Data Management

Maurizio Lenzerini

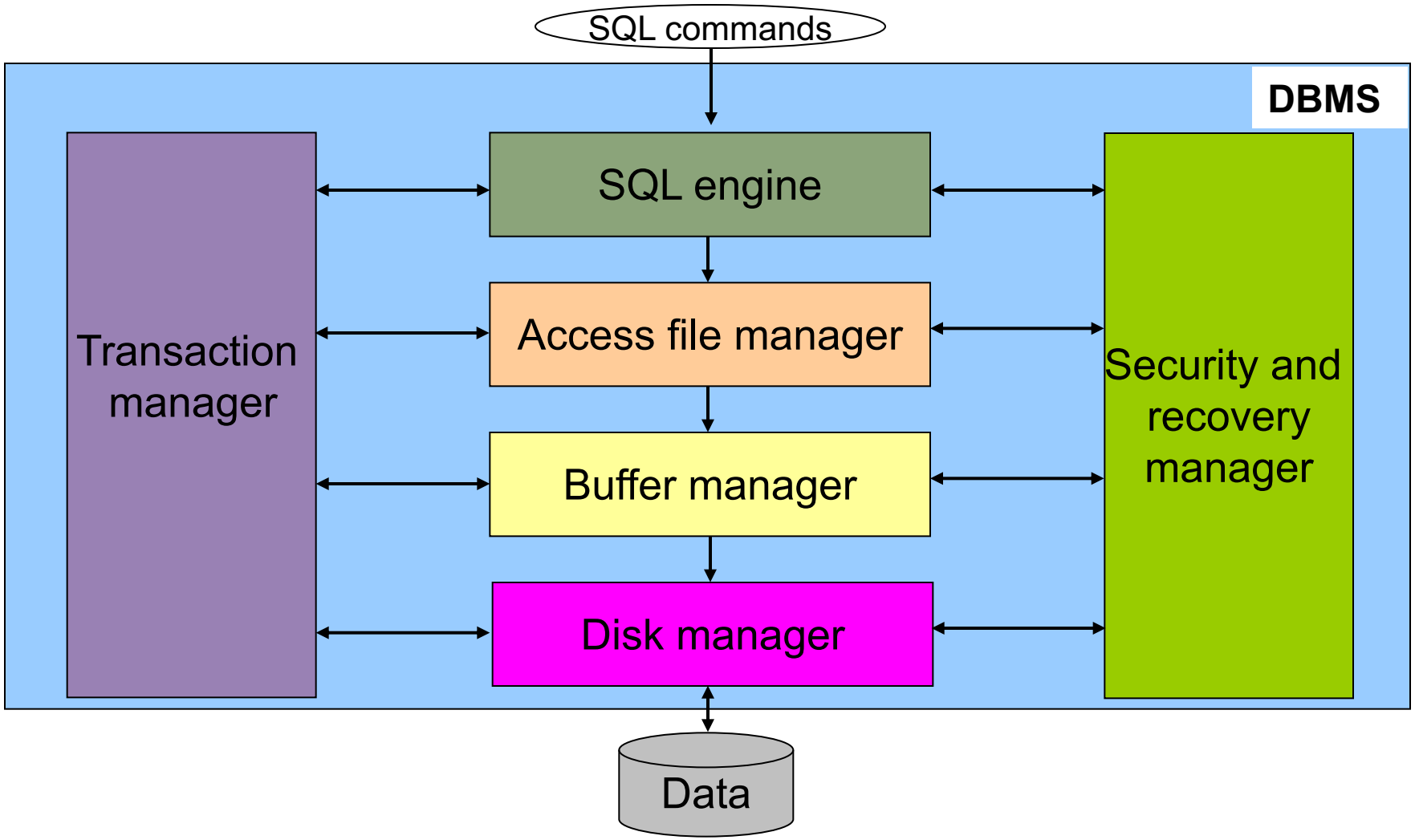
Dipartimento di Informatica e Sistemistica "Antonio Ruberti"
Università di Roma "La Sapienza"

Academic Year 2024/2025

Part 6
Access file manager

http://www.dis.uniroma1.it/~lenzerin/index.html/?q=node/53

Architecture of a DBMS



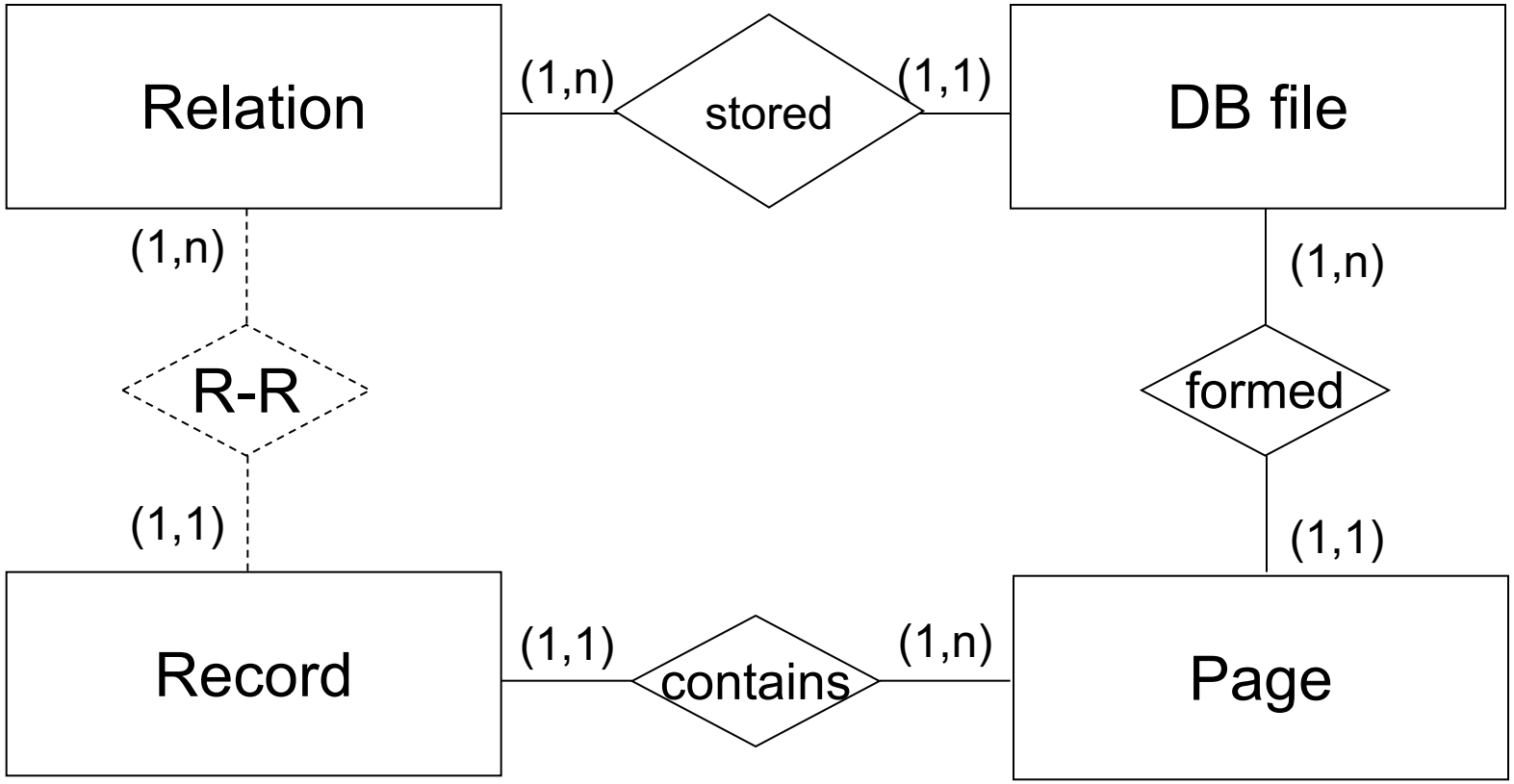
6. Access file manager

6.1 Pages and records
6.2 Simple file organizations
6.3 Index organizations

6. Access file manager

6.1 Pages and records
6.2 Simple file organizations
6.3 Index organizations

Relations, files, pages and records



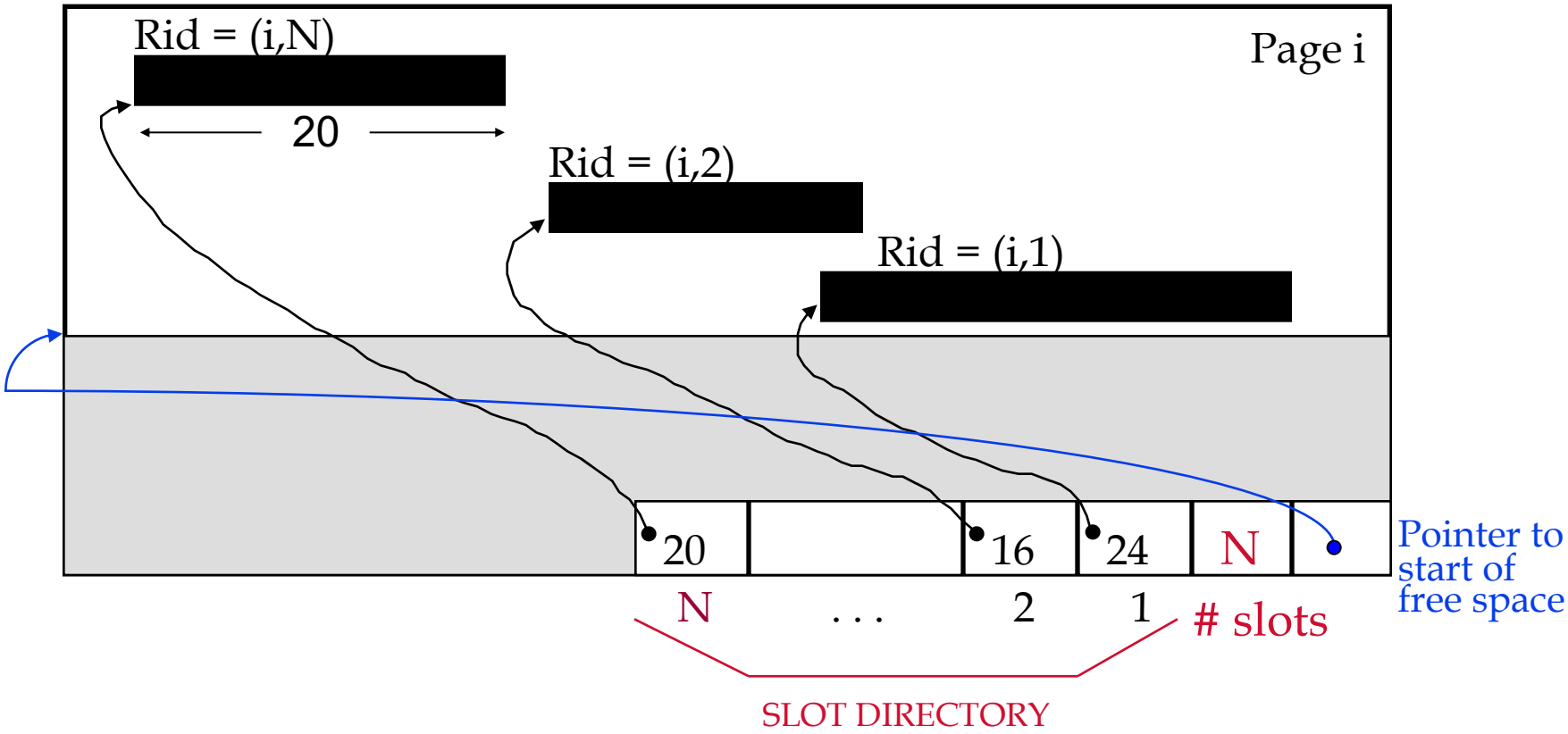
Nota: R-R is a derived relationship

Pages and records

- The usual dimension of a page is that of a block (unit of transfer from and to the buffer pool)
- A page has an address (page id) and is physically constituted by a set of slots, each one with an address; additionally, the page may contain a header (e.g., with pointers to other pages)
- A slot is a memory space that may contain one record (typically, but not always, all slots of a page contain records of one relation) and has a number (slot number or slot id) that identifies it in the context of the page (offset inside the page)
- Each record has therefore an identifier (record id, o rid)

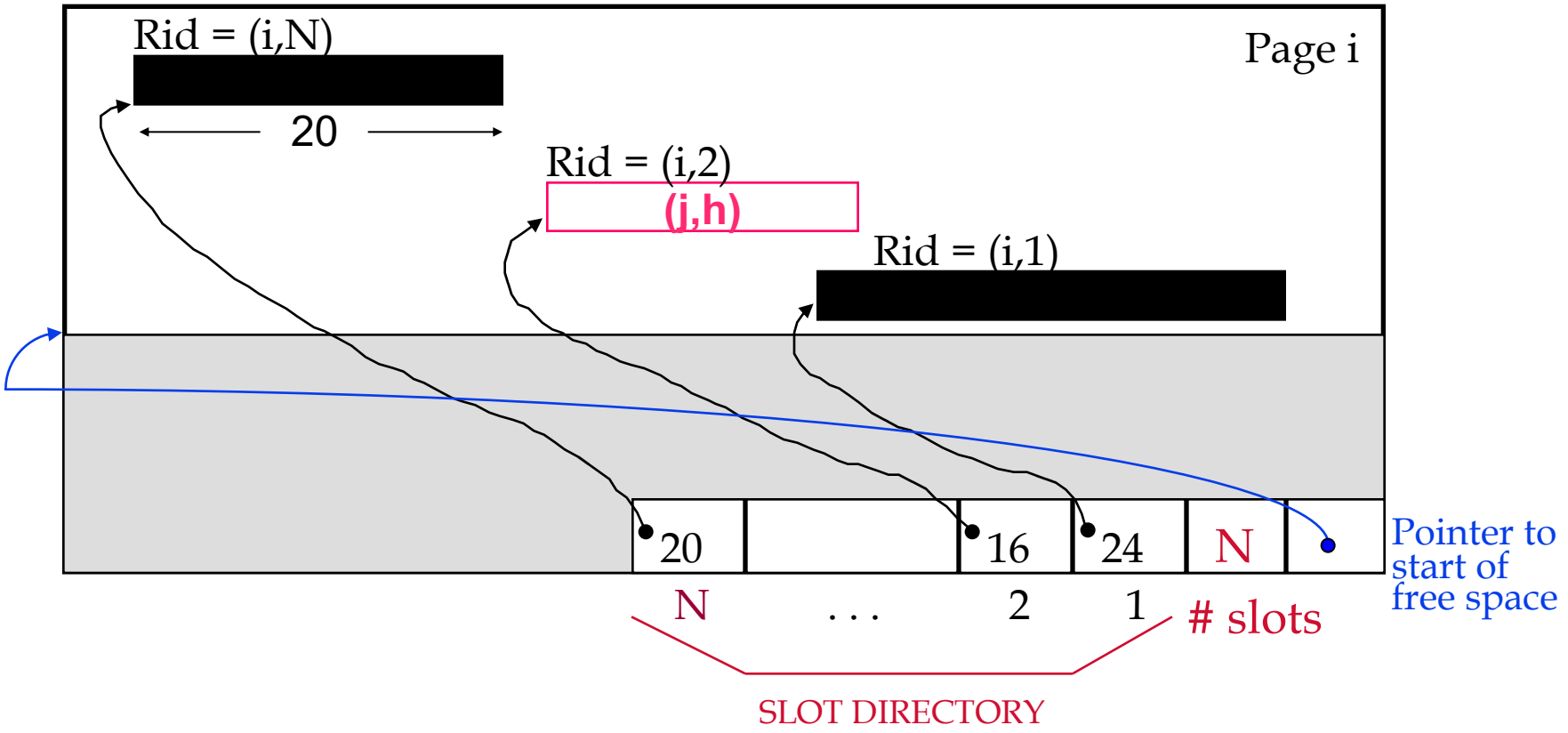
  rid = <page id, slot number>
- Usually, a record has also a header, for example including the pointer to the definition of the record schema, the length of the record, timestamps indicating the time the record was last modified, or last read, etc.

Page with variable length records



No problem in moving records within the same page! Deleting a record means to set to -1 the value of the corresponding slot. The data area and the free space area are re-organizing regularly.

Page with variable length records



When a record (for example the one with Rid (i,2) in the picture) moves to another page (page j), we can store in the old position the address of the new position (in terms of the page id j and the position h within page j).

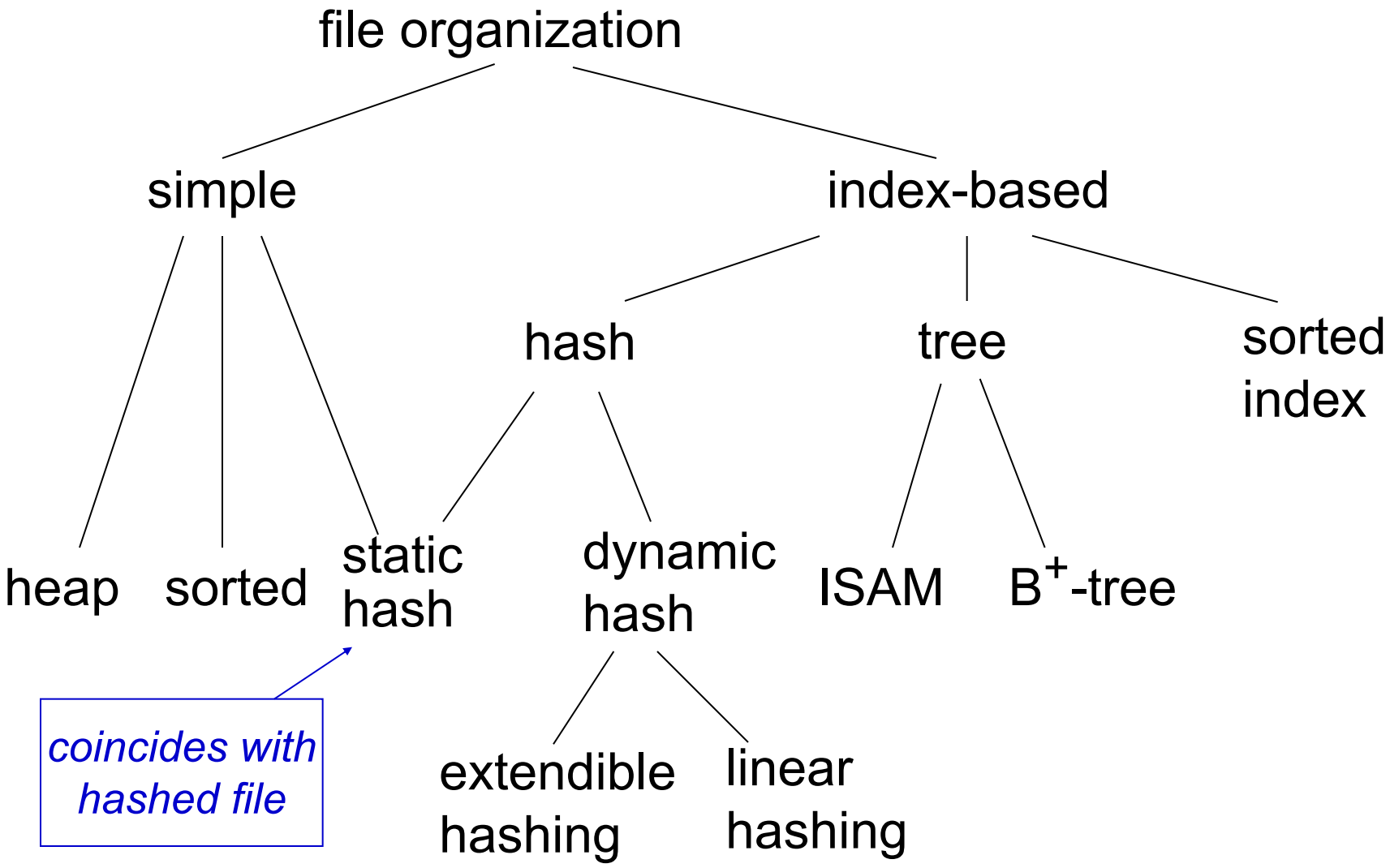
6. Access file manager

6.1 Pages and records
6.2 Simple file organizations
6.3 Index organizations

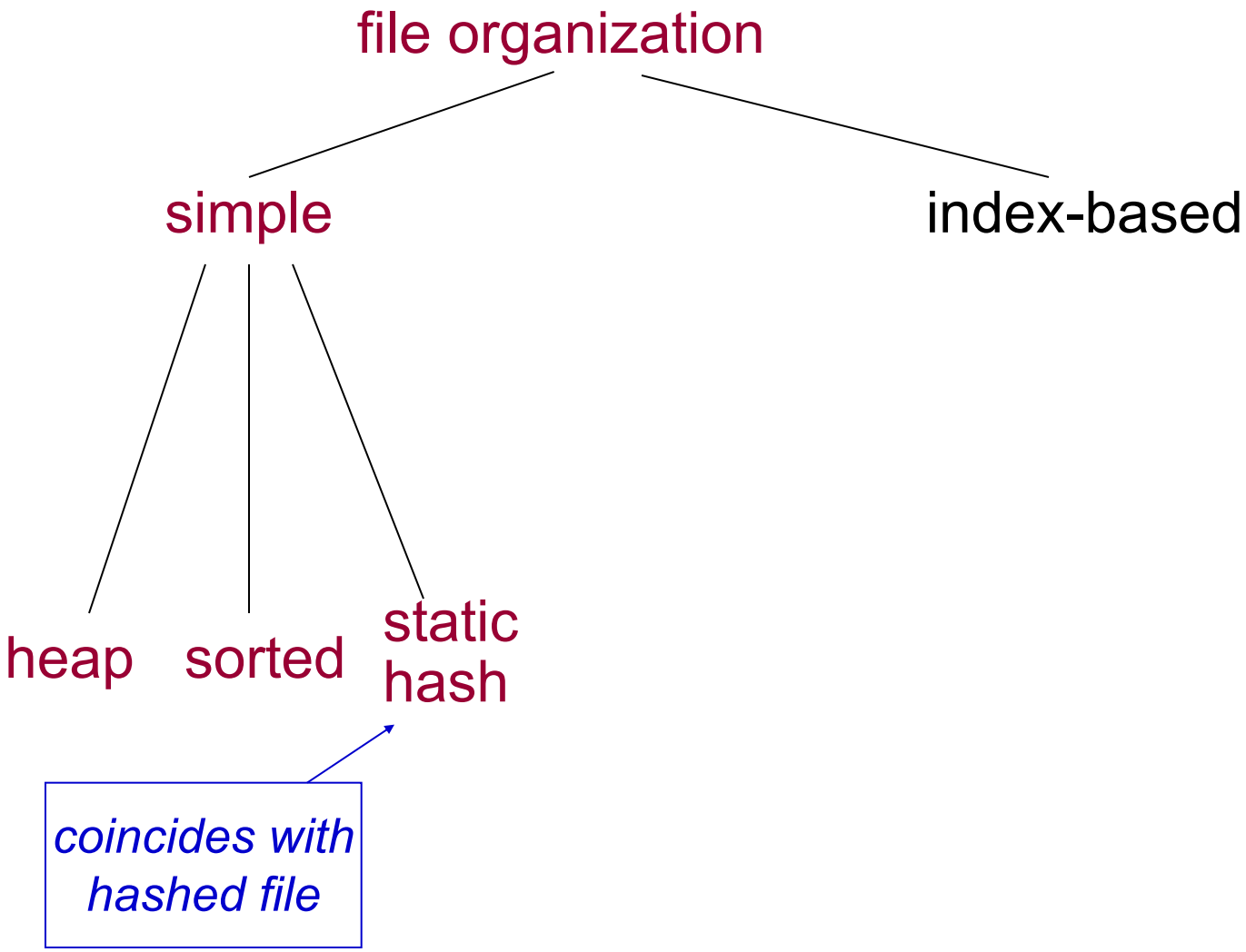
File

- A file is a collection of pages, each one containing a collection of records (as we saw before).
- The usual case is the one where all such records belong to the same relation, i.e., one file is used for one relation, but there are cases where a file is used for more than one relation
- A file organization should support the following operations:
  - insert/delete/update a record in one page of the collection
  - read the record specified by its rid
  - scan all the records in all its pages, possibly focusing on the records satisfying some given condition

