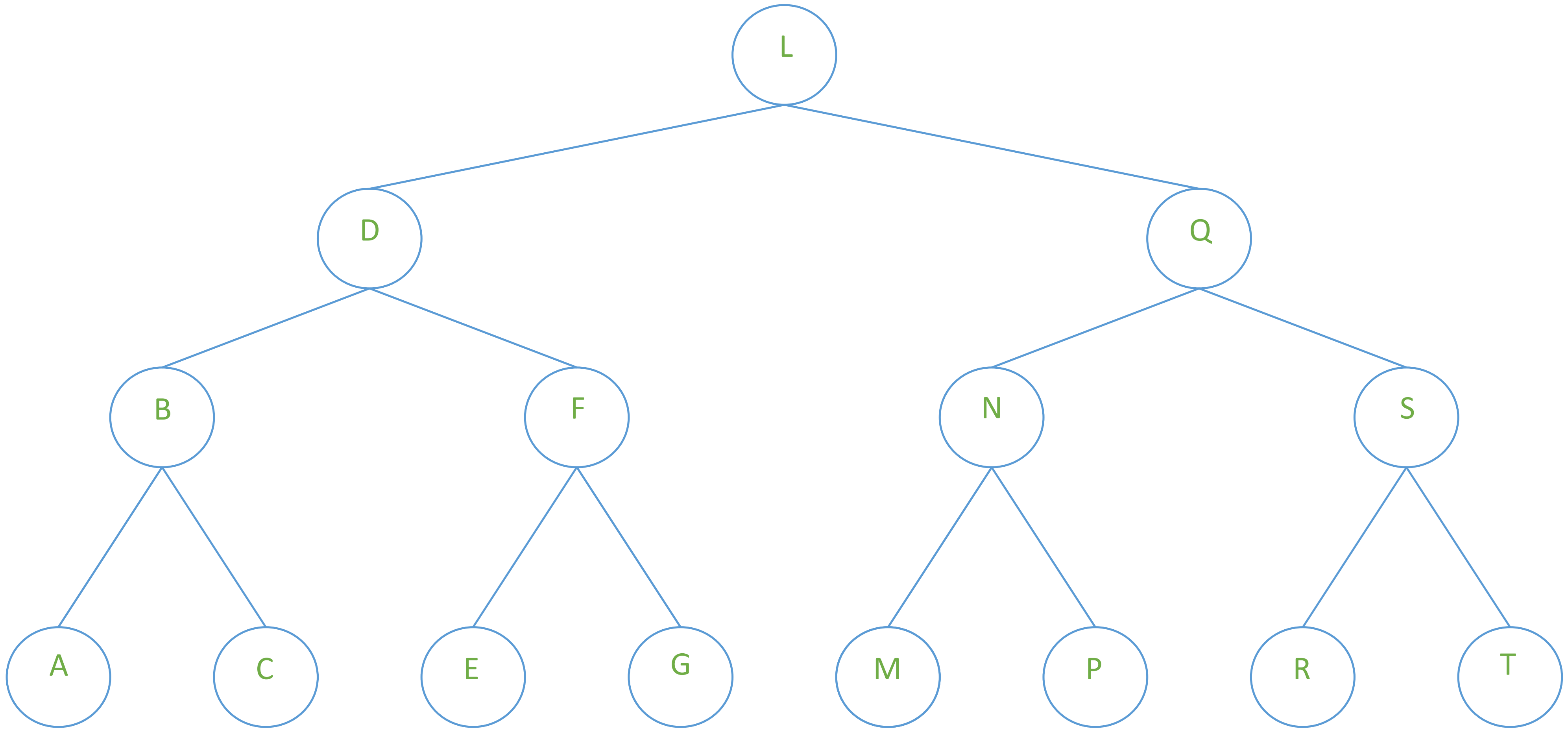
Root = 1

Free = 3

1	2	4	I	INF_I
2	8	7	E	INF_E
3	-6	NULL		
4	NULL	5	L	INF_L
5	NULL	NULL	T	INF_T
6	-9	NULL		
7	NULL	NULL	G	INF_G
8	NULL	NULL	B	INF_B
9	NULL	NULL		

- operations in a binary tree (BT)
  - searching for a record with key value K_0
  - inserting a record
  - removing a record
  - traversal - partial (between 2 values) / total

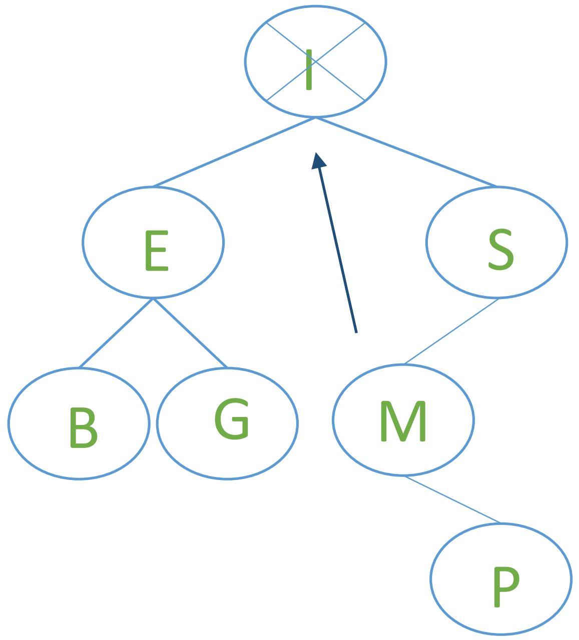
* Example - binary tree (only key values are shown)
    - key values: {L, D, B, Q, N, F, S, R, T, M, E, G, P, A, C}