File organizations



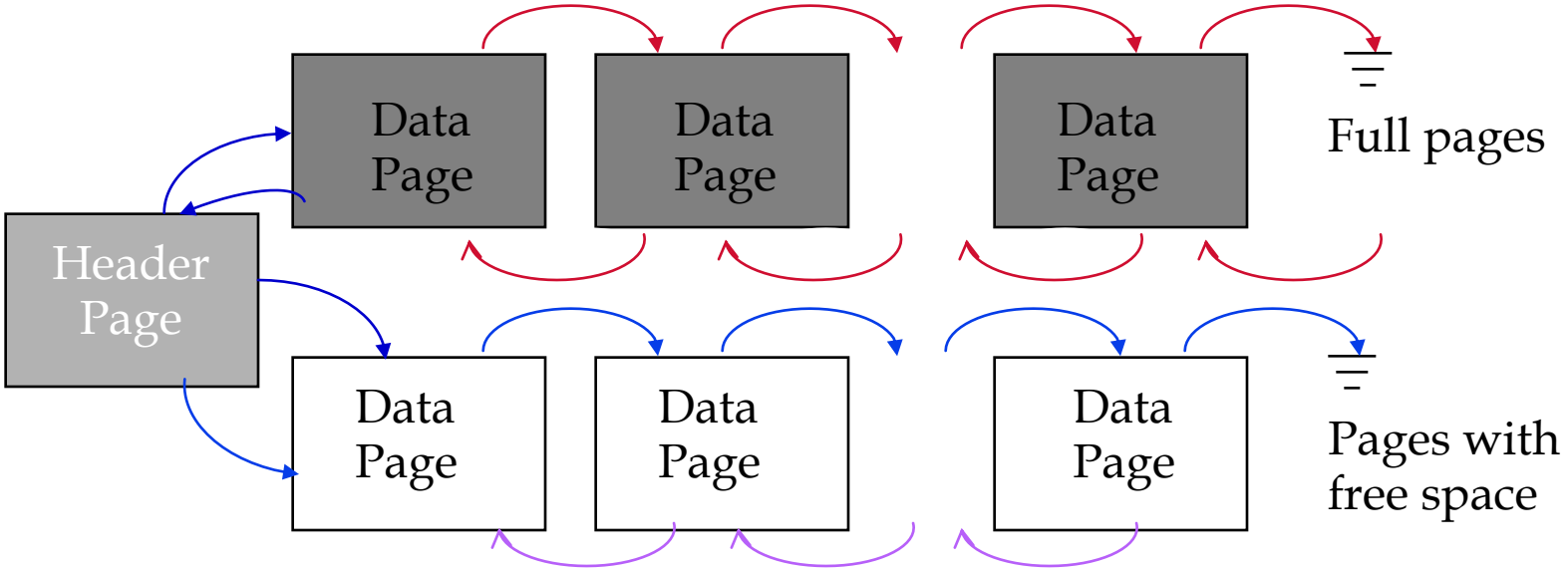
Simple file organizations



Heap File

- In the “heap file organization”, the file representing the relation contains a set of pages, each one with a set of records, with no special criterion or order for both the pages and the records
- When the relation grows or shrinks, pages are allocated or de-allocated
- To support the operations, it is necessary:
  - To keep track of the pages belonging to the file
  - To keep track of the record in the pages of the file
  - To keep track of free space in the pages of the file

Heap File represented through lists



- When a new page is needed, the request is issued to the disk manager, and the page returned by the disk manager is put as a first page of the list of pages with free space
- When a page is not used anymore (i.e., it is constituted by only free space), it is deleted from the list

File with sorted pages

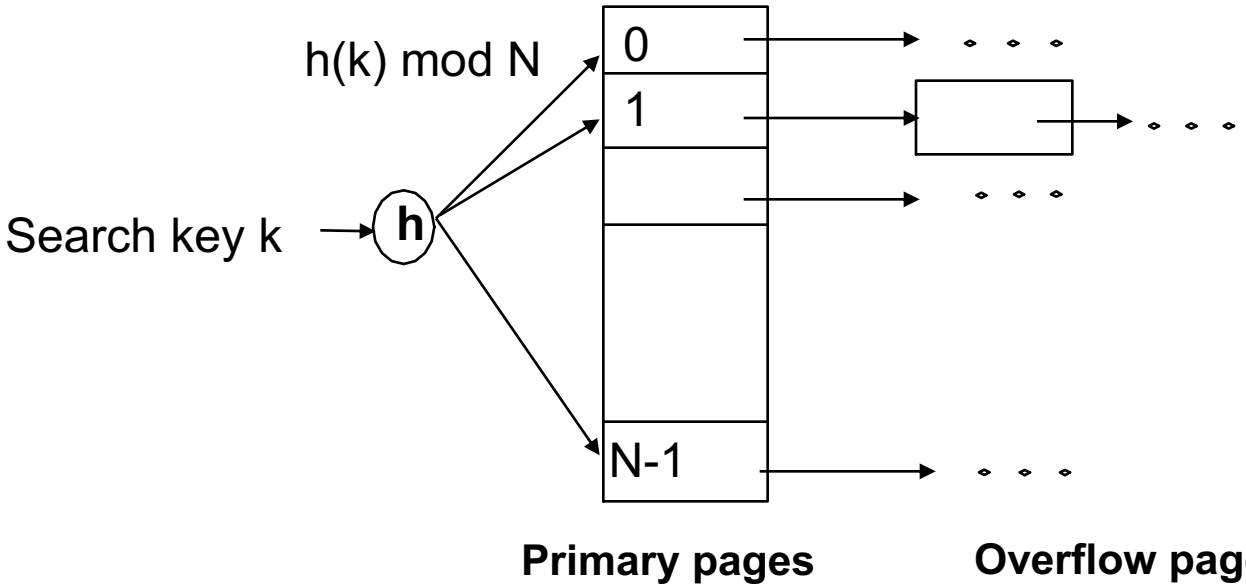
- Records are sorted within each page on a set of fields (such set forms the so-called “search key”)
- Pages are sorted according to the sorting of their records
- The pages are stored contiguously in a sequential structure, where the order of pages reflects the sorting of records
- Sorting is useful for several tasks, as we will see later (order-by in queries, more efficient search, joins,…)

Hashed file

- The pages of the relation are organized into groups, where each group is called a *bucket*
- A bucket consists of:
  - one page, called *primary page*
  - possibly other pages (called *overflow pages*) linked to the primary page
- A set of fields of the relation is chosen as the “search key”. When searching for a record R with a given value k for the search key, we can compute the address of the bucket containing R by means of the application of a function (called *hash function*) to k

Hashed file

- Fixed number of primary pages N (i.e., N = number of buckets)
  - sequentially allocated
  - never de-allocated
  - with overflow pages, if needed
- **h**(*k*) mod N = address of the bucket containing record with search key *k*
- (**h** ○ mod N) should distribute the values uniformly into the range 0..N-1



Cost model (wrt execution time)

B: number of pages in the file

R: number of records per page

D: time for writing/reading a page to/from secondary storage

- Tipically: 15 ms

C: average time for processing one record (e.g., comparing a field with a value)

- Tipically: 100 ns

→ I/O operations dominate main memory processing

→ Therefore, we concentrate only on the number of page accesses expressed with respect the number B of pages in order to characterize the execution time of operations

Operations on data and their cost

- Scan the records of the file
  - Cost of loading the pages of the file
  - CPU cost for locating the records in the pages
- Selection based on equality
  - Cost of loading the pages with relevant records
  - CPU cost for locating the records in the pages
  - The record is guaranteed to be unique if equality is on relation key (search based on relation key)
- Selection based on range of values
  - Cost of loading the pages with relevant records
  - CPU cost for locating the records in the pages

Operations on data

- Insertion of a record
  - Cost of locating the page where insertion must occur
  - Cost of loading the page
  - Cost of modifying the page
  - Cost of writing back the page
  - Cost of loading, modification and writing of other pages, if needed
- Deletion of a record
  - Similar to insertion

Operations on data

In what follows, we provide an estimation of the execution time of the basic operations for the three simple file organizations, trying to be as precise as possible.

We will concentrate only on the page access cost, i.e., we are interested in characterizing the cost of an operation for a certain file organization in terms of the number of page accesses required in the worst case by the execution of the operation, abstracting from the value of R, D and C. We will also characterize such cost using the big O notation (worst case asymptotic complexity)

Heap file - cost of operations

For simplicity, we will ignore the cost of locating and managing the pages with free space

- Scan:

$O(B)$

  - For each of the B pages of the file
    - load it
    - for each of the records of the page: process it

- Equality selection:

$O(B)$

→ the data record can be absent, or many data records can satisfy the condition

→ if the data record is just one (and therefore is unique) and present, and the probability that the record is in the i-th page is 1/B, then the average cost is B/2 (disregarding constant factors)

Heap file - cost of operations

- Range selection:

  $O(B)$

- Insertion: $O(1)$
  - Load the (last) page
  - Insert in the page (we assume there is space)
  - Write the page
- Deletion
  - If the record is identified by rid: $O(1)$
  - If the record is specified through an equality or range selection: $O(B + Y)$
    - Y number of pages with records to be deleted

Sorted file: search based on key

- Recall that the set of attributes on which the relation is sorted is called the "search key"
- The trivial method to perform the equality selection on the search key is by scanning the file. The average cost is B/2, both in the case of record present and in the case of record absent.
- In the case where the data are stored in contiguous pages with addresses in the range (a_1, a_B), a much more clever method to search K (where K is a value of the search key) is given by invoking the following (generic) algorithm with range (a_1, a_B).

Sorted file: search based on key

- To search the record with value K of the search key in the range of page addresses (h_1, h_2):
  1. if the range (h_1, h_2) is empty, then stop with failure
  2. choose a tentative page address *i* (h_1 <= *i* <= h_2) and load the page p_i at address *i*
  3. if the record with K is in the page p_i, then stop with success
  4. if K is less than the minimum key value in the page p_i, then repeat the algorithm using the range (h_1, i-1), else repeat the algorithm using the range (i+1, h_2)

- Clearly, the above is actually a generic algorithm, while a specific algorithm is obtained by selecting the criterion used in step 2 for choosing the address *i*. Two interesting cases are:
  - Binary search
  - Interpolation search

Sorted file: search based on key

- Binary search: the tentative address is the one at the half of the range

- Interpolation search: if the searck key values are numeric, and uniformly distributed in the range (K_{min}, K_{max}), and if K is the value to search, then, assuming that the distance between addresses is analogous to the distance between key values, we can choose as tentative address

$$i = a_1 + (K - K_{min}) / (K_{max} - K_{min}) \times (a_B - a_1)$$

where, as we said before, a_1 is the address of the first page, and a_B is the address of the last page.

Sorted file: other operations

- Range selection: a search for range (K_1, K_2) reduces to searching for K_1 and then scanning the subsequent pages to look for values that are less than or equal to K_2
- Insertion: either we move the records to maintain the order, or we use an overflow storage (and when insertions are too many, we rearrange the data)
- Deletion: search for the record, and then modify the page containing the record, and possibly other pages to keep the file compact and sorted

Sorted file - cost of operations

- Scan: $O(B)$

- Equality selection on search key (with binary search)

  $O(\log_2 B)$

  - binary search for locating the page with the relevant record
    - $1 + \log_2 B$ steps for locating the page
    - at each step, 1 I/O operation + 2 comparisons
  - binary search for locating the relevant record in the last page

- Equality selection on search key (with interpolation search)
  - in the average case $O(\log_2 \log_2 B)$
  - in the worst case: $O(B)$

Sorted file - cost of operations

- Range selection on search key (or equality search on non-key):
  - we analyze the case of binary search
  - if the range that we search is (K_1, K_2), and the keys in this range are uniformly distributed in the range (K_{min}, K_{max}), then

$$f_s = (K_2 - K_1) / (K_{max} - K_{min})$$

is the expected portion of pages occupied by records in the range, and the cost of the operation is:

$$O(\log_2 B + f_s \times B)$$

where $\log_2 B$ is the cost of locating the first record with the value K_1, and $(f_s \times B - 1)$ is the cost of searching for the other records.

Sorted file - cost of operations

- Insertion: $O(B)$
  - Worst case: first page, first position
  - Cost of searching the page to insert
  - Insertion of record
  - If we decide not to use overflow pages, the we must add the cost of loading and writing the other pages: 2B

- Deletion:
  - Similar to insertion (if we decide not to leave empty slots)
  - If the deletion condition is an equality selection of non-key fields, or a range selection, then the cost depends also on the number of records to be deleted

Hashed file - cost of operations (rough analysis)

- Scan:

  1.25 B → O(B)

  We assume (as usual in hashing) that pages are kept at about 80% occupancy, to minimize overflows as the file expands.
- Equality selection on search key:

  number depending on overflows

  We assume direct access through the hash function
- Range selection on search key: O(B)
- Insertion: O(1) + overflow cost
- Deletion: O(1) + overflow cost

Comparison

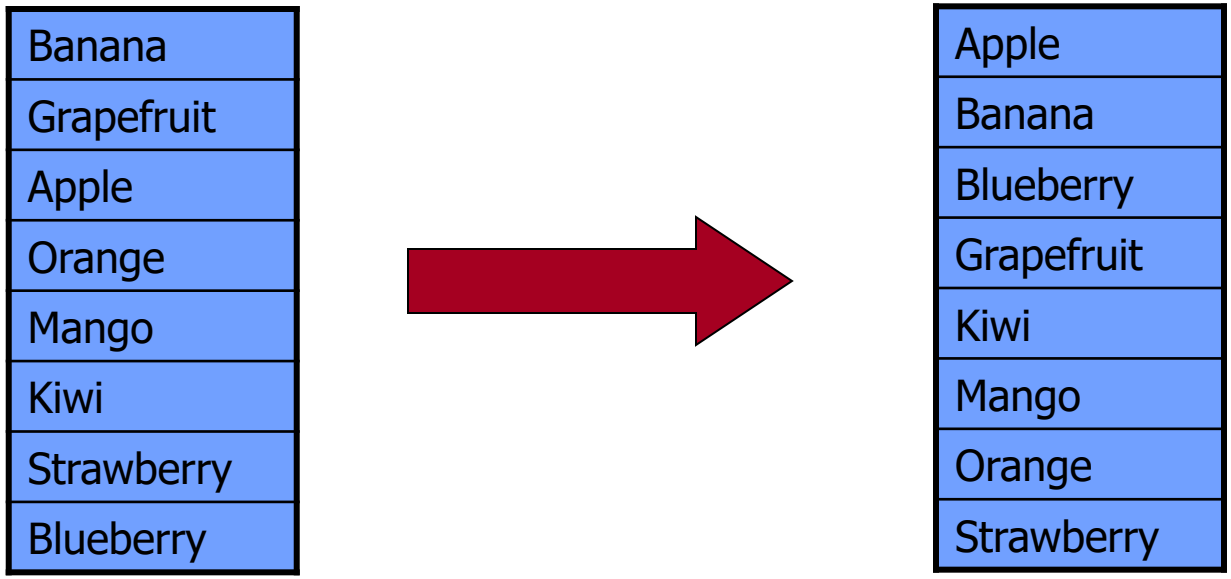
Organization	*Scan*	*Equality selection*	*Range selection*	*Insertion*	*Deletion*
Heap file	$O(B)$	$O(B)$	$O(B)$	$O(1)$	$O(B)$
Sorted file	$O(B)$	(search based on key) $O(\log_2 B)$	(on key) $O(\log_2 B)$ + number of relevant pages	$O(\log_2 B + B)$	$O(\log_2 B + B)$
Hashed file	$O(B)$	$O(1)$ + number of relevant pages	$O(B)$	$O(1)$	$O(1)$ + number of relevant pages

In the above table we have assumed the use of binary search for sorted file and no overflow for hash file.

Sorting is only useful for sorted files?

We have seen that the sorted file is a possible organization. But this is not the only reason to sort a file. For example:

- Users may want data sorted as result of queries
- Sorting is first step in bulk-loading a B+ tree (see later)
- Sorting is useful for eliminating duplicates
- Sort-merge algorithm for join and other operators involves sorting (see later)



Algorithms for sorting

- Don’t we know how to sort?
  - Quicksort
  - Mergesort
  - Heapsort
  - Selection sort
  - Insertion sort
  - Radix sort
  - Bubble sort
  - Etc.
- Why don’t we use these algorithms for databases?

Sorting in secondary storage

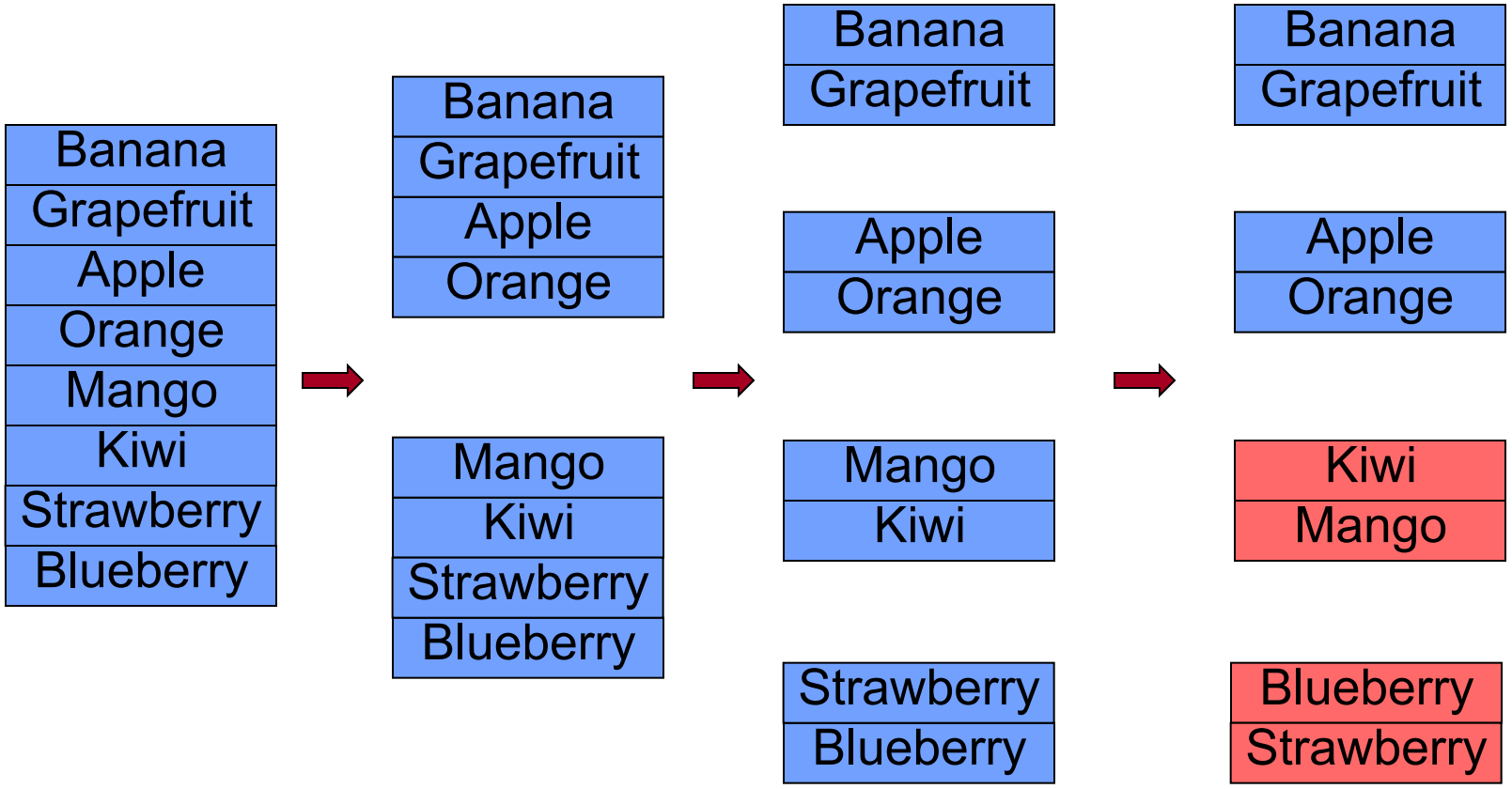
The problem is how to sort data that does not fit in main memory

Sorting of data in secondary storage (called external sorting) is very different from sorting an array in main memory (called internal sorting).

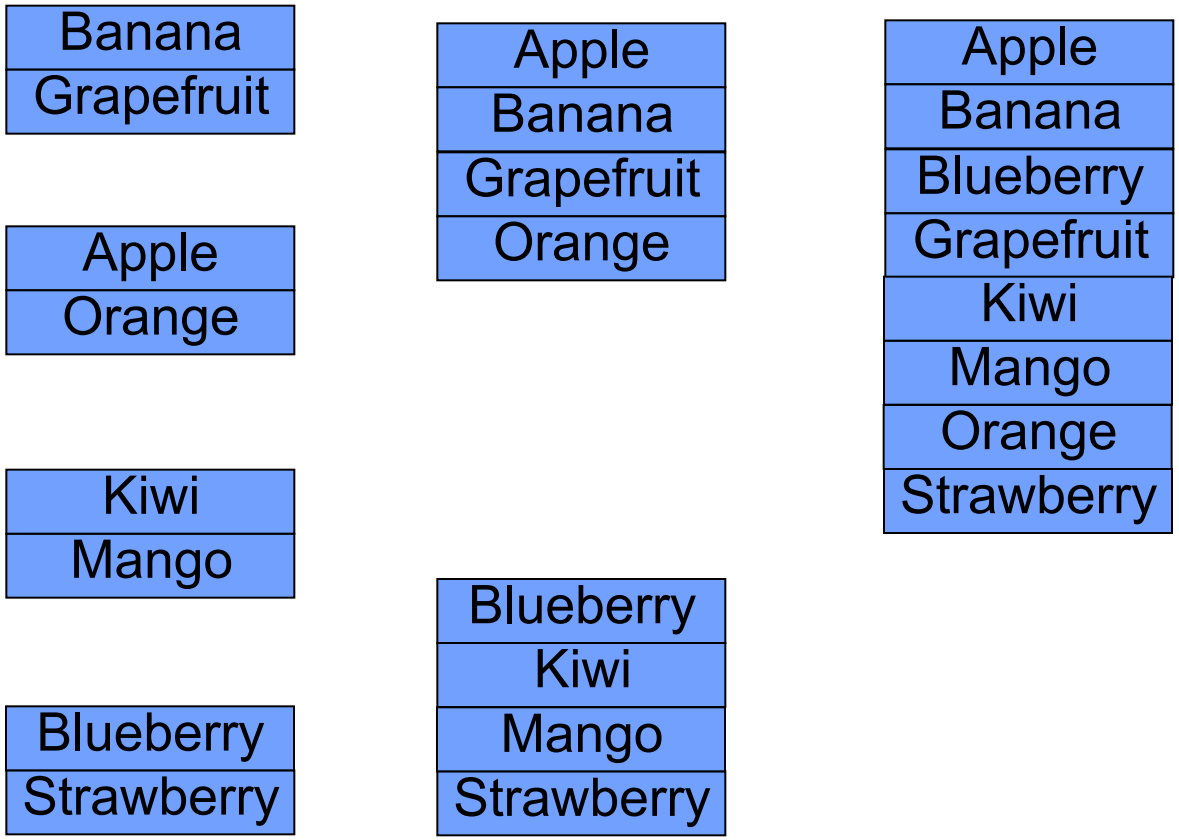
Let us recall for the moment how an internal sorting algorithm works, in particular, the merge-sort algorithm

Example: merge sort in main memory

Thanks to Brian Cooper (Yahoo! Research) for some of the slides/figures/animations



Example: merge sort in main memory

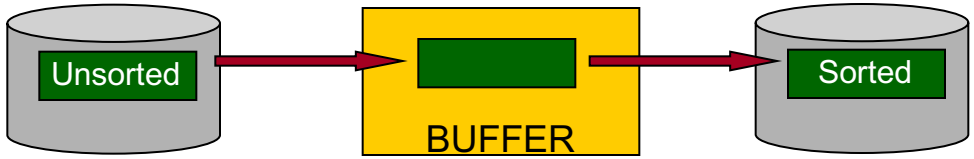


Isn't that good enough?

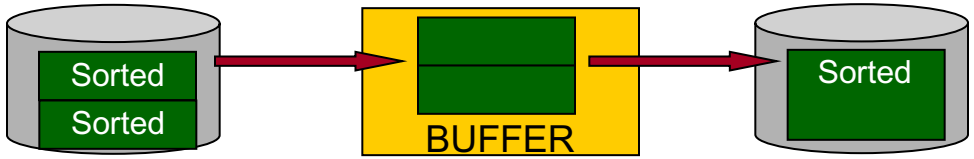
- Consider a file with N records
- Merge sort requires $O(N \log_2 N)$ comparisons
- We want to minimize disk I/Os
  - Don't want to go to disk $O(N \log_2 N)$ times!
- Key insight for external sorting (i.e., sorting data in secondary storage): sort based on pages, not records
  - Read whole pages into BUFFER, not individual records
  - Do some in-memory processing
  - Write processed blocks out to disk
  - Repeat

External sorting with 2-way sort

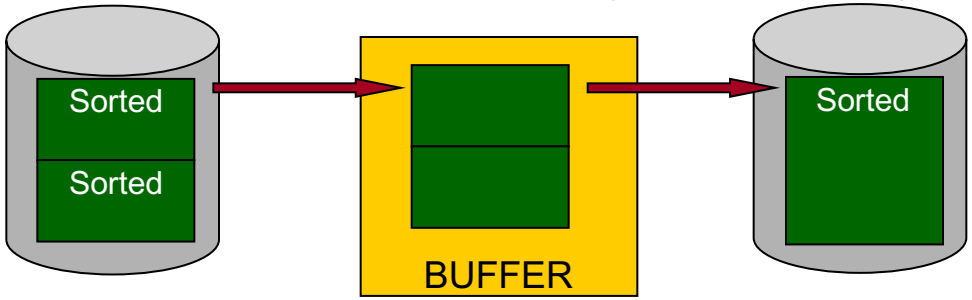
- Pass 0: sort each page



- Pass 1: for each pair of pages, merge such 2 pages into one run



- Pass 2: for each pair of runs, merge such runs (each of 2 pages) into one run



- Pass 3: for each pair of runs, merge such runs (each of 4 pages) into one run
- …
- Sorted!

What did that cost us?

- B pages in the file
- Each pass reads and writes B pages
- How many passes?
  - Pass 0
  - Pass 1: from B pages to B/2 runs
  - Pass 2: from B/2 runs to B/4 runs
  - …
- After the various passes 1,2,…k, the number of runs is:

  $$B/2^1,\ B/2^2, B/2^3, ..., B/2^k$$

  and the algorithm stops when $1 >= B/2^k$ (i.e., $k >= \log_2 B$)
- So the total number of passes is $(\lceil \log_2 B + 1 \rceil)$ and the total cost is $2 \times B \times$ number of passes =

  $$2 \times B \times (\lceil \log_2 B + 1 \rceil)$$

What did that cost us?

- Why is this better than plain old merge sort?
  - N >> B
  - So $O(N \log_2 N) >> O(B \log_2 B)$
- Example:
  - 1,000,000 record file
    - 8 KB pages
    - 100 byte records
    - = 80 records per page
    - = 12.500 pages
  - Plain merge sort: 41.863.137 disk I/O's
  - 2-way external merge sort: 365.241 disk I/O's
  - 4.8 days versus 1 hour

Can we do external sorting more efficiently?

- 2-way merge sort only uses 3 frames in the buffer
  - Two frames to hold input records
  - One frame to hold output records
    - When that frame fills up, flush to disk
- Usually, we have a lot more buffer frames than that
  - Set aside 100 MB for sort scratch space = 12.800 buffer pages
- Idea: each pass, read as much data as possible into buffer
  - Thus, reducing the number of passes
  - Recall that total cost is: 2 × B × #passes

External k-way or multipass merge sort: basic idea

- Assume we have F frames in the buffer
- Pass 0: Read the file in runs of F pages, and, at each run, internally sort the records in such F pages and write a file (called run) to disk
- In all subsequent passes, we reserve F-1 input frames for input, and 1 frame for output
- Pass 1: Merge F-1 runs into one output run
  - For each input run, read one page in one dedicated frame
  - When a page is used up, read next page of input run into the same frame
  - The result of merge is buffered in the output frame: when such frame is full, its content is written in the output run in secondary storage
- Pass 2: Merge F-1 runs into one output run
  - For each run, read one page in one dedicated frame
  - When a page is used up, read next page of input run into the same frame
  - The result of merge is written in the output frame: when such frame is full, its content is written in the output run in secondary storage
- …
- When we have one run, the file is sorted!

The multipass merge sort algorithm

Suppose we have F frames free in the buffer that we can use for sorting relation R

1. (**Pass 0**) repeat until no more pages in R

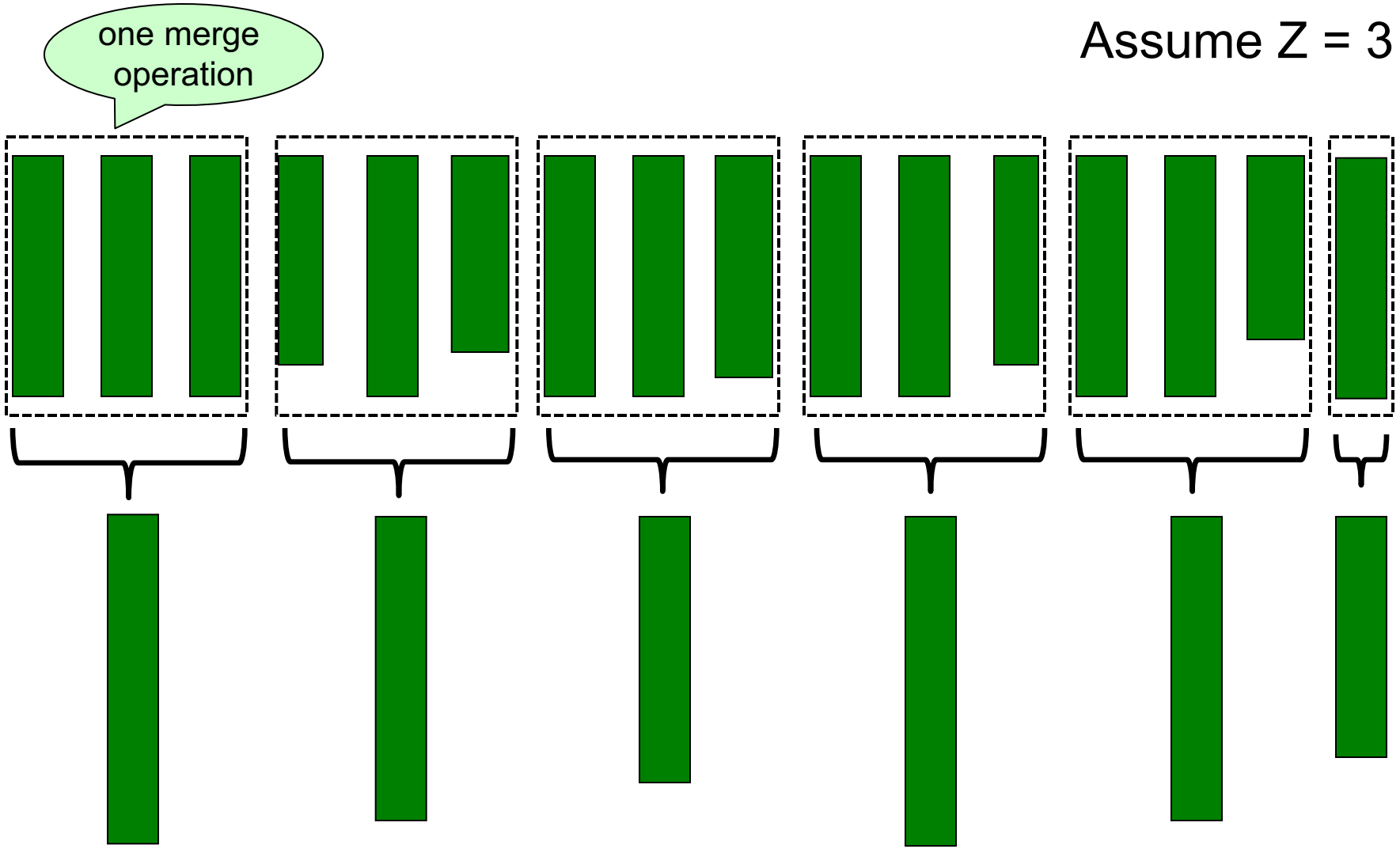
   bring F pages of relation R in the buffer, sort their records, and write the corresponding pages in a file (called run, or sorted sublist)

2. set i = 1 and repeat until we have just one run

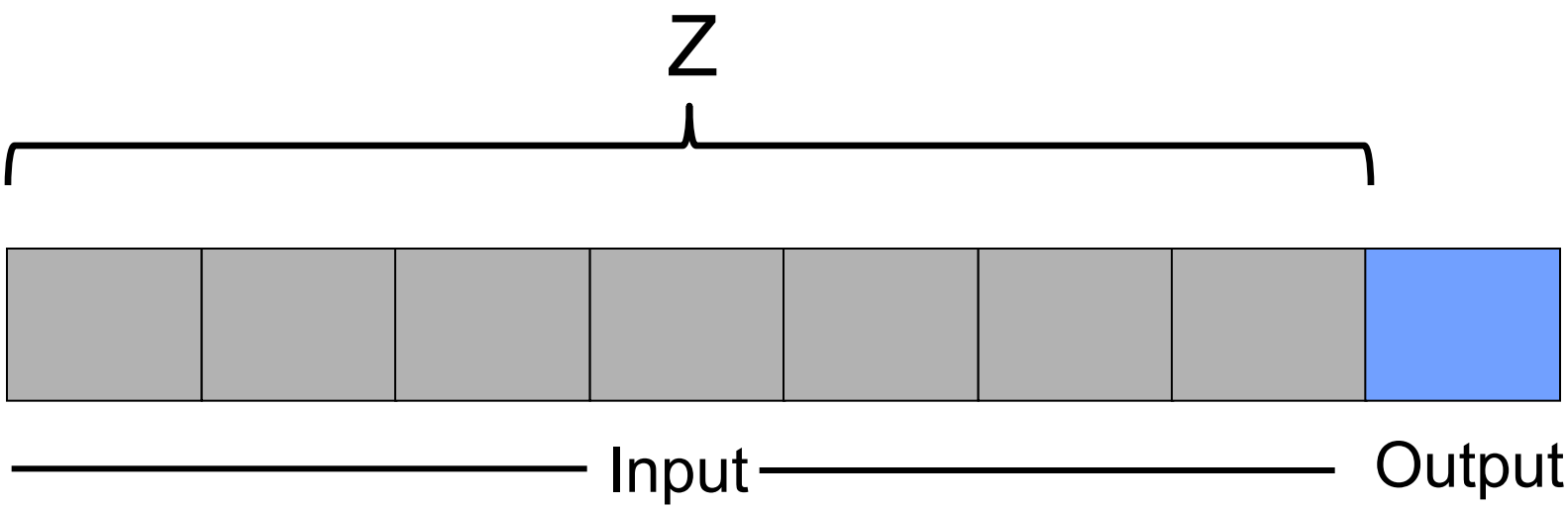
   (**Pass i**) repeat until we do not have runs to consider

   (**merge**) merge the records of the first Z < F runs not yet analyzed into a single new run: this is done by reading the pages of the Z runs, using one frame for each run (when frame j is used up, we load in the frame the next page of the j-th run), and writing the result in the output file (new run), one page at a time using one output frame in the buffer; add 1 to i

Example of pass i: repeat the merging of Z runs until needed

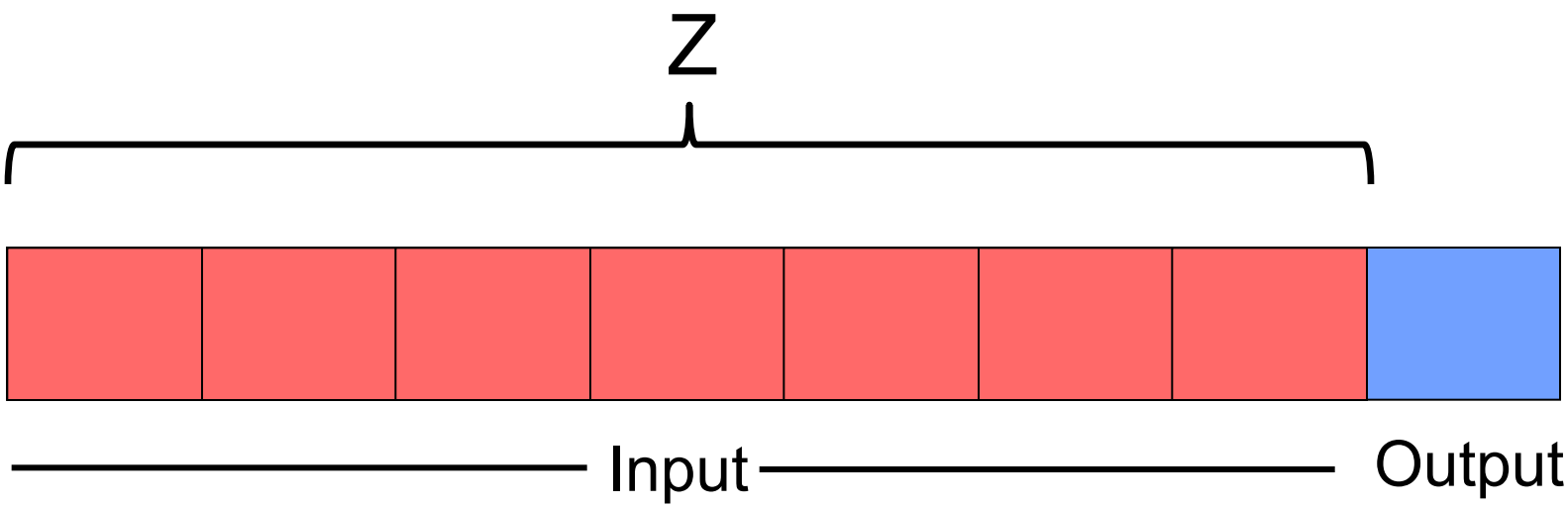


Example of one merging operation

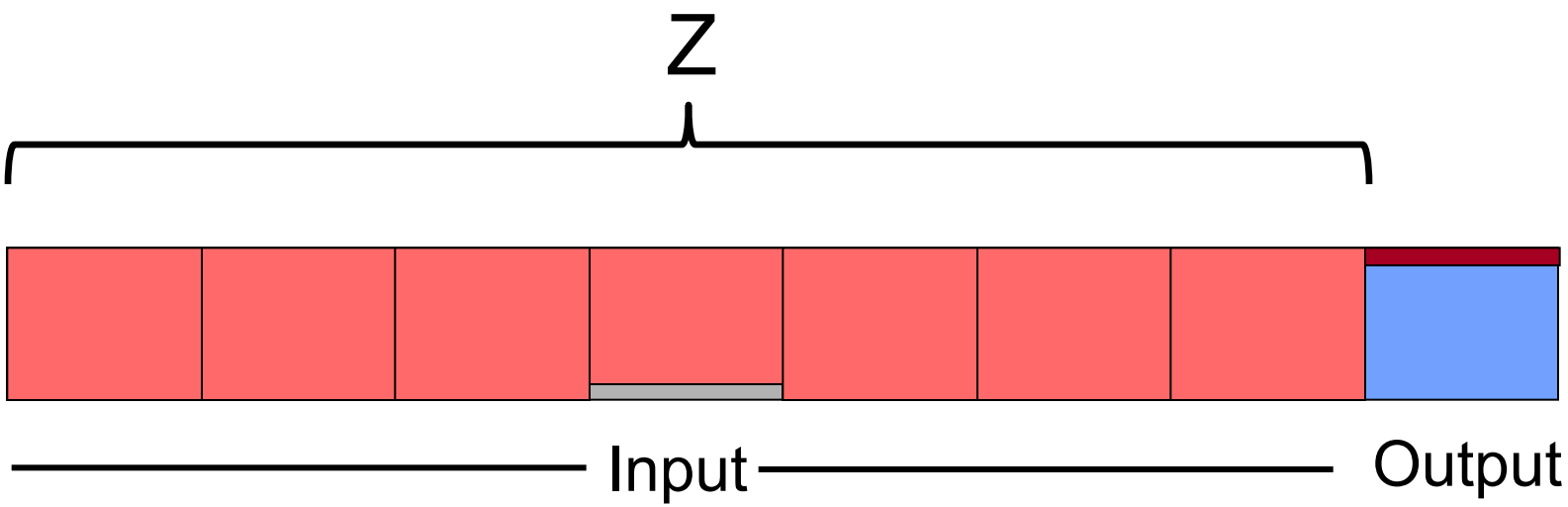


Imagine the Z sorted files to be merged at the bottom: when you see a page coming from the bottom, it comes from one of such files. When you see a page leaving the last cell on the right (the blue one), it is written in the output file.