- searching for a record
  - search starts at the root, follows PointerL / PointerR - based on a comparison result, a subtree is not considered anymore
  - how many comparisons to find record with key value M in the previous tree?
  - analysis, e.g.:
    - collection of English texts; tuples: (L, 50k), (D, 100k), etc
    - analyzing the most common letters in English; how many times does a letter appear in the collection?

- inserting a record
    - detect the position of the record
    - store the record in a free node
    - link the node to its parent

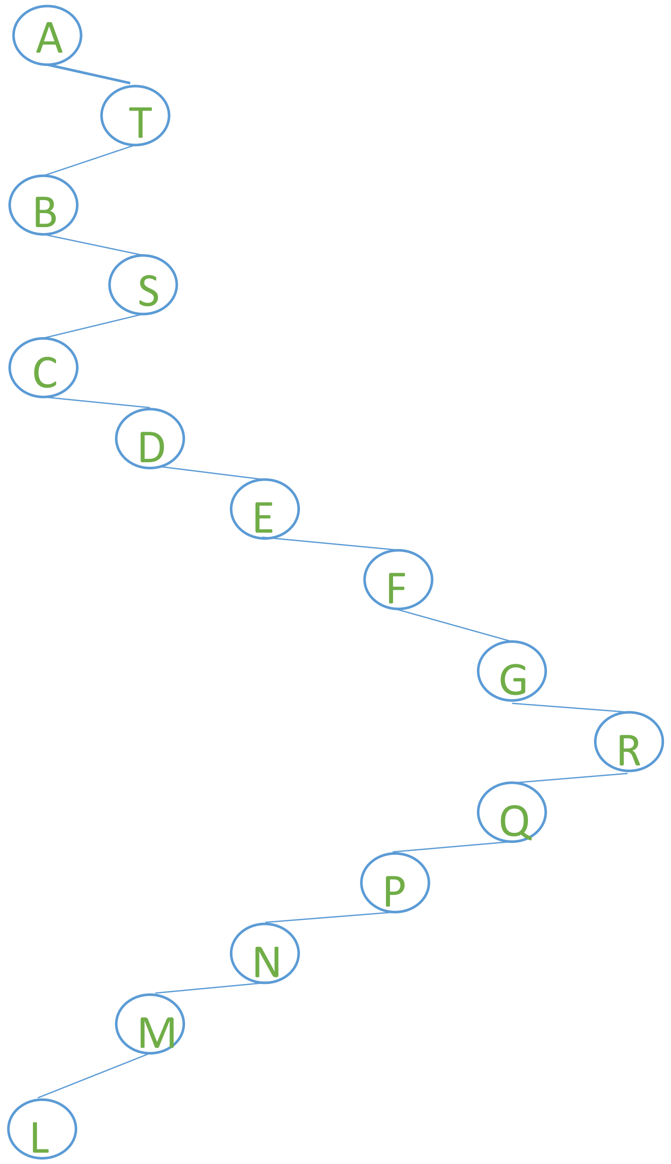
- removing a record
  - search for the record
  - 3 cases
    - it doesn't have children
      - the parent's pointer := NULL
    - it has 1 child
      - the child is attached to the parent
    - it has 2 children
      - it's replaced with the closest value (e.g., in-order successor)
  - the node is added to the empty nodes list