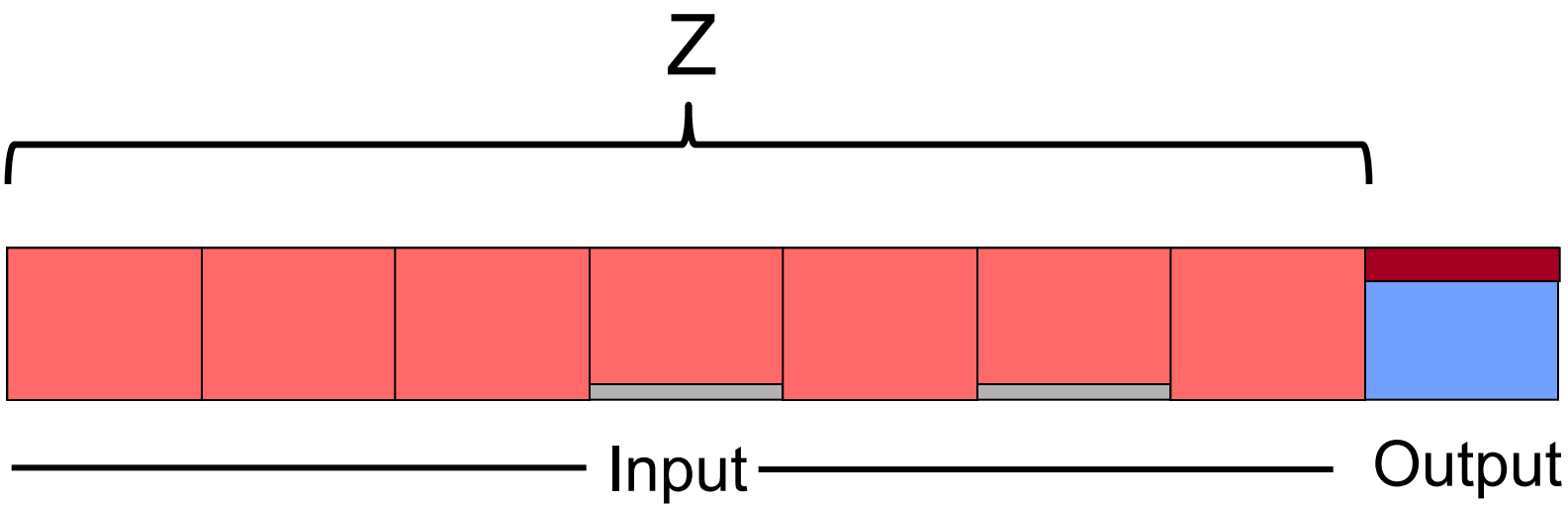
Example of merge



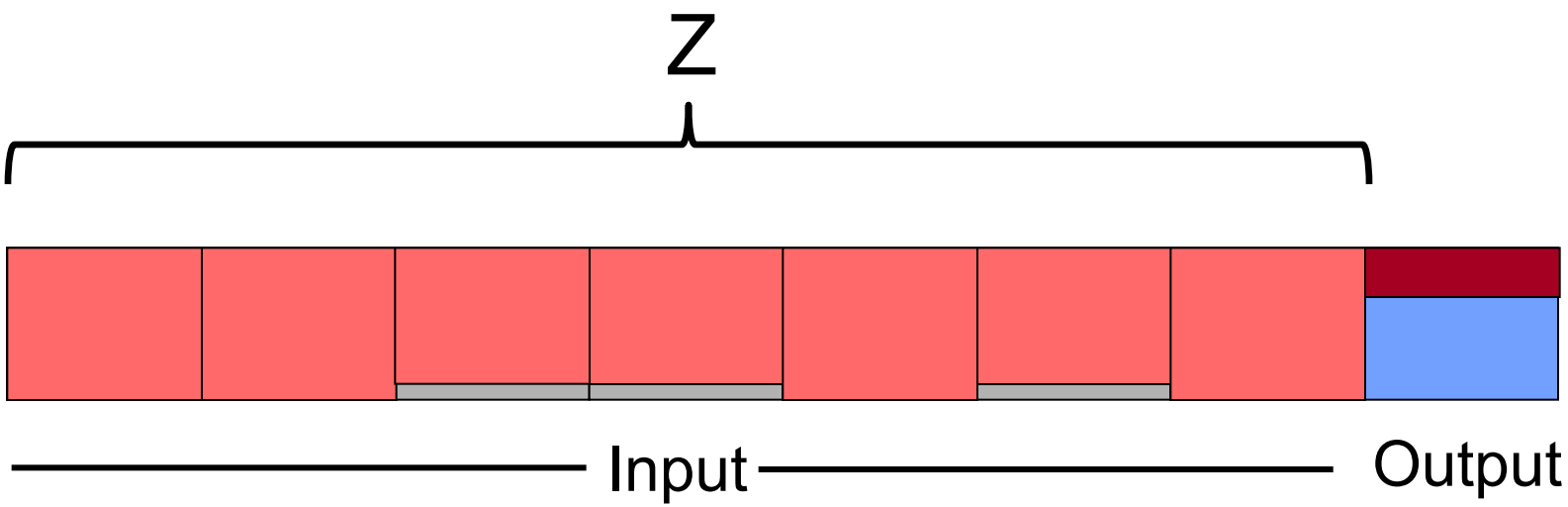
Example of merge



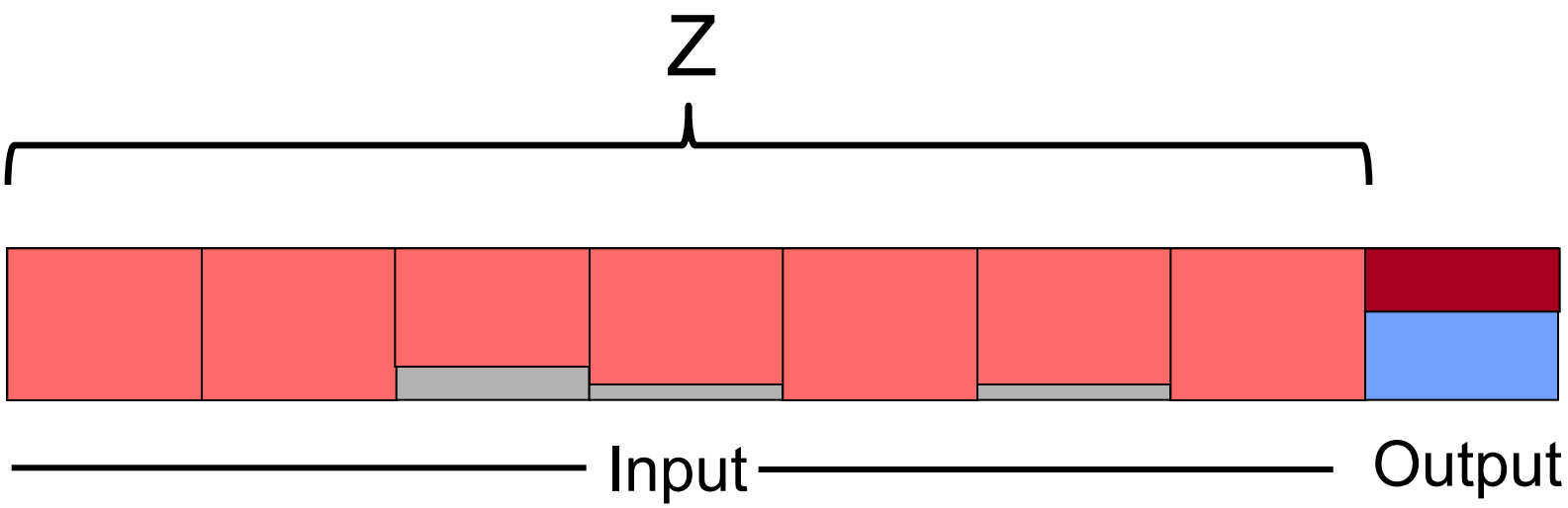
Example of merge



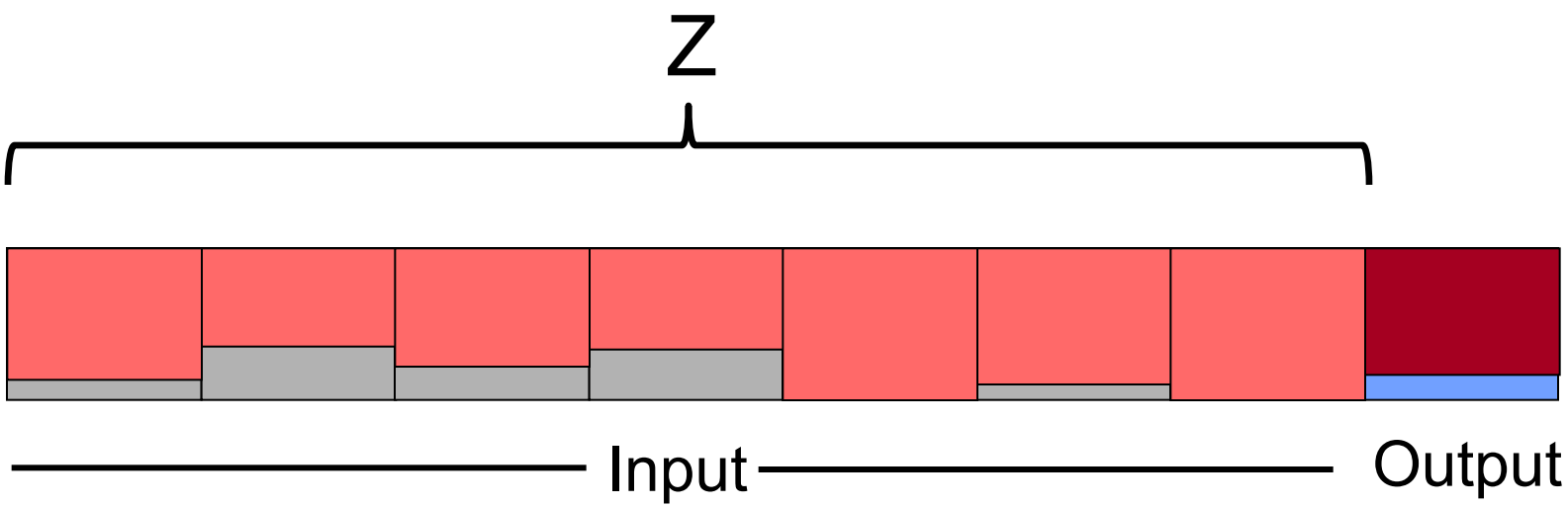
Example of merge



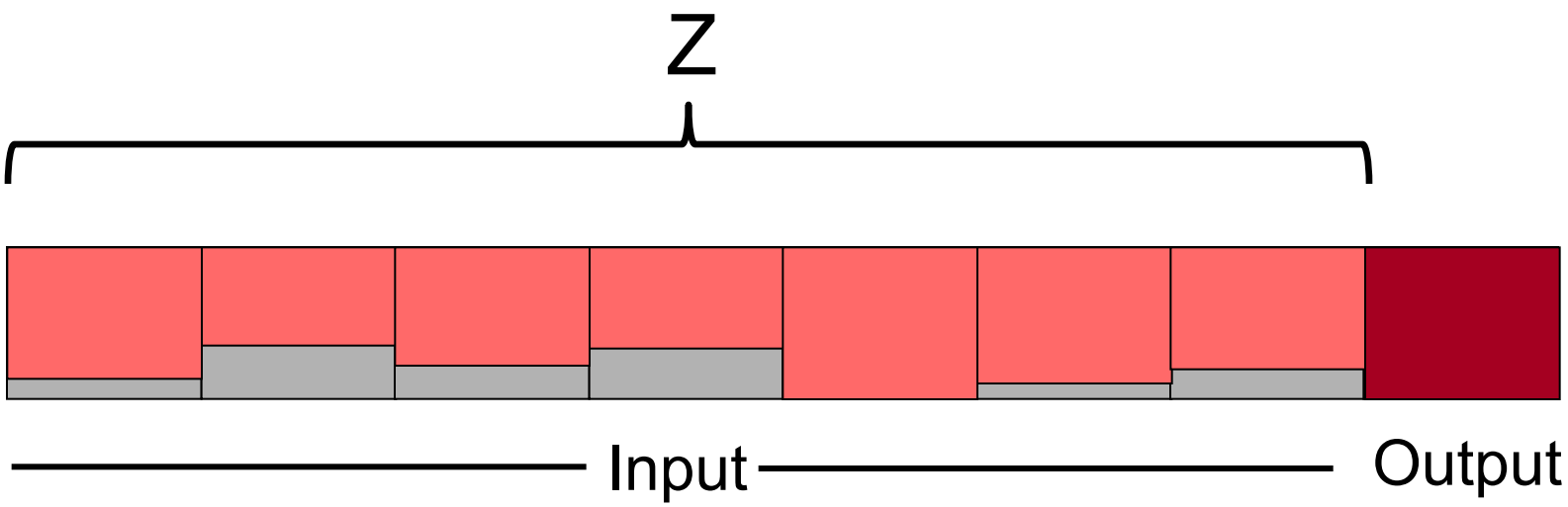
Example of merge



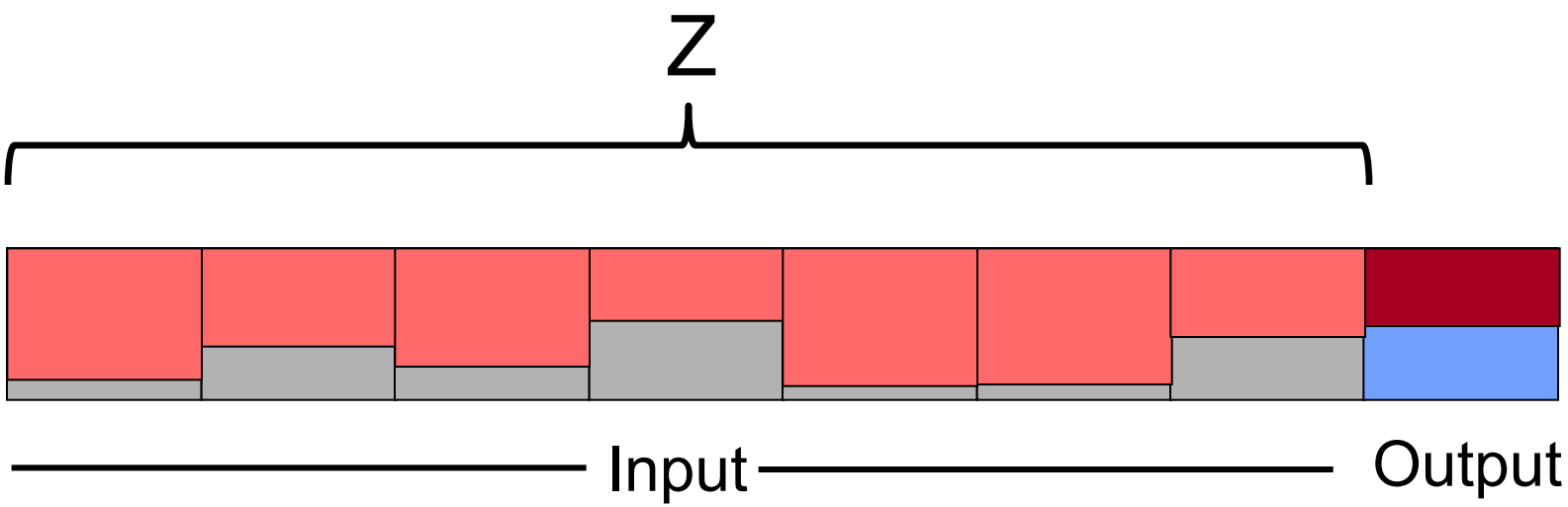
Example of merge



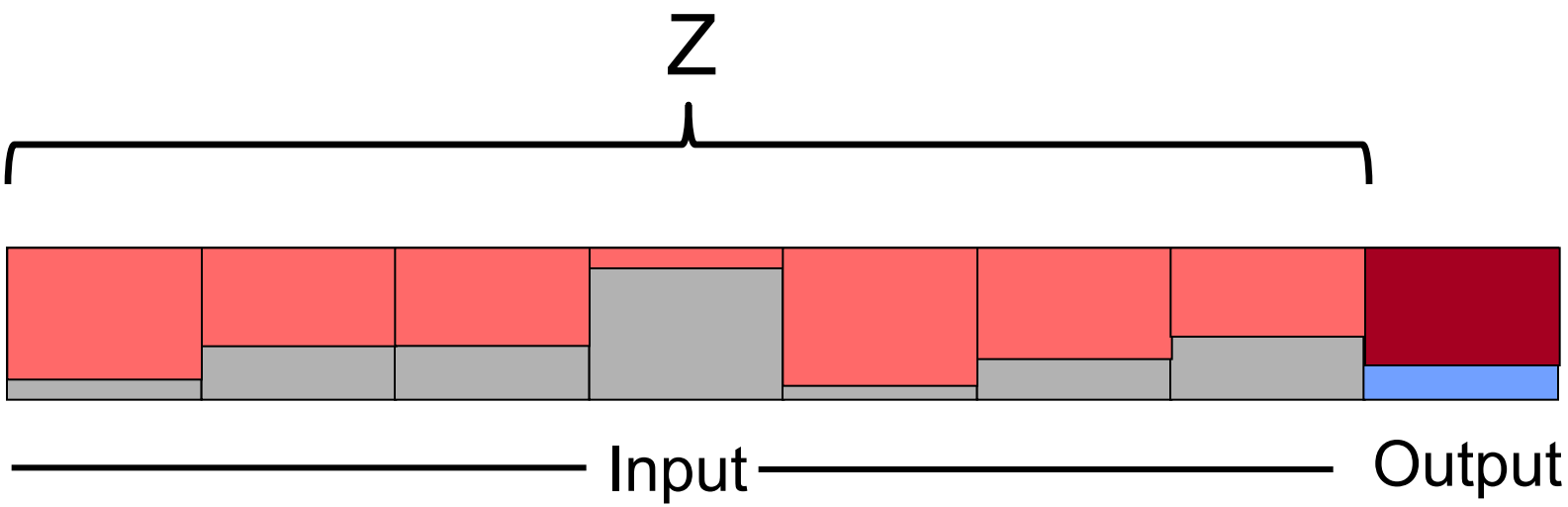
Example of merge



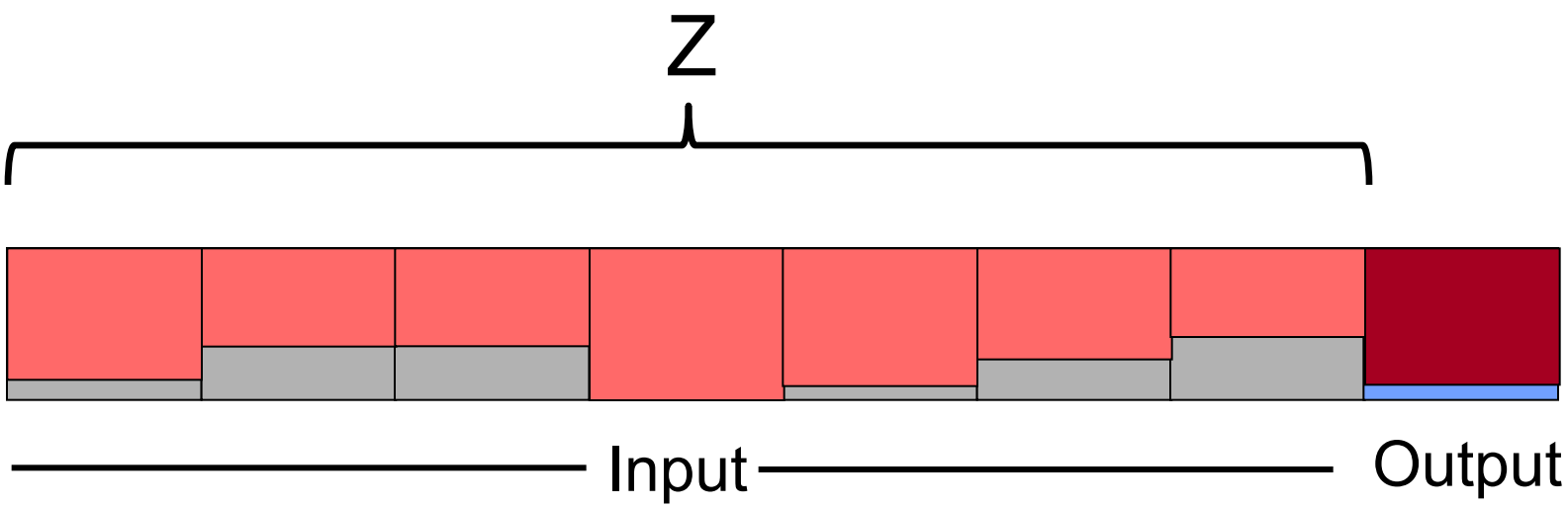
Example of merge



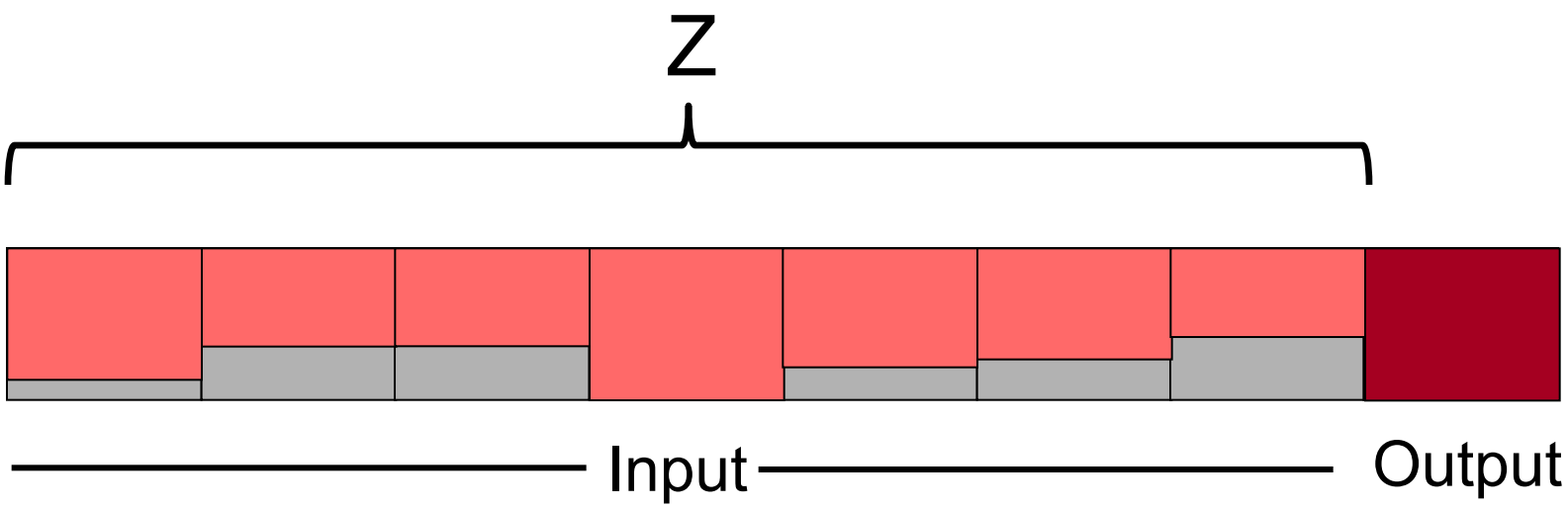
Example of merge



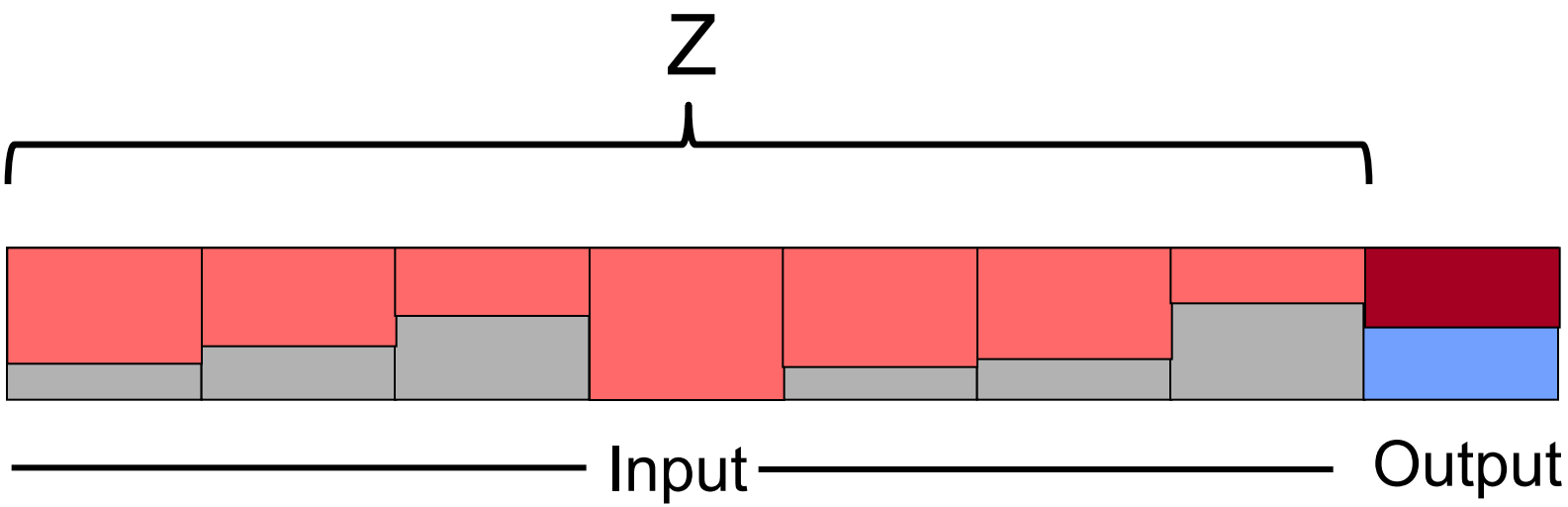
Example of merge



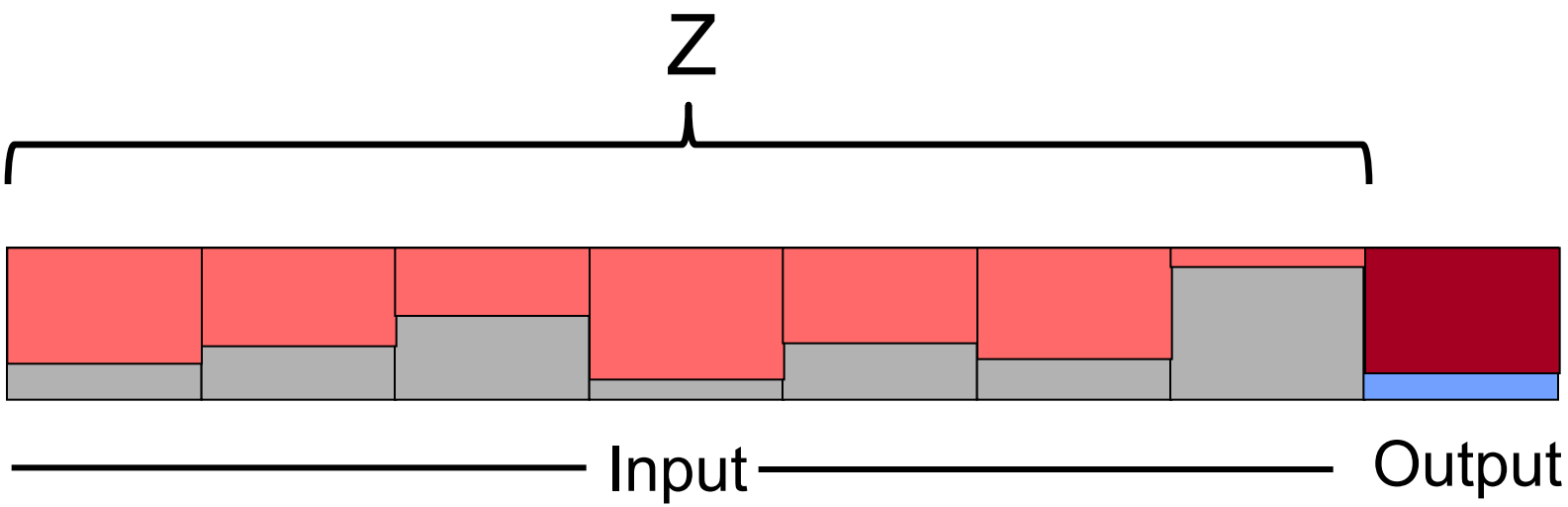
Example of merge



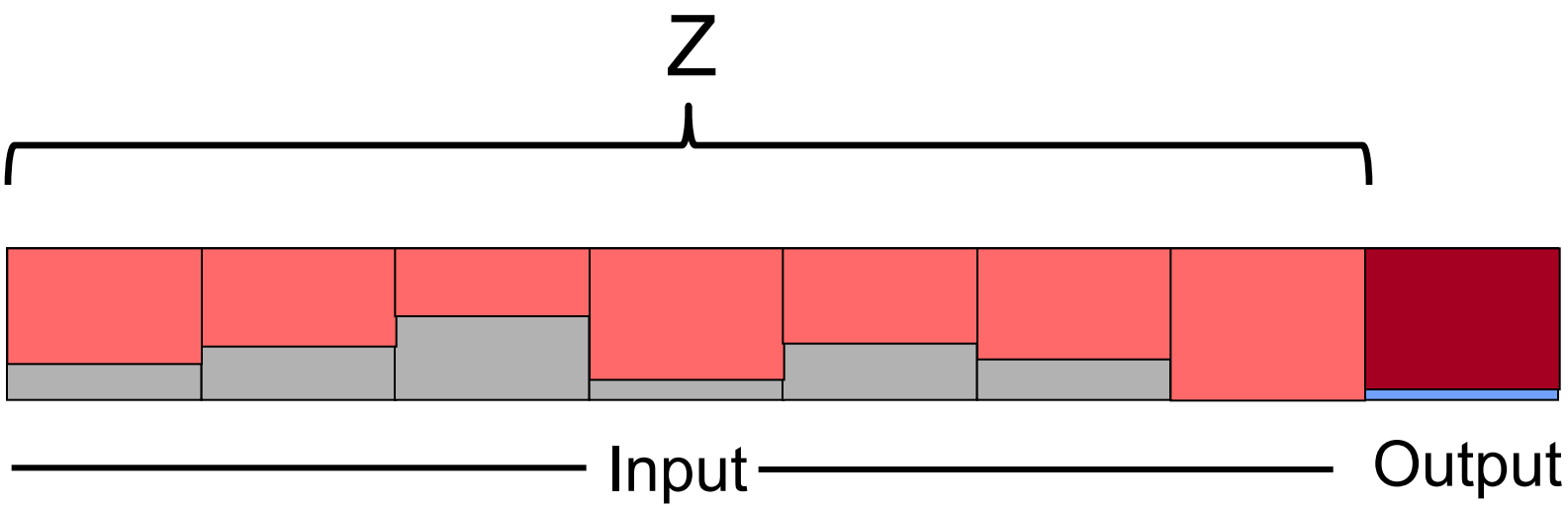
Example of merge



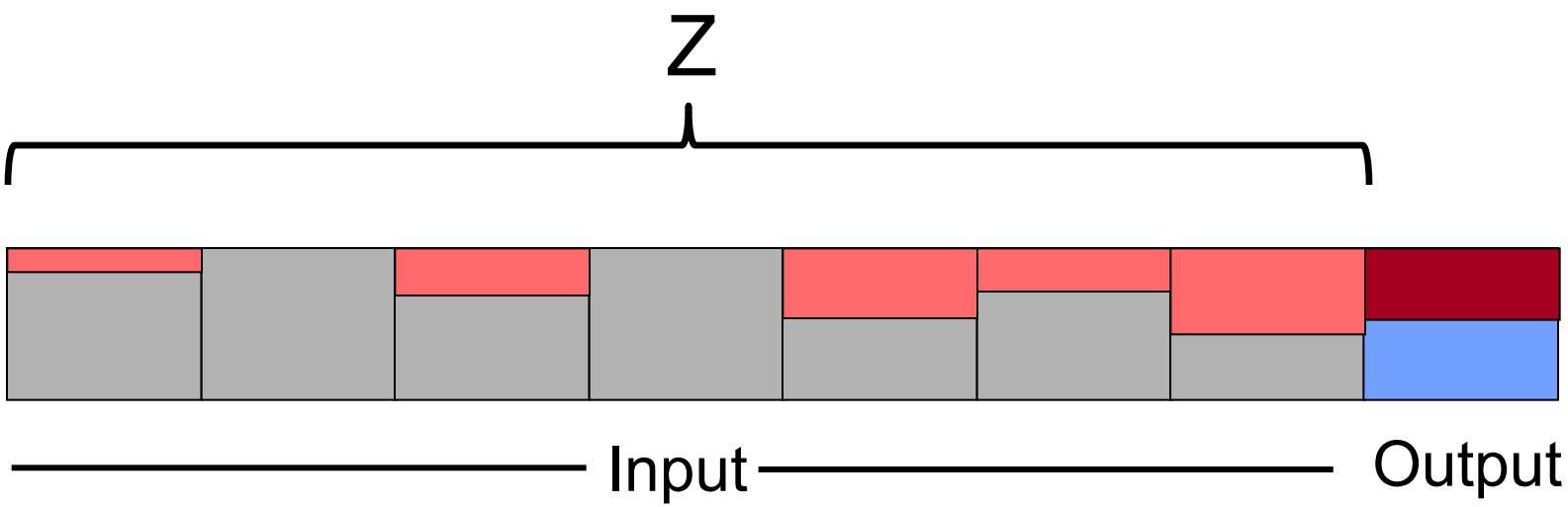
Example of merge



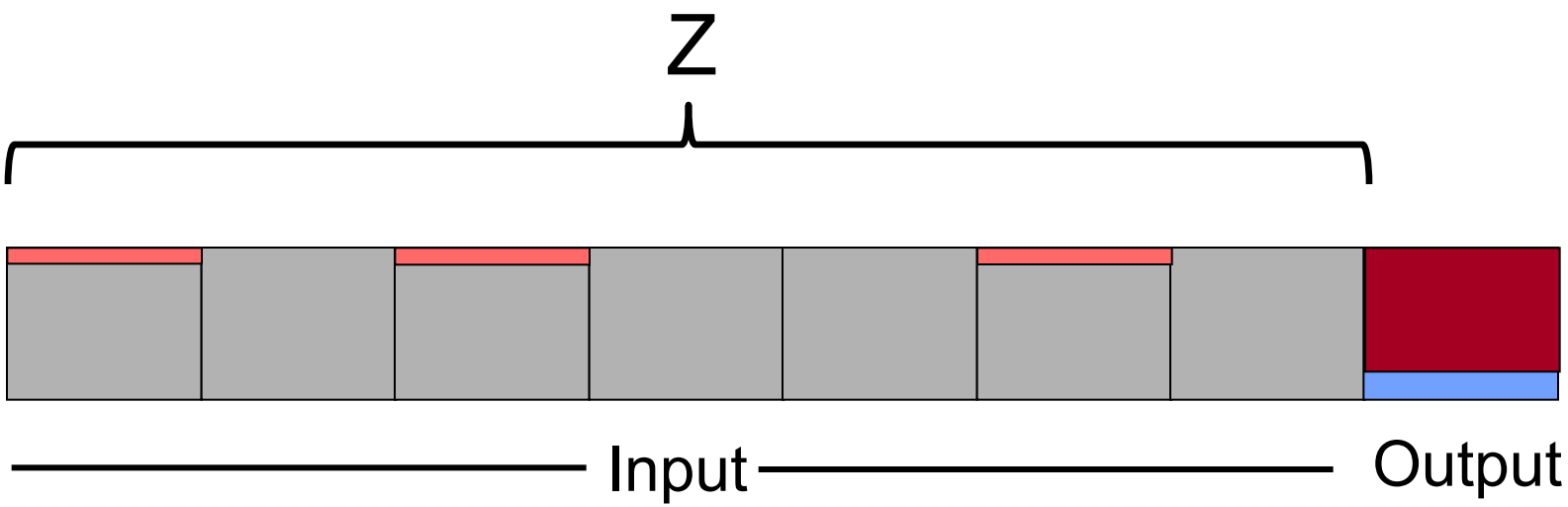
Example of merge



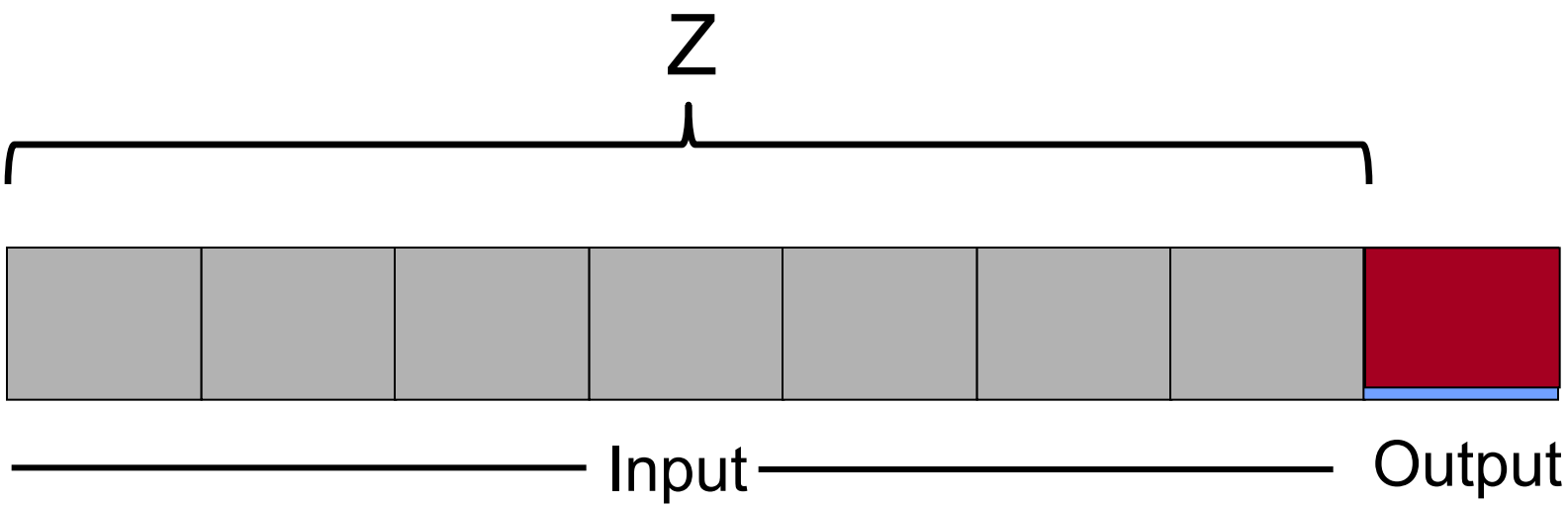
Example of merge



Example of merge



Example of merge



The sort-merge algorithm: complexity

If B is the number of pages of relation R, and we use F buffer frames, then the number of runs created at pass 0 is S = B/F.

Each time we enter in the repeat loop (2) we have a new pass. The number of passes depends on Z (whose value depends on the fact that we use Z+1 frames at each execution of the inner loop, and therefore Z is usually F-1).

At pass i > 0, the number of runs produced is S/Z^i; therefore, the number of runs produced at the various passes is:

$$S/Z^1,\ S/Z^2, S/Z^3, ..., S/Z^k$$

and the algorithm stops when $1 >= S/Z^k$ (i.e., $k >= \log_Z S$).

At every pass (including 0) we read and write every page of the relation, therefore the cost (in terms of page accesses) is:

$$\mathbf{2 \times B \times (\lceil \log_Z S + 1 \rceil) = 2 \times B \times (\lceil \log_Z (B/F) + 1 \rceil)}$$

If Z=F-1, then $\log_Z(B/F) = \log_{F-1}B - \log_{F-1}F$, and the cost in terms of page accesses can be approximated to $\mathbf{2 \times B \times \log_{F-1} B}$.

Example

- 1.000.000 records in 12.500 pages
- Use 10 buffer pages in memory
- 4 passes
- 100.000 disk I/Os
  - 17 minutes versus 1 hour for 2-way sort

Exercise 1

Under which conditions I can do the algorithm in two passes?

(remember that a pass is a part of the algorithm where we read and write every page of the relation)

Exercise 1: solution

If the algorithm has only two passes, then it has this form:

- Pass 0: read file and create sorted runs
- Pass 1: merge runs

- Given F buffer frames, if we have only one pass for merging (i.e., pass 1), we must have (F-1) runs to work with at pass 1 (one frame is for the output). Each such run has been sorted at pass 0 in the buffer, and therefore each run cannot contain more than F pages

- Therefore, we can do the algorithm in two passes if

$$B \leq F * (F-1)$$

which we can approximated to

$$B \leq (F-1)^2$$

Exercise 2

- Under which conditions can we do the algorithm in three passes?
- Under which conditions can we do the algorithm in N passes?