* Example

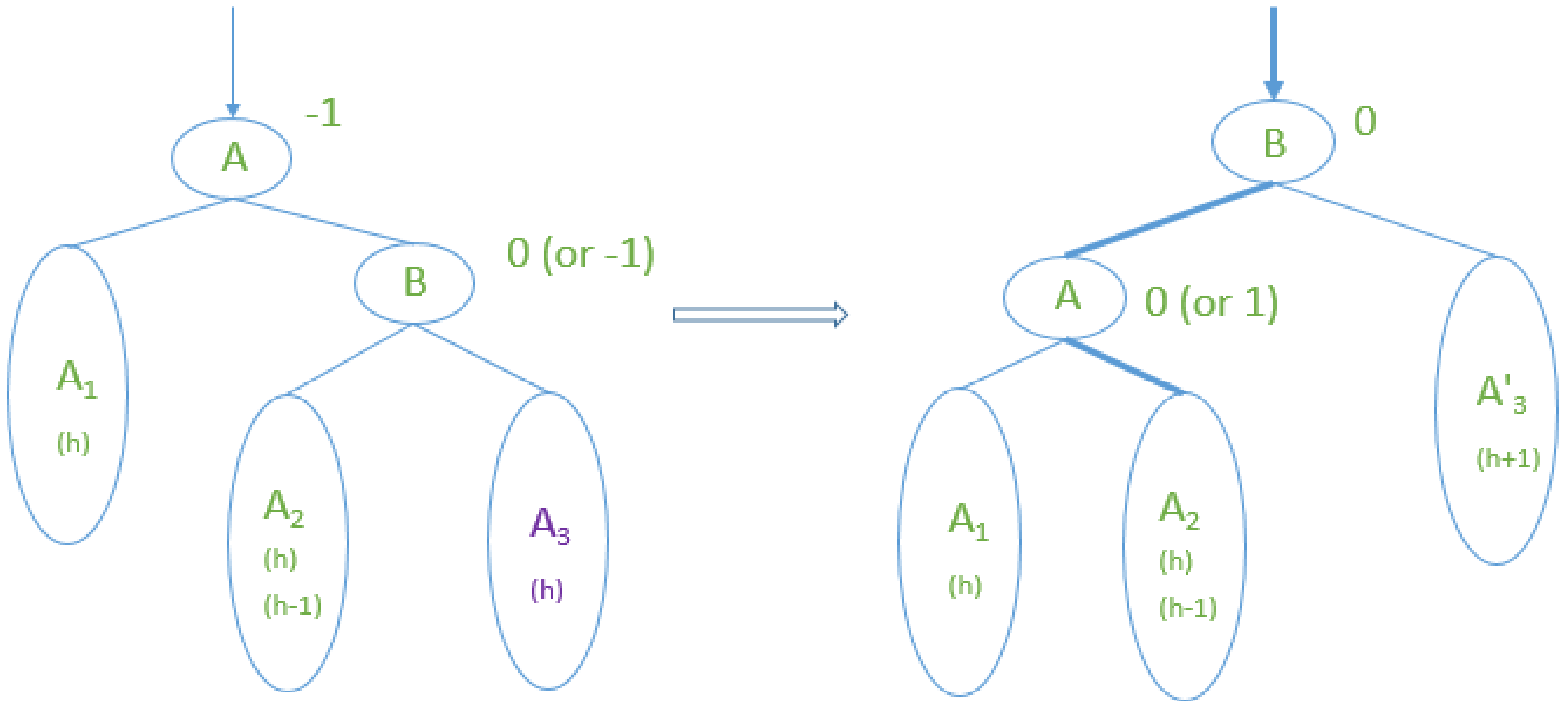
- key values {L, D, B, Q, N, F, S, R, T, M, E, G, P, A, C} are now provided in a different order: {A, T, B, S, C, D, E, F, G, R, Q, P, N, M, L}
- the tree on the right is obtained
- how many comparisons to find record with key value M?

- *degenerate tree*
  - sequential search
- the shape of the tree, and hence the search time, depends on the order in which data is added to the collection
- minimum search time - an *optimal tree*, i.e., one in which terminals are stored on at most 2 consecutive levels