Recursive formulation of the multipass sort-merge algorithm

- Base step: If R fits in the F frames available in the buffer (i.e., B(R) ≤ F), then sort R in the buffer, using any main memory algorithm and write the sorted relation to secondary storage
- Inductive step: If R does not fit in the buffer, partition the pages of R into F-1 groups $R_1,...,R_{F-1}$, and recursively sort R_i for each i=1,2,...,F-1. Then merge the F-1 sorted sublists using one frame for the output and write the sorted relation to secondary storage.

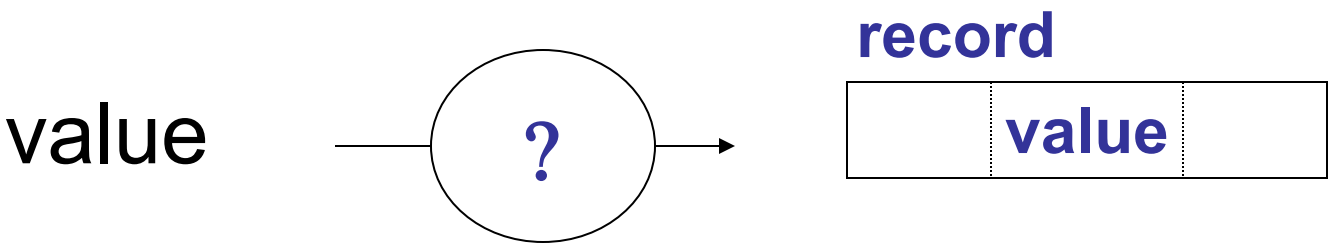
Exercise: prove that this formulation is equivalent to the "iterative" formulation we have described before.

5. Access file manager

5.1 Pages and records
5.2 Simple file organizations
5.3 Index organizations

The notion of index

An index is any method that takes as input a property of a set of records – typically the value of one or more fields, and finds the records with that property "quickly"



The notion of index

Any *index* organization is based on the value of one or more predetermined fields of the records of the relation we are interested in, which form the so-called search key.

- Any subset of the fields of a relation can be taken as the search key for the index
- Note: the notion of search key is different from the one of key of the relation (a key is a minimal set of fields uniquely identifying the records of the relation)

Obviously, it may happen that the search key coincides with the key of the relation.

Data entry, index entry and data record

An implementation of a relation R by means of an index-based organization comprises:

- Index file (sometimes absent, i.e., virtual, e.g., in the case of hash-based index), containing
  - Data entries, each containing a value k of the search key, and used to locate the data records in the data file related to the value k of the search key
  - Index entries (at least for some index organizations), used for the management of the index file.
- Data file, containing the data records, i.e., the records of relation R

Properties of an index

1. Organization of the index
2. Structure of data entries
3. Clustering/non clustering
4. Primary/secondary
5. Dense/sparse
6. Simple key/Composite key
7. Single level/multi level

Organization of the index

- Sorted index

  the index is a sorted file

- Tree-based index

  the index is a tree

- Hash-based index

  the index is a function from search key values to record addresses

Possible structures of a data entry

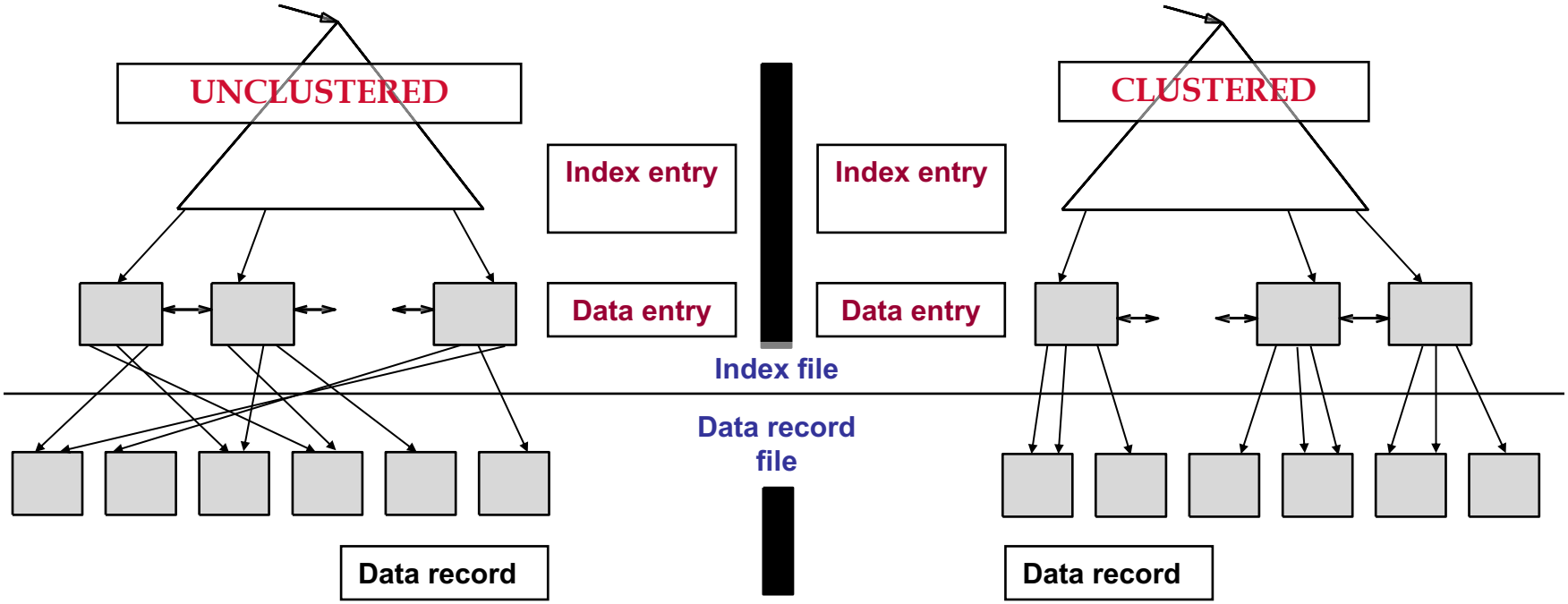
There are three main alternative techniques for storing a data entry whose search key value is k (such a data entry is denoted with **k***):

1. **k*** is a data record (with search key equal **k**)
   - this is an extreme case, because it does not really correspond to having data entries separated by data records (usually, the hashed file is an example of this case)
2. **k*** is a pair **(k,r)**, where **r** is a reference (for example the record identifier or the identifier of the page where the record is) to a data record with search key equal **k**
   - the index file is separate from the data file
3. **k*** is a pair **(k,r-list)**, where **r-list** is a list of references (for example, a list of record identifiers) to data records with search key equal **k**
   - the index file is separate from the data file
   - better use of space, at the cost of variable-length data entries

Note that if we use more than one index on the same data file, at most one of them will use technique 1.

Clustering/non-clustering

An index (for data file F) is *strongly clustering (also called strongly clustered)* when its data entries are stored according to an order that is coherent with (or, identical to) the order of data records in the data file F. Otherwise, the index is *non strongly clustering (or, unclustered)*.



Clustering/non-clustering

Given an index *IND*, in the case where the corresponding data file is not stored in any order, or in the case where it is ordered according to a different ordering key with respect to *IND*, the index is not strongly clustering.

However, *IND* may still be *weakly clustering*: we say that an index *IND* is weakly clustering if, for every value V of the search key, all the tuples of the indexed data file with value V for the search key used in *IND* appears in the same page of the data file (or, on roughly as few «linked» pages as can hold them).

Indeed, some authors define an index *IND* to be clustering if it is weakly clustering according to our definition.

Note that an index that is strongly clustering is also weakly clustering (although the converse is not true). In the following, when we say "clustering" (or, clustered), we implicitly mean "strongly clustering", and when we want to refer to weakly clustering, we explicitly use the term "weakly clustering". Similarly, when we say non-clustering (or, unclustered) we mean non-strongly clustering.

Clustering/non-clustering

- An index that is based on sorting and whose data entries are stored with technique 1 is clustered by definition.
- As for the other alternatives, an index is clustered only if the data records are sorted in the data file according to the order of the values of the search key.
- If the index is clustered, then it can be effectively used for interval-based search (see later for more details).
- In general, there can be at most one clustered index per data file, because the order of data records in the data file can be coherent with at most one index search key.

Primary and secondary indexes

- A *primary key index (or simply primary index)* is an index on a relation R whose search key includes the primary key of R. If an index is not a primary key index, then is called *non-primary key index (also called secondary index)*.

- NOTE: In some text, the term “primary index” is used with the same meaning that we assign to the term “strongly clustering index”, and the term “secondary index” is used with the same meaning that we assign to the term “non strongly clustering index”

Primary and secondary indexes

- Let us call *duplicate* two data entries with the same values of the search key.

  → A primary index cannot contain duplicates

  → Typically, a secondary index contains duplicates

  → A secondary index is called *unique* if its search key contains a (non-primary) key. Obviously, a unique secondary index does not contain duplicates.

Primary and secondary indexes

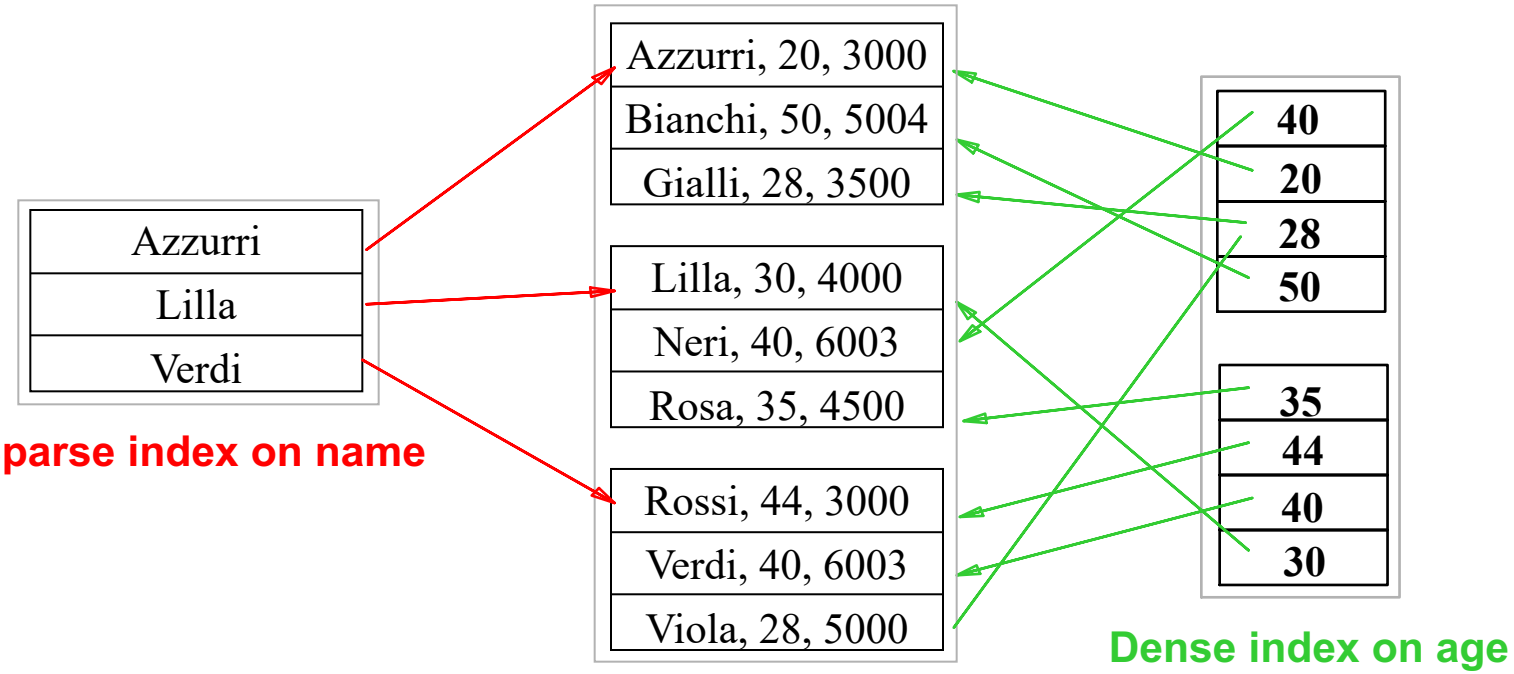
A *secondary, non-unique index* can be organized so as not to contain duplicates. Two possible organizations of this kind are:

– The index uses alternative (3), and therefore every relevant value of the search key is stored only once in the index, but with a list of rids associated to it.

– The index uses alternative (2), and the index is weakly clustering. In this case, for each relevant value K of the search key, we have only one data entry in the index, pointing to the first data record R with the value K for the search key. Since the index is weakly clustered, the other data records with value K for the search key follow immediately R in the data file.

Sparse vs dense

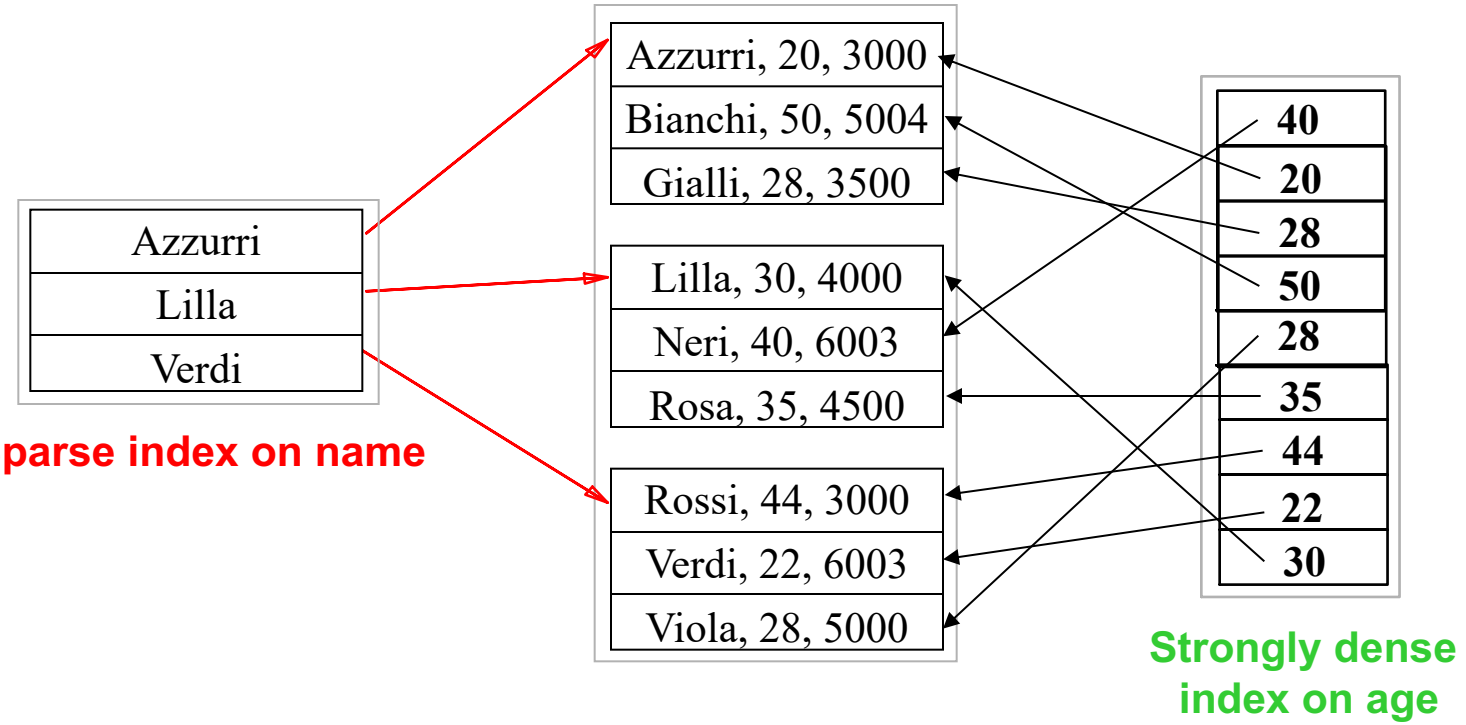
An index is *dense* if every value of the search key that appears in the data file appears also in at least one data entry of the index. An index that is not dense is *sparse*, i.e., in a sparse index, only some of the search-key values have a corresponding data entry. Notice that a sparse index is obviously clustered.



- An index using technique 1 is dense by definition
- A sparse index is more compact than a dense one
- A sparse index is clustered; therefore we have at most one sparse index per data file

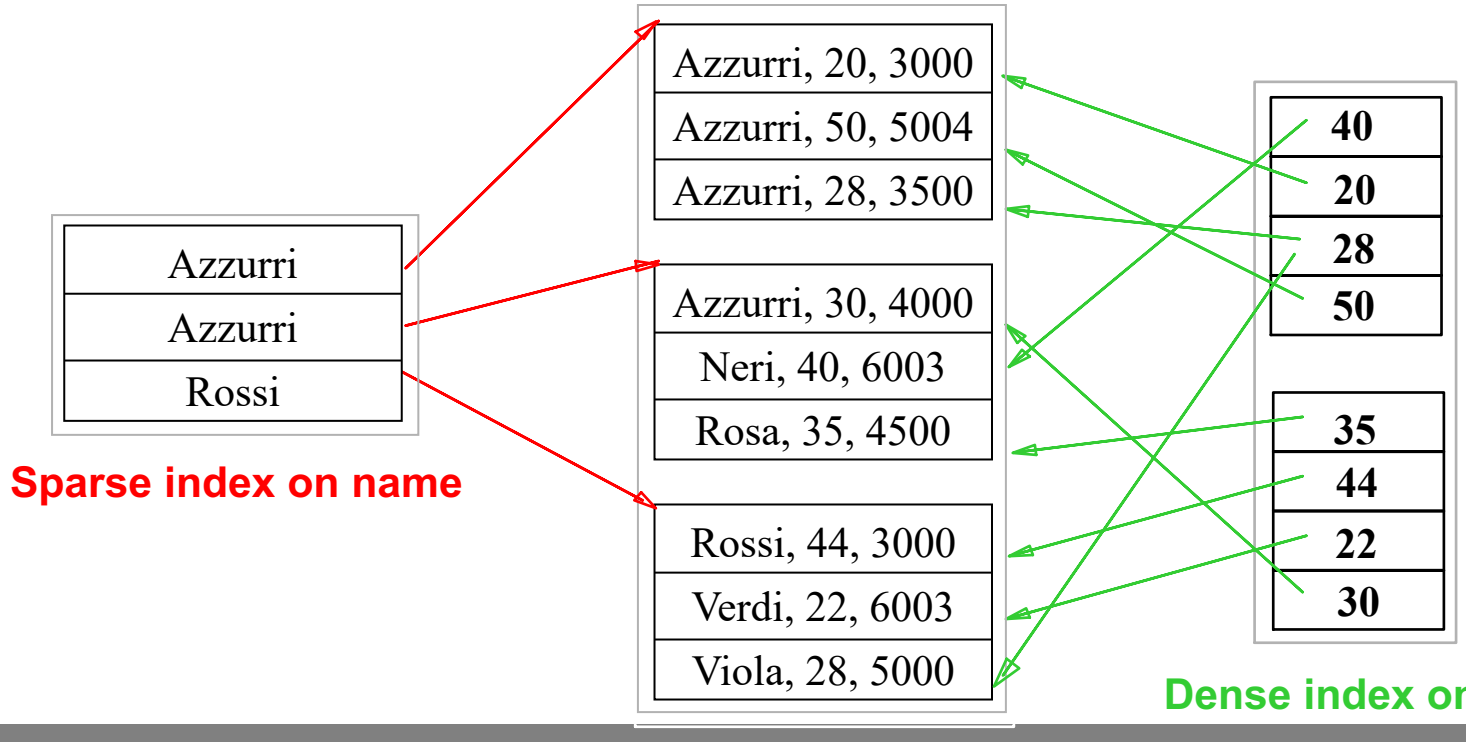
Sparse vs dense

A *dense* index is strongly dense if we have exactly one data entry for each data record of the data file, and the value of the search key in each data entry is the value held by the referenced data record. This means that if we use alternative 2 or 3, the references associated to data entries are record identifiers (see green arrows in the figure).



Sparse vs dense

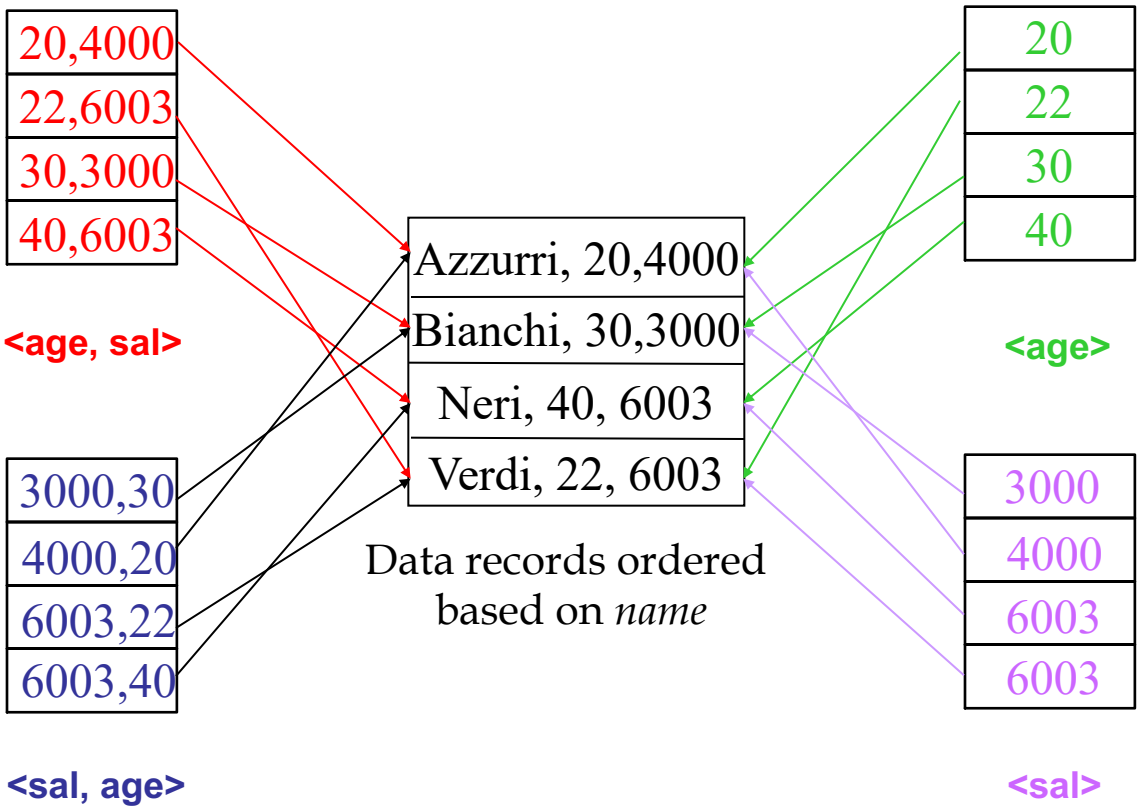
As said before, in a *sparse* index, only some of the search-key values have a corresponding data entry. Typically, we have one data entry per data page, where the value of the search key of the data entry is the value held by the first data record in the corresponding data page (recall that a sparse index is clustered). This means that if we use alternative 2 or 3, the references associated to data entries denote page identifiers (see red arrows in the figure).



Single vs composite key

- A search key is called *simple* if it is constituted by a single field, otherwise is called *composite*.
- If the search key is composite, then a query based on the equality predicate (called equality query) is a query where the value of each field is fixed, while a query that fixes only a prefix of the sequence of the fields forming the search key is actually a range query.
- A composite index supports a greater number of queries. With a composite index, a single query is able to extract more information. For example, when all the fields that are relevant in the query are part of the search key, we can even avoid accessing the data records, i.e., we can carry out an *index-only evaluation*
- On the other hand, a composite index is generally more subject to update than a simple one.

Single vs composite key

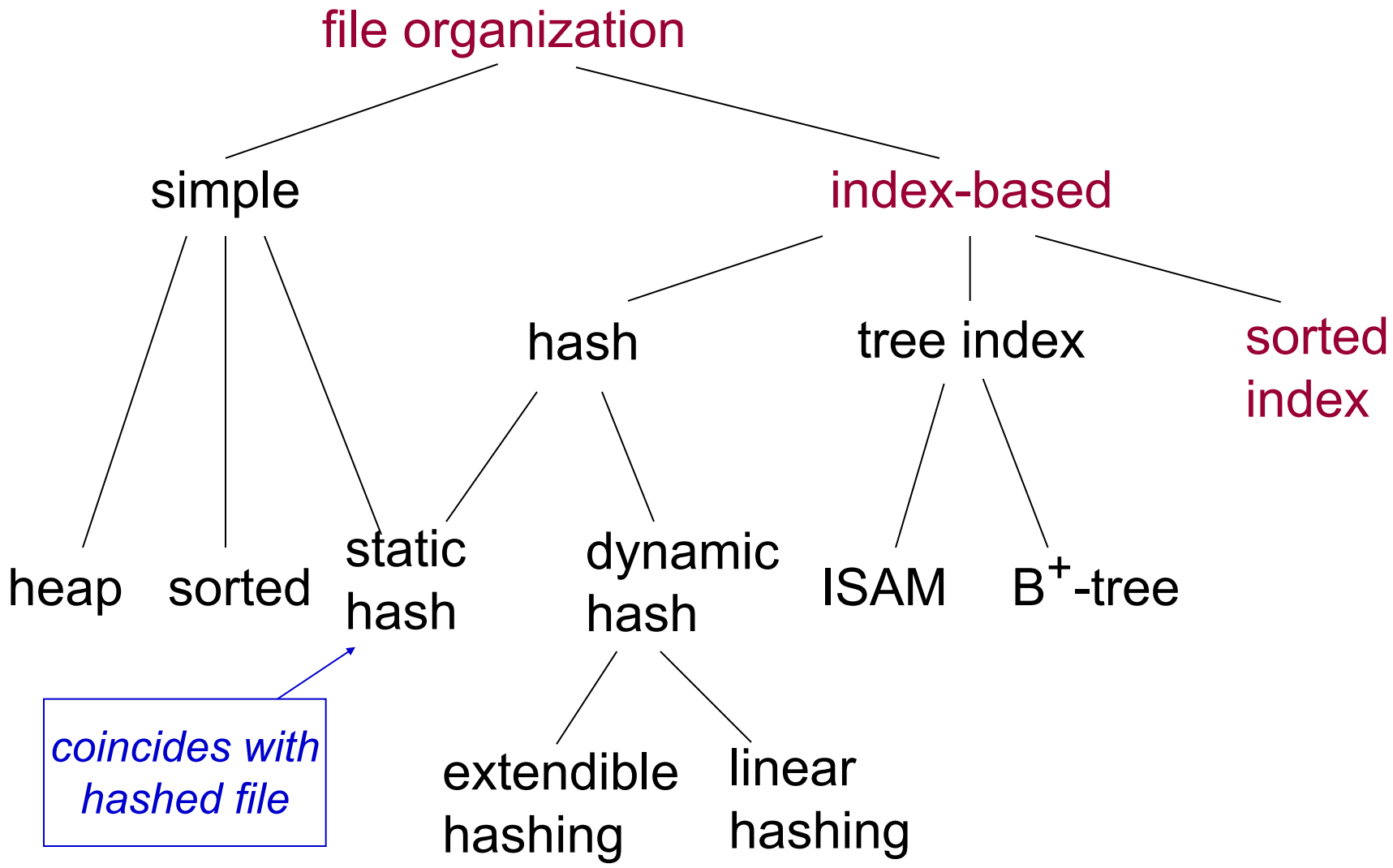


Single level/multi level

- A single level index is an index where we simply have a single index structure (i.e., a single index file), and the indexed data file

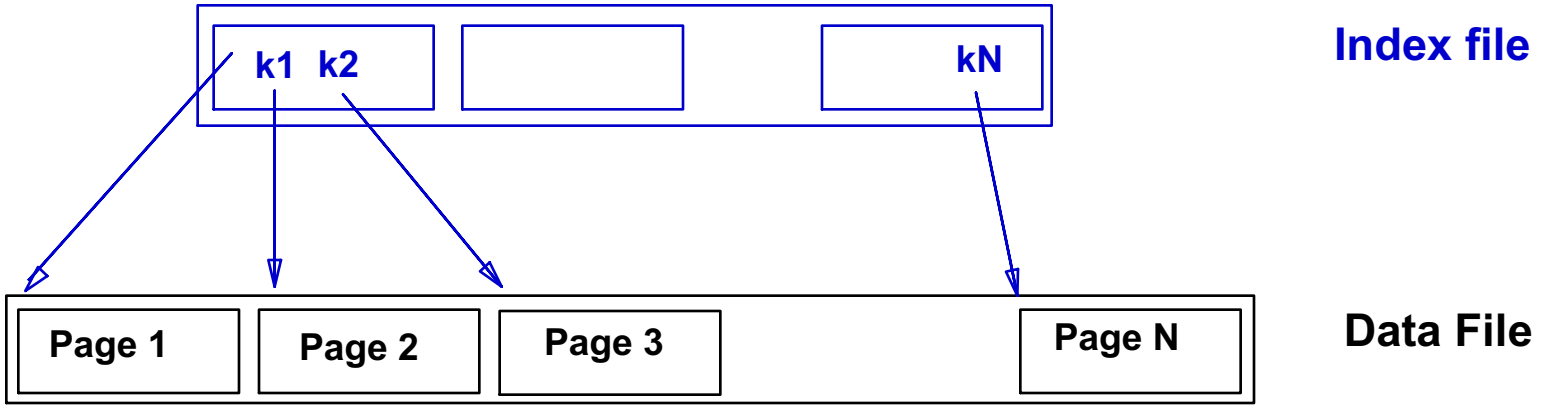
- A multi level index is an index organization where an index is built on a structure that is in turn an index file (such index file is either for a data file or, recursively, for another index structure).

Sorted index organization



Sorted index: basic idea

- Find all students with avg-grade > *27*
  - If students are ordered based on avg-grade, we can search through binary search
  - However, the cost of binary search may become high for large files
- Simple idea: create an auxiliary sorted file (index), which contains the values for the search key and pointers to records in the data file



The binary search can now be carried out on a smaller file (data entries are smaller than data records, and we can even think of a sparse index)

Sorted index

- Clustering sorted index
  - The data file is a sorted file, with the file sorted on the same search key of the sorted index
  - Usually (but not always) the search key coincides with the primary key (if so, the index is a primary key index)
  - This organization is also called indexed sequential file
- Non-clustering sorted index
  - The data file is either unsorted (for example, a heap), or is a sorted file, but sorted on attributes different from the search key of the sorted index

- In both cases, alternative 1 is not used! In what follows, if not otherwise stated, we assume to use alternative 2

Clustering sorted index: two cases

1. Clustering, sorted index, primary or unique (i.e., on a relation key)

2. Clustering, secondary non-unique sorted index (i.e., on non-key attributes)

Clustering sorted index: the first case

1. Clustering, sorted index, primary or unique (i.e., on a relation key)

2. Clustering, secondary non-unique sorted index (i.e., on non-key attributes)

Clustering sorted index (primary or unique)

- Dense

  every value of the search key that appears in the data file appears also in at least one data entry of the index (or, strongly dense: there is one data entry in the index file for every occurrence of value of the search key in the data file)

- Sparse

  only some of the data records have a corresponding data entry in the index file, i.e., a data entry with the same value of the search key. Often, there is one data entry per page in the data file.

Thanks to Hector Garcia Molina (Stanford University) for some of the figures/animations in the next slides.

Clustering sorted index (unique, dense)

Sequential sorted file

10	
20	

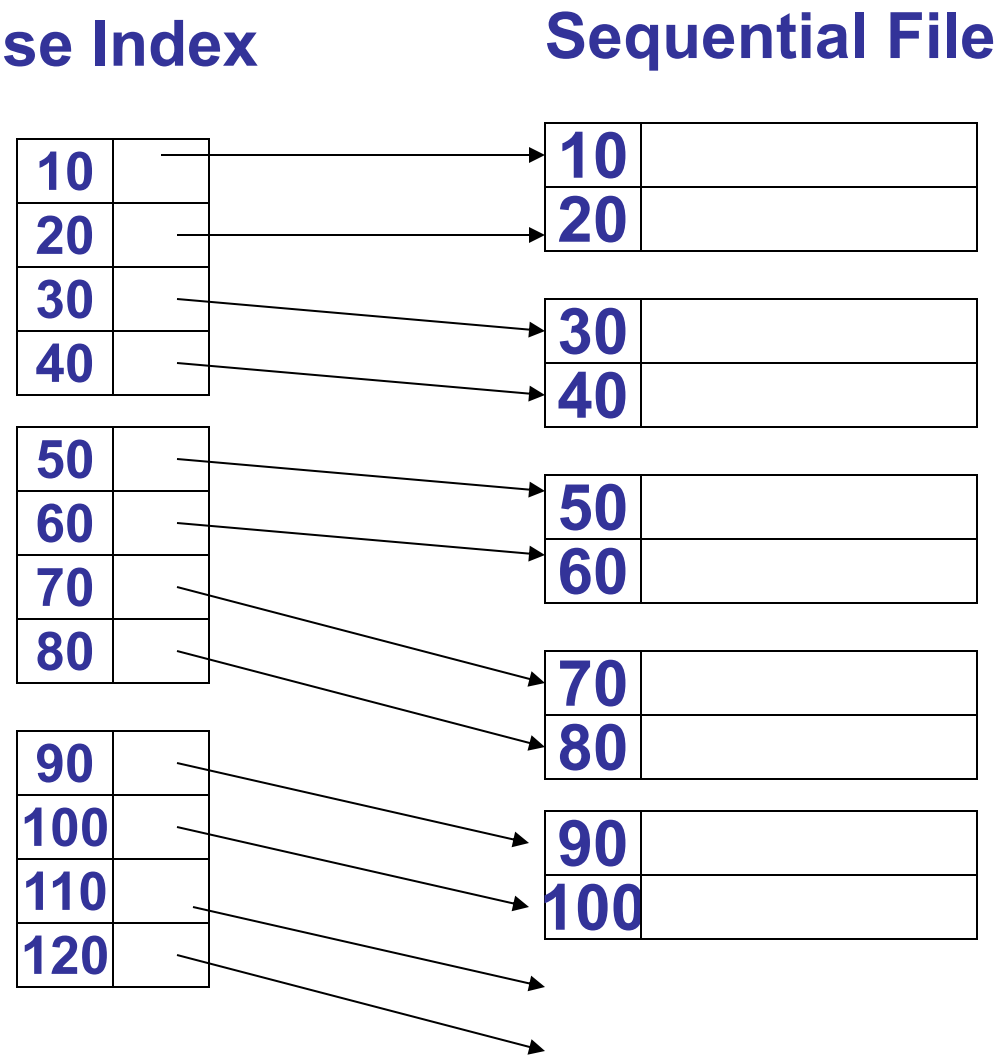
30	
40	

50	
60	

70	
80	

90	
100	

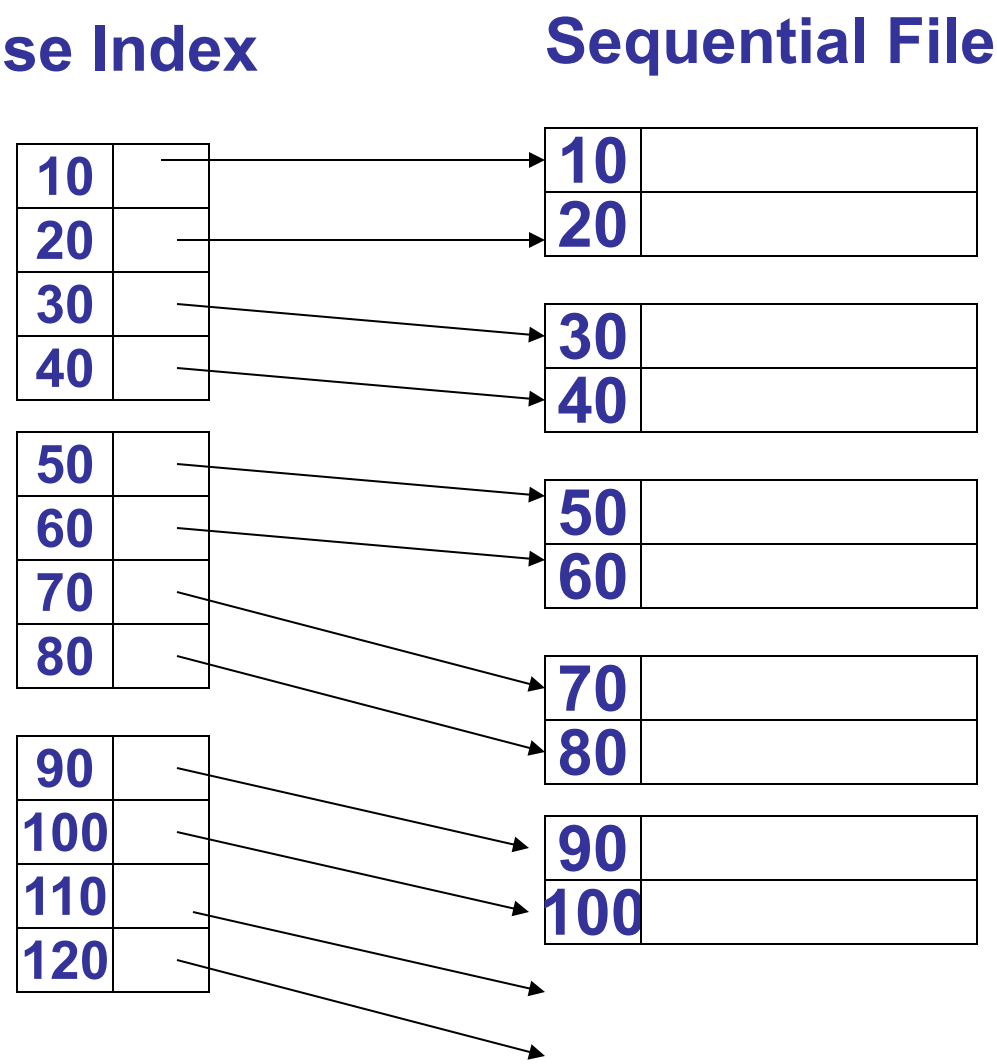
Clustering sorted index (unique, dense)



Clustering sorted index (unique, dense)

Being clustering, the index and the data file have the same ordering of the search key values

Note that the pages in both the data file and the index can be contiguous or chained by a list with pointers



Clustering sorted index (unique, dense)

- Since a data entry (key and pointer) takes in general much less space that a complete data record, we use many fewer pages for the index than for the data file.
- The index is especially advantageous when it, but not the data file, can fit in main memory (the buffer), in which case we can find any record given its search key with only one disk page access
- In any case, we can use binary search or interpolation search on the index for a search on equality on the search key.

Clustering sorted index (unique, dense): example

We have a relation with 1.000.000 tuples, ten of which fit in a 4096-byte data page. The total space required is over 400 megabytes (maybe too much for main memory). Suppose the fields are 30 bytes and pointers are 8 bytes. We can therefore keep 100 key-pointer pairs in a page.

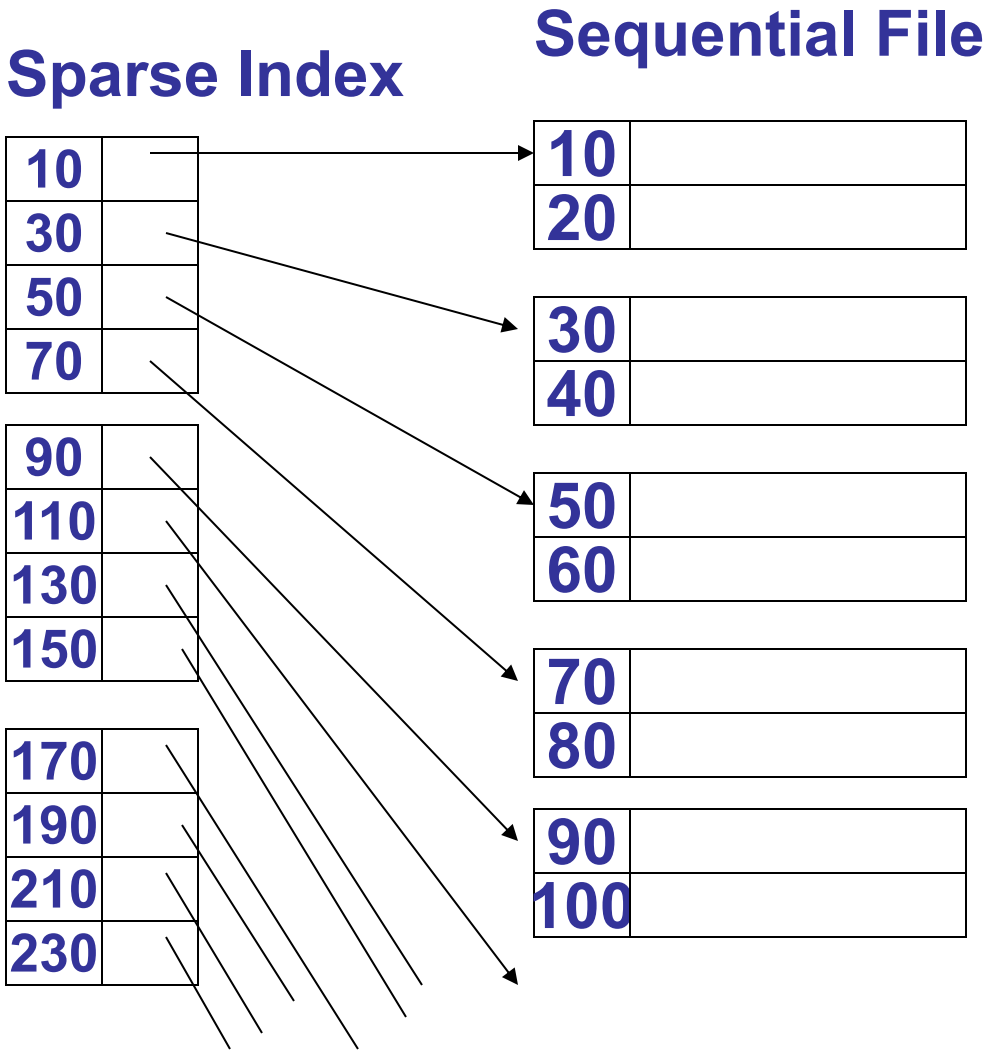
A dense sorted index requires 10.000 pages, or 40 megabytes. It is not unusual to have such a space in the buffer. If we do not have it, since $\log_2 10.000$ is about 13, we only need at most 13 or 14 page accesses in a binary search for a key.