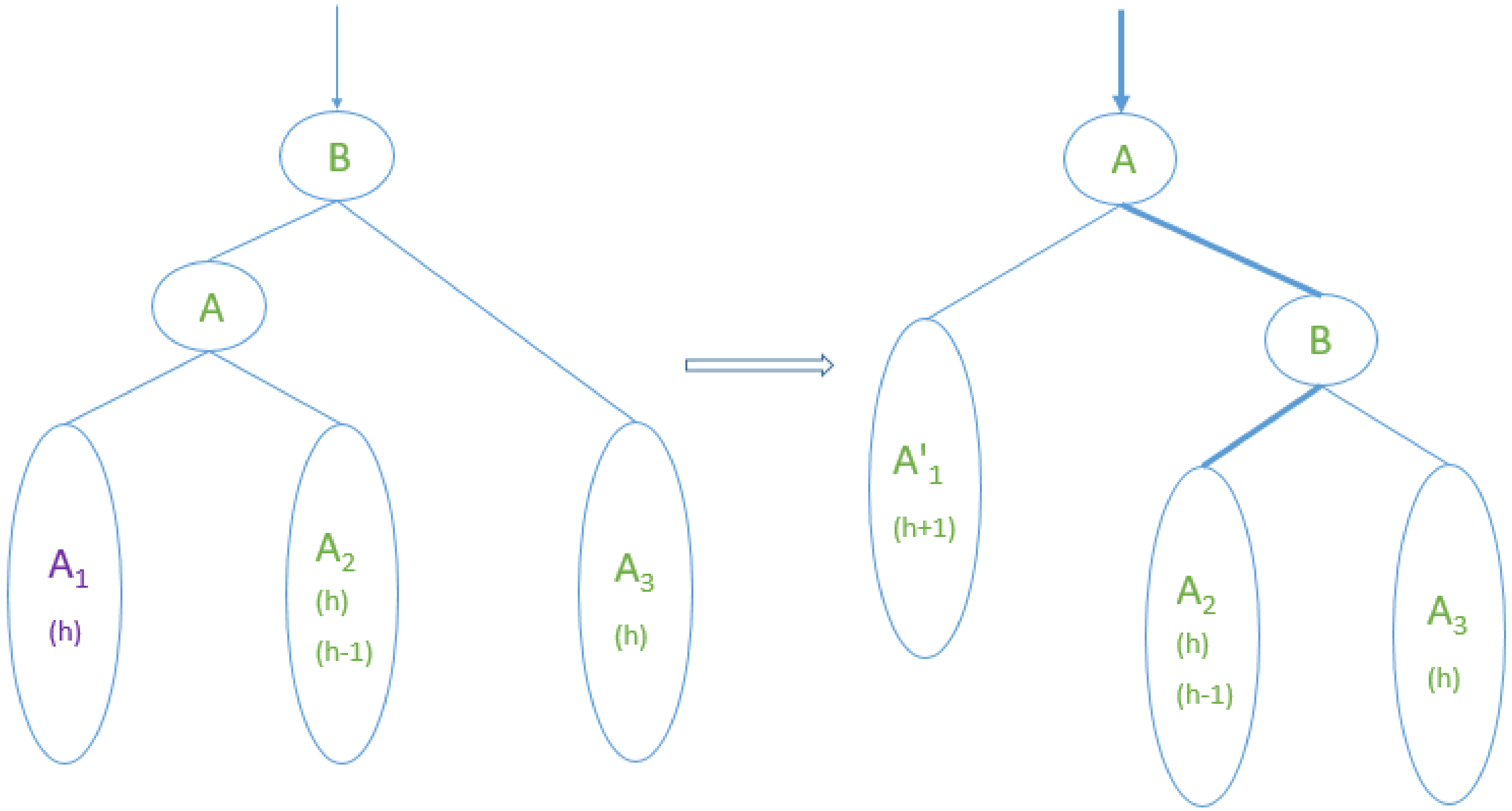
- definitions
  - *height* of the tree
    - length of longest path from root to terminals
  - a binary tree is *balanced* if, for every node N, the difference between the heights of N's subtrees is 0, 1 or -1
- obs. operations on a balanced tree can unbalance it; the tree can be rebalanced through a small number of changes
- a value is added to the A_3 tree on the next page (v1), causing its height to change (it increases by 1)
  - the subtrees' heights are shown between parentheses; for nodes A and B, the difference between the heights of their subtrees is also shown
  - after the insertion, the tree becomes unbalanced
  - the right-hand side of the figure shows a transformation that rebalances the tree with the root in A

v1

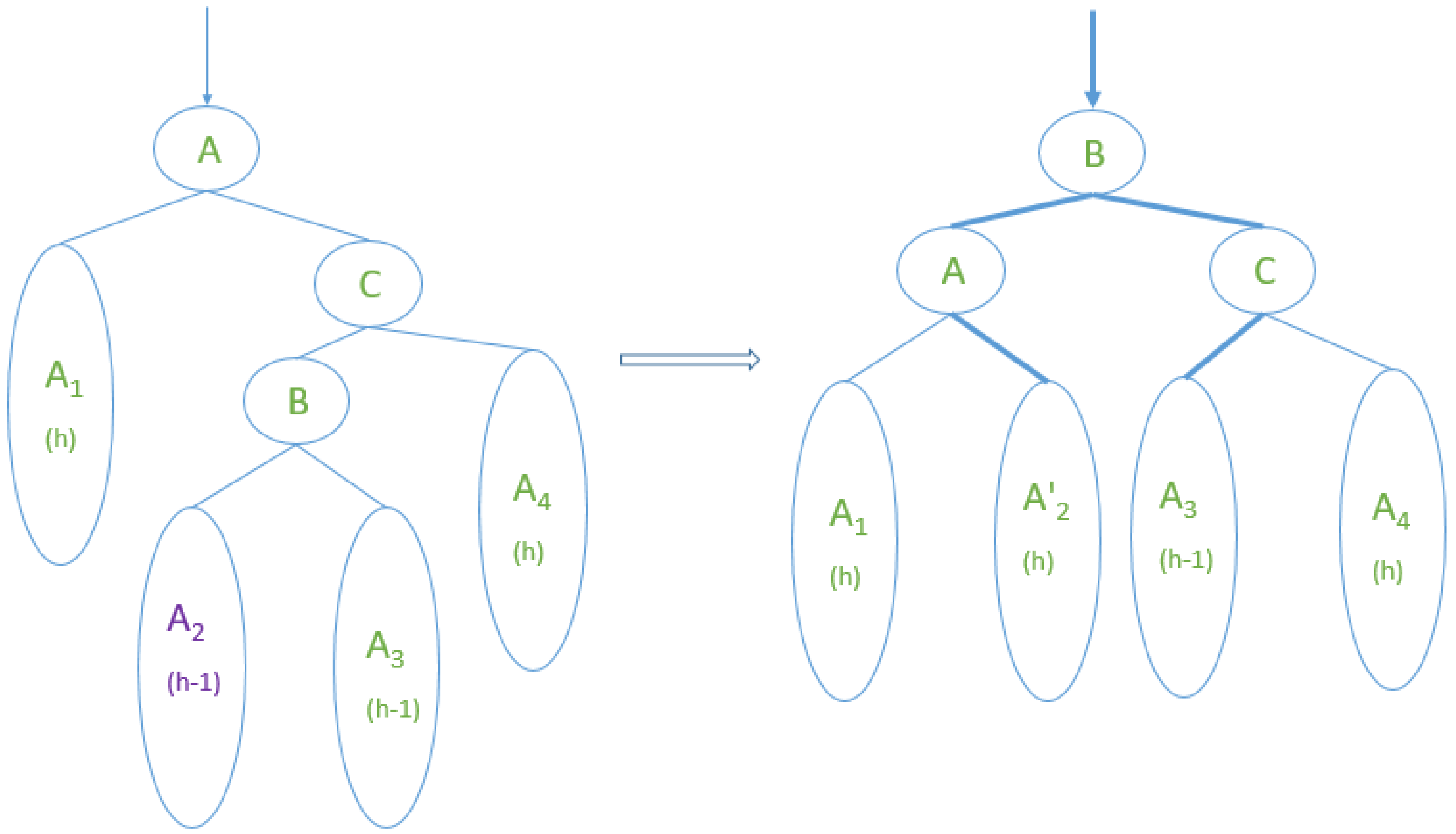


- some assignments are necessary for this rebalancing in the memory area that stores the tree (PointerR for A, PointerL for B, pointer referring to A will refer to B)
- [Kn76] enumerates all 6 rebalancing transformations

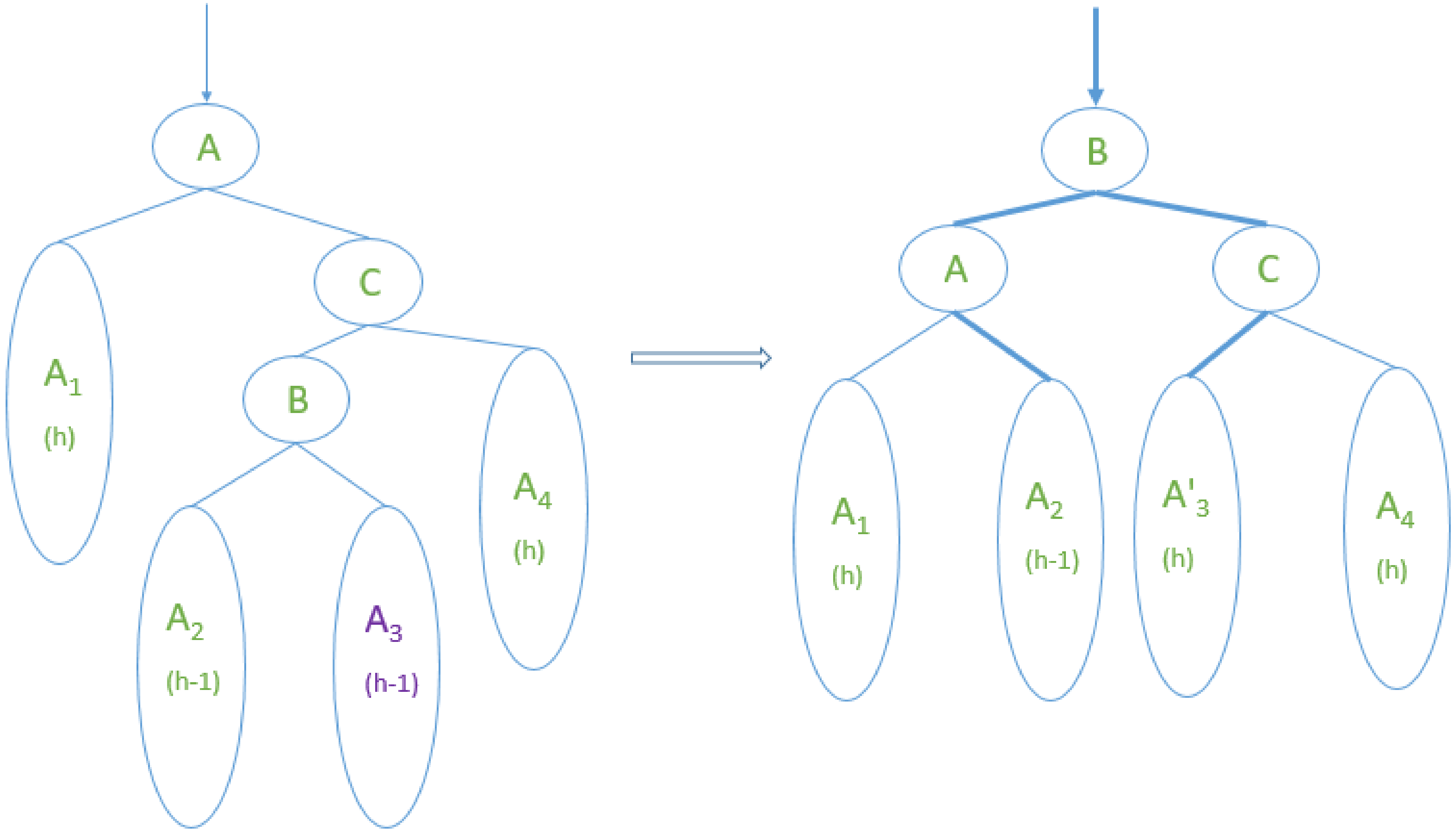
v2



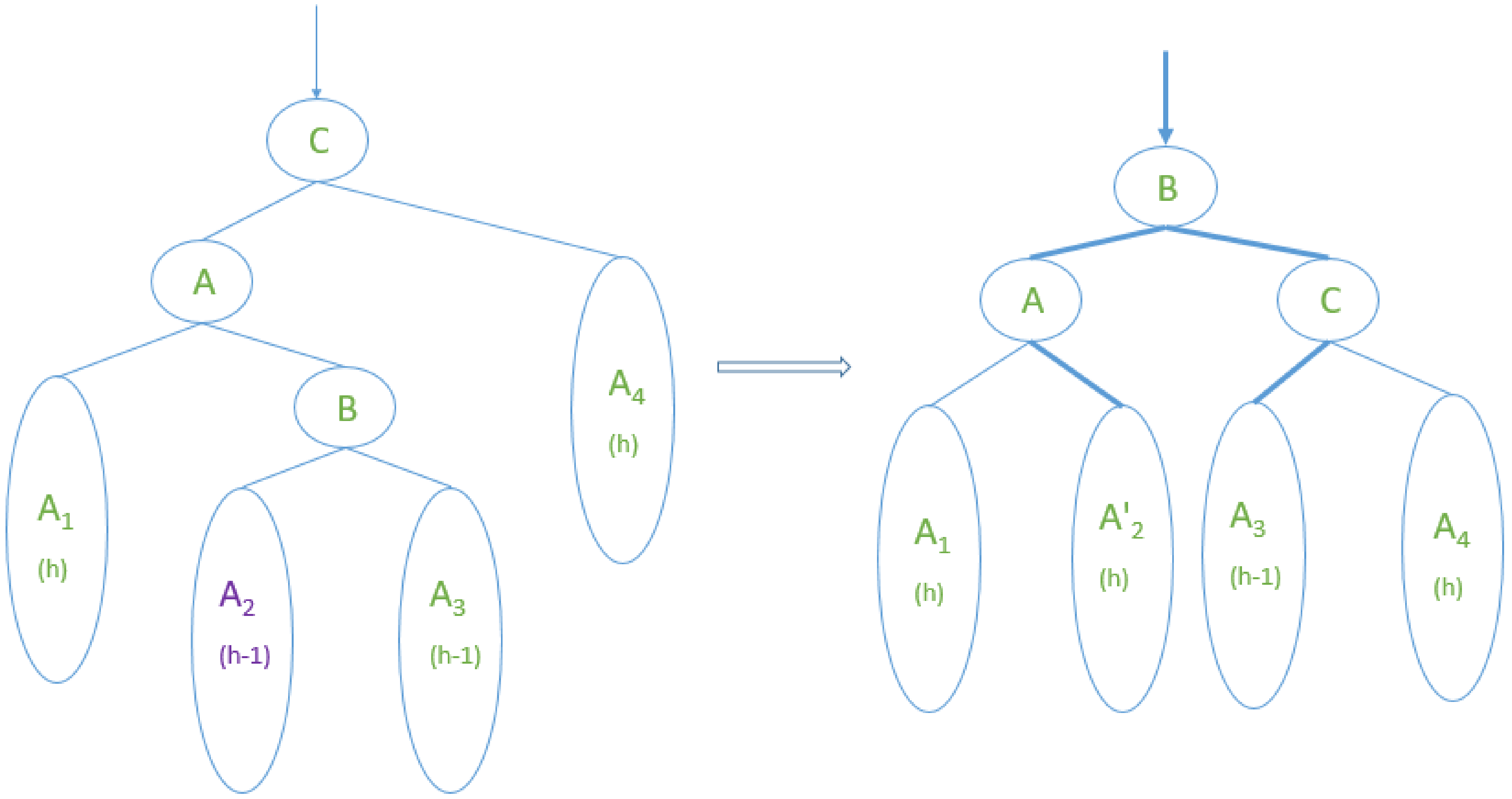
v3



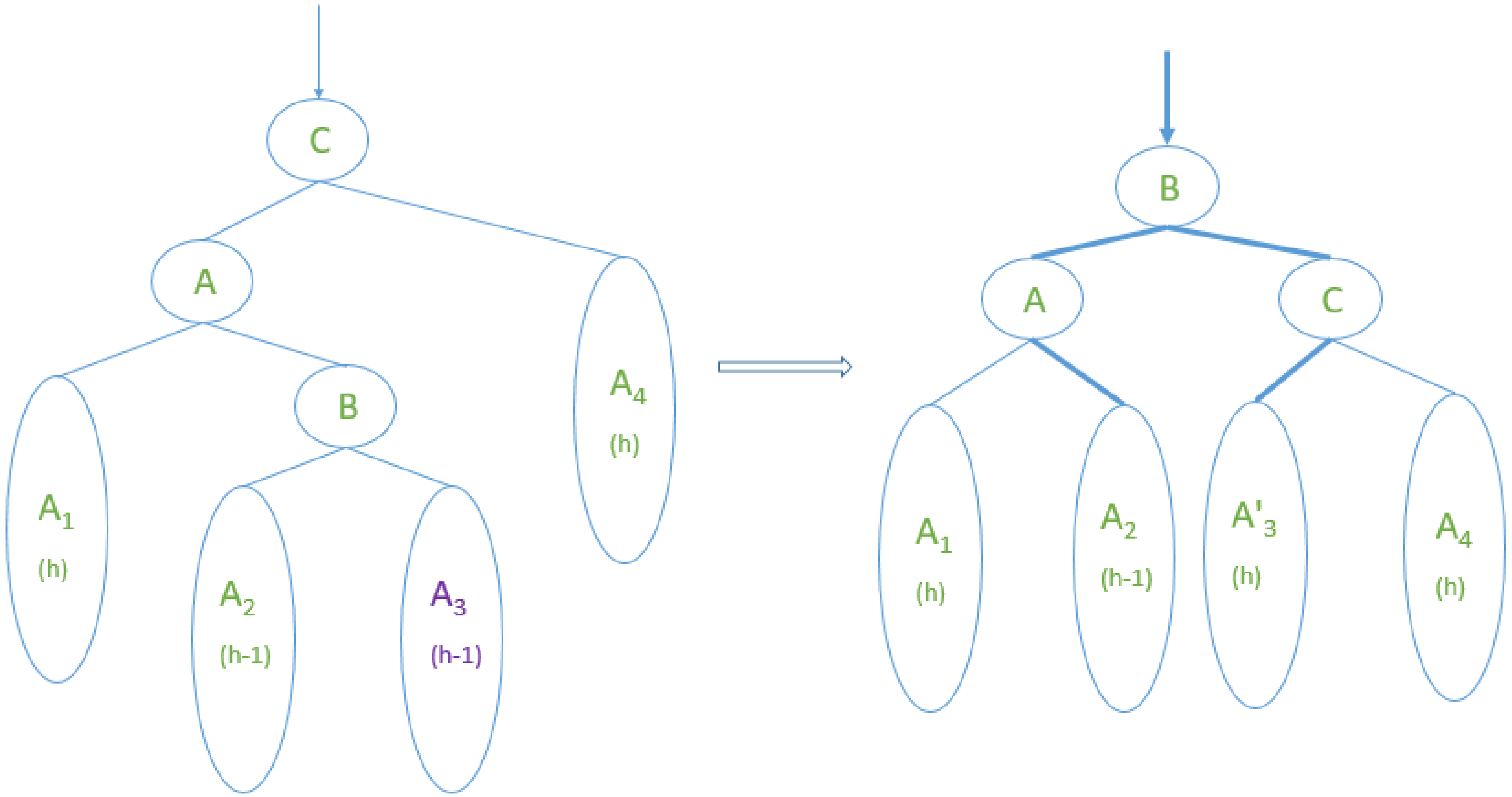
v4



v5



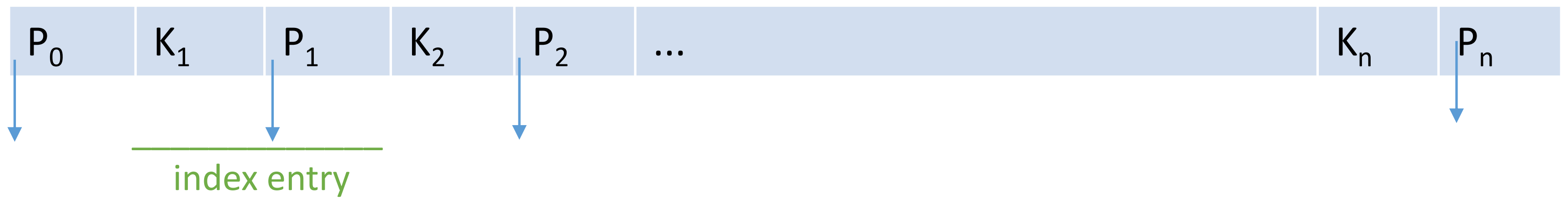
v6



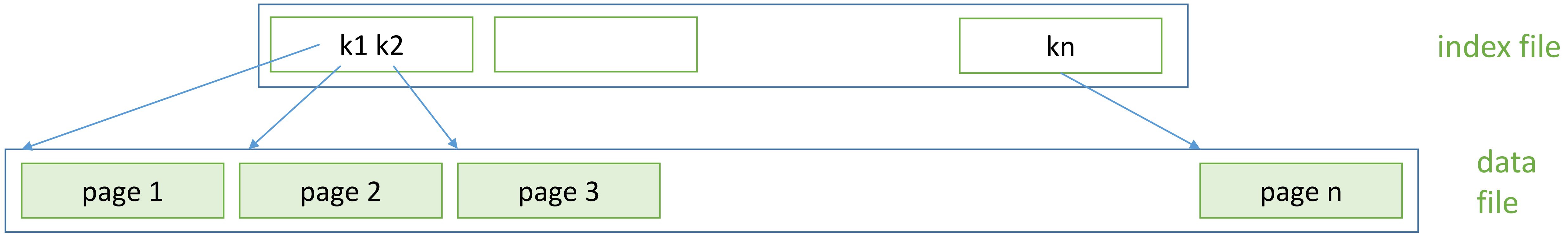
Indexed Sequential Access Method (ISAM)

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

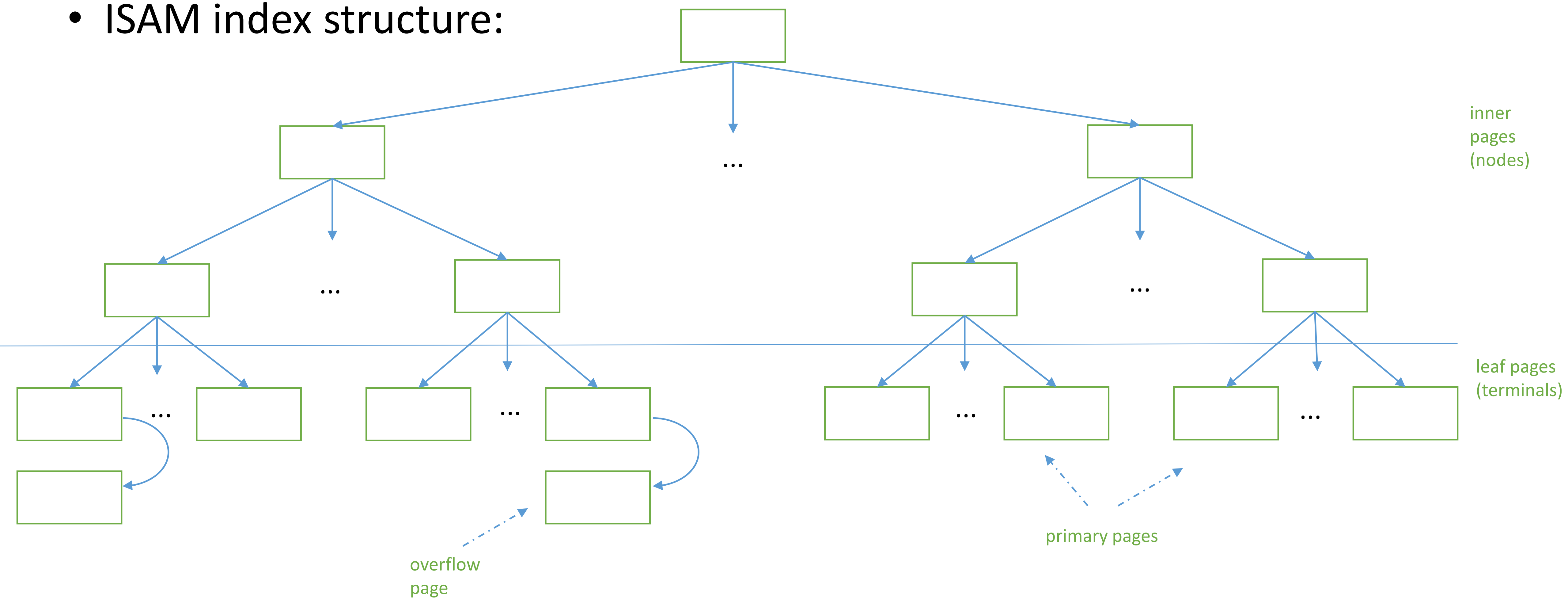
- data stored in sorted file - identify 1st phone using binary search; scan file to get the rest of the phones
- large file => potentially expensive binary search
- create another file - one record / page in the data file
- records of the form <1st key on the page, pointer to the page>, sorted on the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



- one-level index file:



- size of index file - much smaller than size of data file => faster binary search
- to further reduce the cost of binary search, index files are repeatedly created on top of previously created ones, until one such file fits on a single page
- ISAM index structure:



data pages
index pages
overflow pages

- file creation
  - allocate leaf pages - sequentially allocated, sorted on the key
  - allocate inner 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
    - $\log_f n$, where n – number of primary leaf pages, f – number of children / index page (*fan-out*)
  - binary search cost: $\log_2 n$
  - one-level index cost: $\log_2(n/f)$

References

- [Ra00] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems (2nd Edition), McGraw-Hill, 2000
- [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
- [Si10] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts, McGraw-Hill, 2010
- [Kn76] KNUTH, D.E., Tratat de programare a calculatoarelor. Sortare și căutare. Ed. Tehnică, București, 1976
- [Ga08] GARCIA-MOLINA, H., ULLMAN, J., WIDOM, J., Database Systems: The Complete Book, Prentice Hall Press, 2008