Clustering sorted index (unique, sparse)

- If a dense index is too large, we can use a sparse index, in particular a sparse index that contains one data entry per page in the data file.
- In a data entry for a page P, the search key value is the one of the first record in the page P.
- To find the record with key K, we search the index (e.g., through binary search) for the largest key value less than or equal to K, and we follow the associated pointer to a data page, where we must search for the record with key value K

Clustering sorted index (unique, sparse)

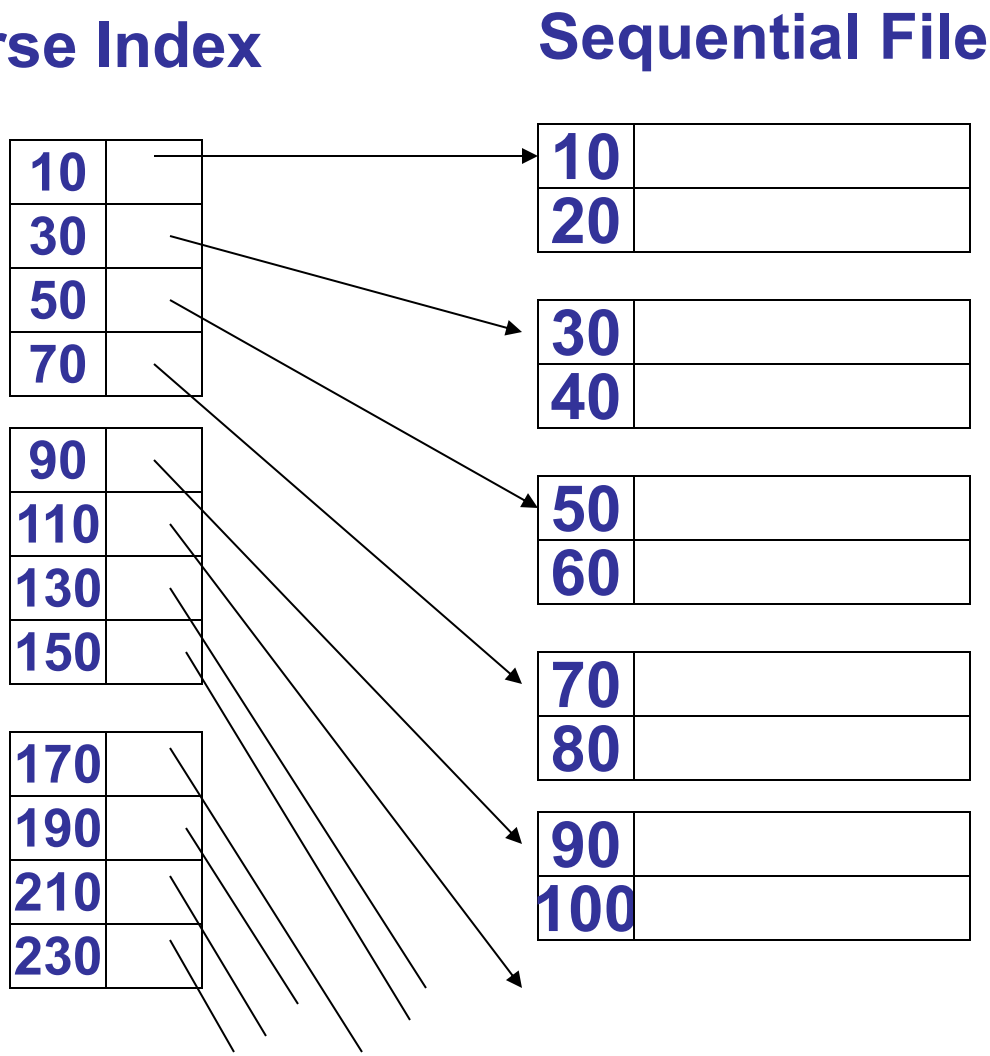
Suppose we look for record R with search key 40. In the index we search for the maximum value that is equal or greater than 40: in this case we find 30. We follow the pointer associated to 30 and we search for R in the page pointed by 30. In this case, we find 40. If 40 was not there, it would had meant that no data entry with 40 is in the data file



Clustering sorted index (unique, sparse)

Note that page pointer can be smaller than record pointers (rids)

Note also that, as before, the pages in both the data file and the index can be contiguous or chained by a list with pointers



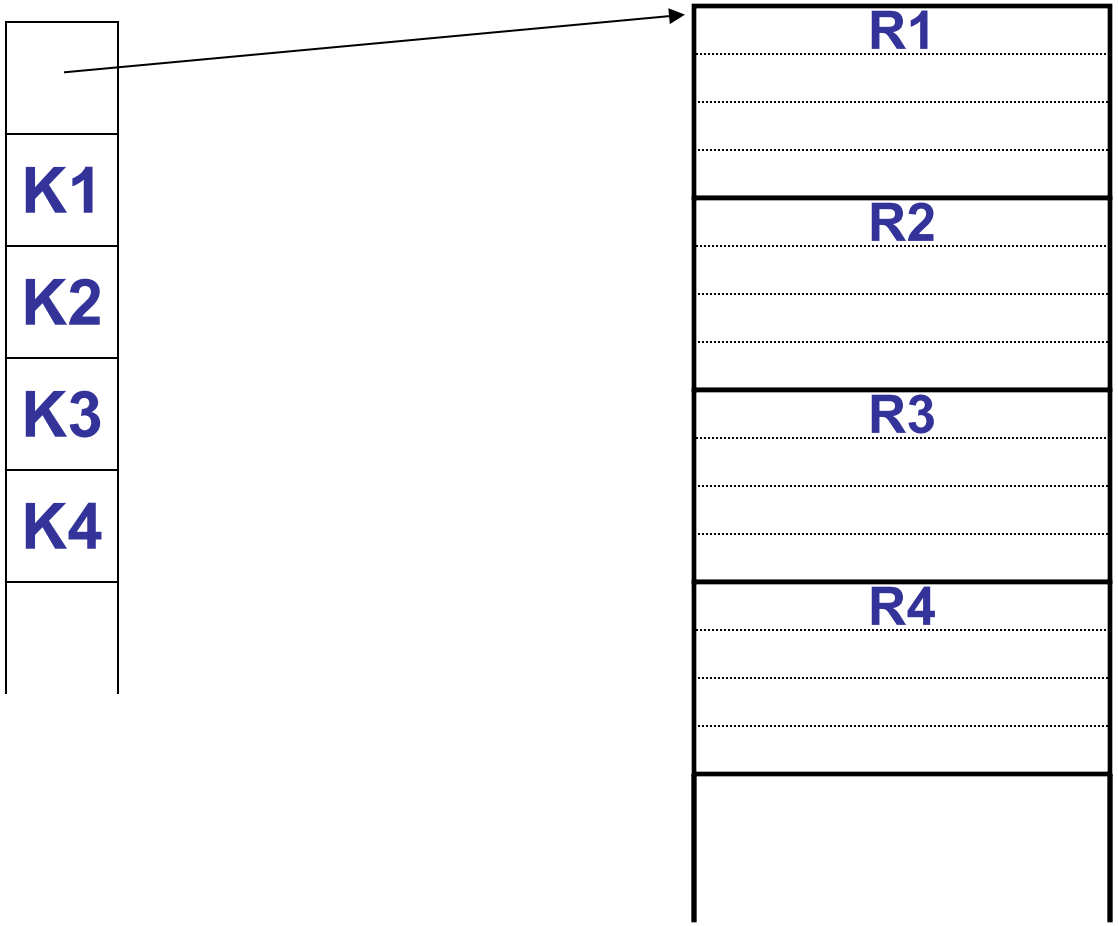
Clustering sorted index (unique, sparse): example

We have a relation with 1.000.000 tuples, ten of which fit in a 4096-byte data page. Since there are 100.000 data pages, and 100 key-pointer pairs fit in one page (assuming the fields are 30 bytes and pointers are 8 bytes), we need only 1.000 index pages if the index is sparse.

So the index uses only 4 megabytes, an amount that could plausibly be allocated in main memory.

Clustering sorted index (unique, sparse): contiguous data file

If the data file is contiguous, then we can omit storing pointers, because we can compute them



Sparse vs dense tradeoff in clustering index

- Dense:
  - Can tell if any record exists without accessing the file (index-only processing)
  - Requires more space
- Sparse:
  - Less index space per record: can keep more of index in memory
  - Checking whether a record exists in general requires accessing the page where the key might be found

From equality search to range search

All the considerations we have discussed for equality search with clustering sorted index can be extended to deal with equality range.

If we search for a range of search key values, we can search for the first value in the range, and then continue the search by moving forward to search for the other values, either in the sorted index (for example, if we do not need the value of other attributes), or in the data file. This is possible when the meaning of “the index is clustering” is the one that imposes that the data records are sorted coherently with the sorting the search keys in the index.

If the meaning of clustering is “weakly clustering” (all the data records with a fixed value for the search key appear on roughly as few pages as can hold them), then in order to use the index, we have to search in the index, considering one by one the values of the range, and, for each value, accessing the data pages.

Clustering sorted index: the second case (non-unique)

1. Clustering, primary (or unique) sorted index (i.e., on a relation key)

2. Clustering, secondary non-unique sorted index (i.e., on non-key attributes)

Clustering, secondary non-unique sorted index

sorted data file with duplicates

10	
10	

10	
20	

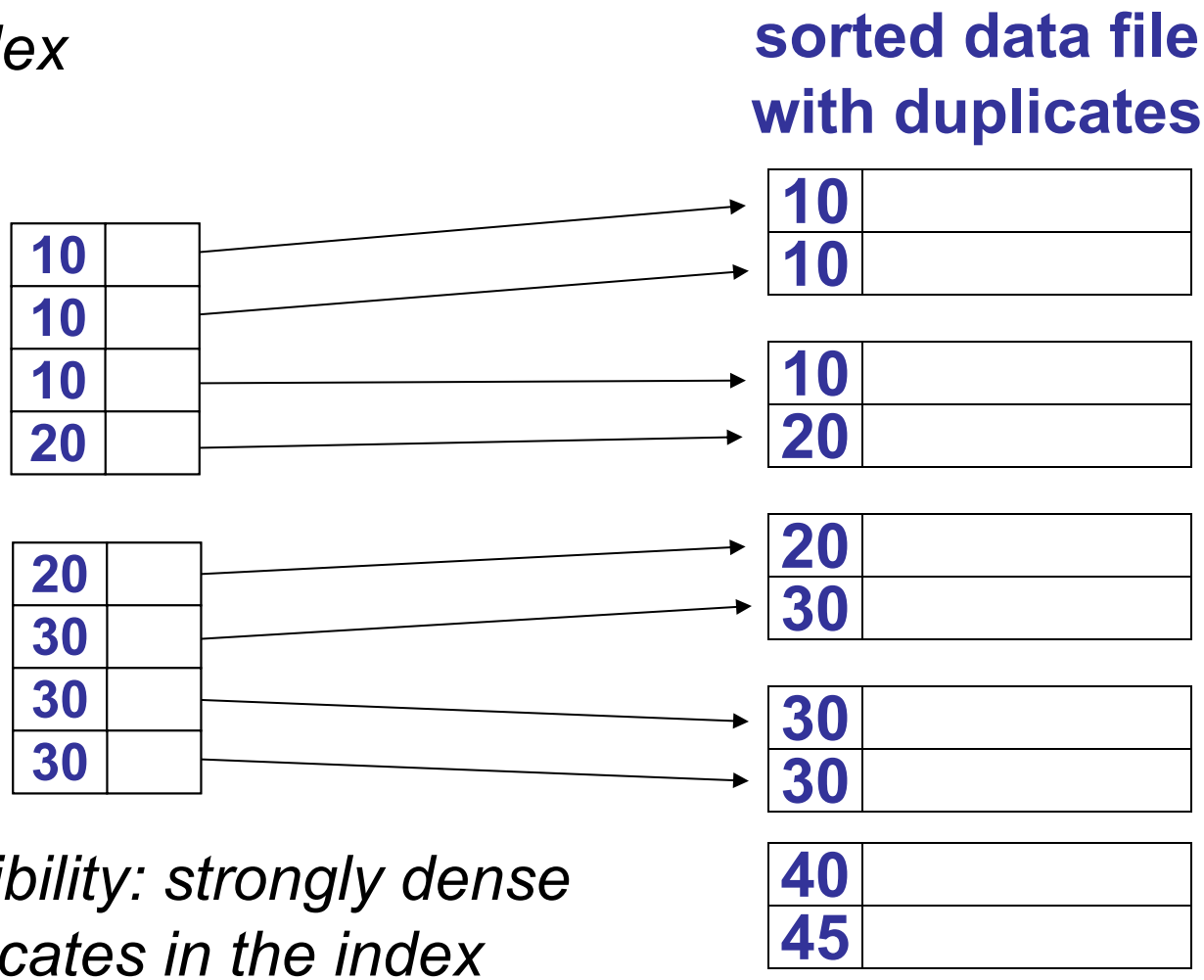
20	
30	

30	
30	

40	
45	

Clustering, secondary non-unique sorted index

Dense index



one possibility: strongly dense with duplicates in the index and with alternative 2

Clustering, secondary non-unique sorted index

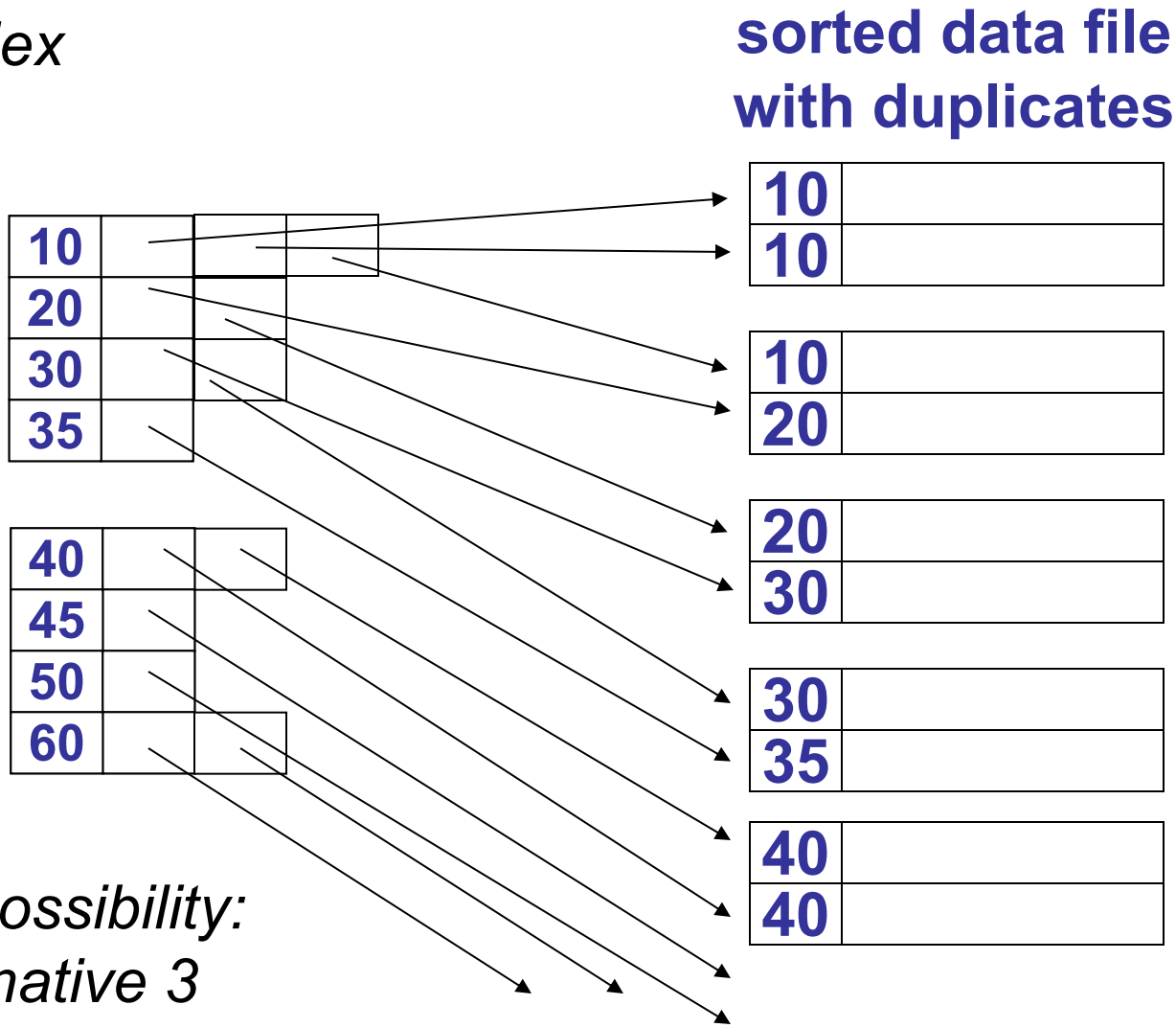
Strongly dense index (with duplicates in the index)

Finding all the records with a given value K for the search key can be done as follows:

- look for the first K in the index file,
- find all the other K's, which must immediately follow,
- pursue all the associated pointers to find the records with search key K

Clustering, secondary non-unique sorted index

Dense index



another possibility:
with alternative 3

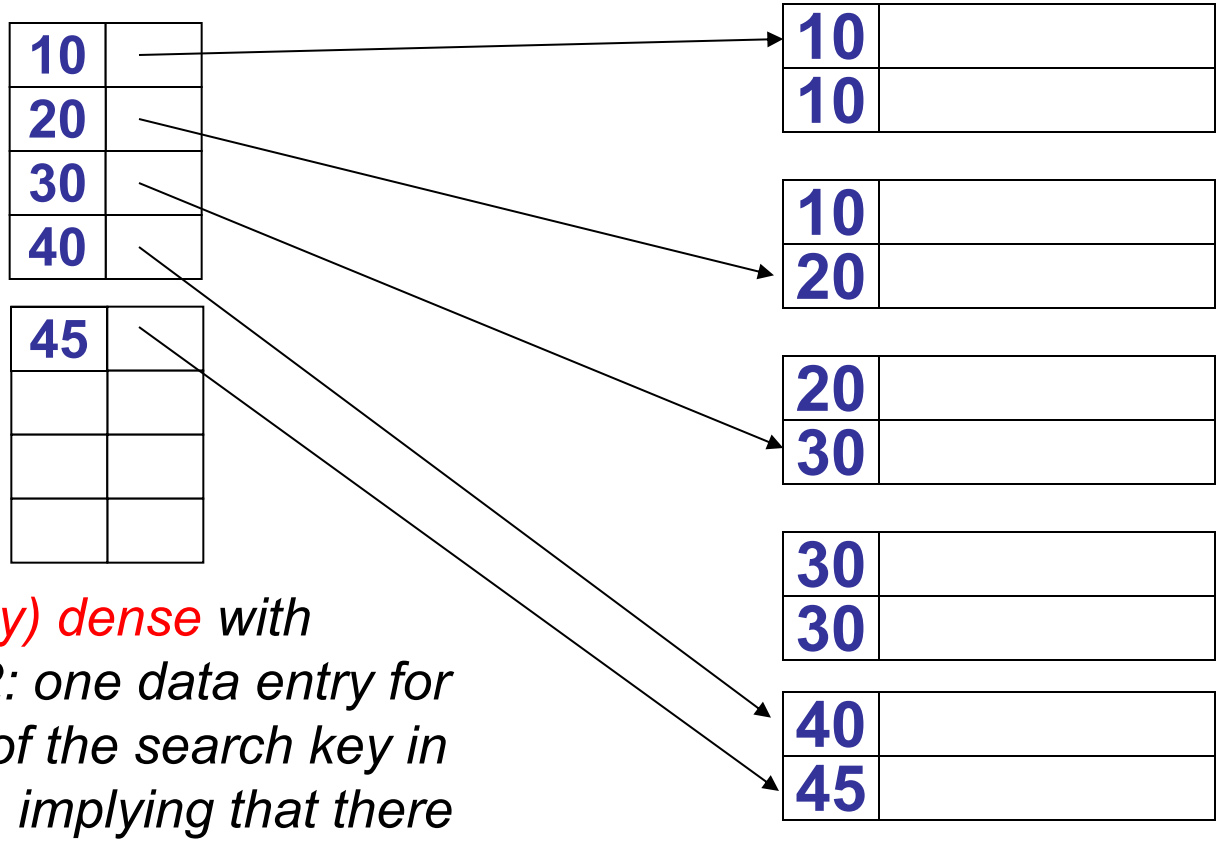
Clustering, secondary non-unique sorted index

Dense index with duplicates in the index

The above two solutions of dense index (alternative 2 with duplicates in the data entries, or alternative 3) are actually rarely adopted: indeed, we will see that the next method, the one without duplicates in the data entries using alternative 2, is better.

Clustering, secondary non-unique sorted index

Dense index



(non-strongly) dense with alternative 2: one data entry for each value of the search key in the data file, implying that there are no duplicates in the index

Clustering, secondary non-unique sorted index

Dense index without duplicates in the index

The pointer associated to a search value K goes to the first data record with value K for the search key.

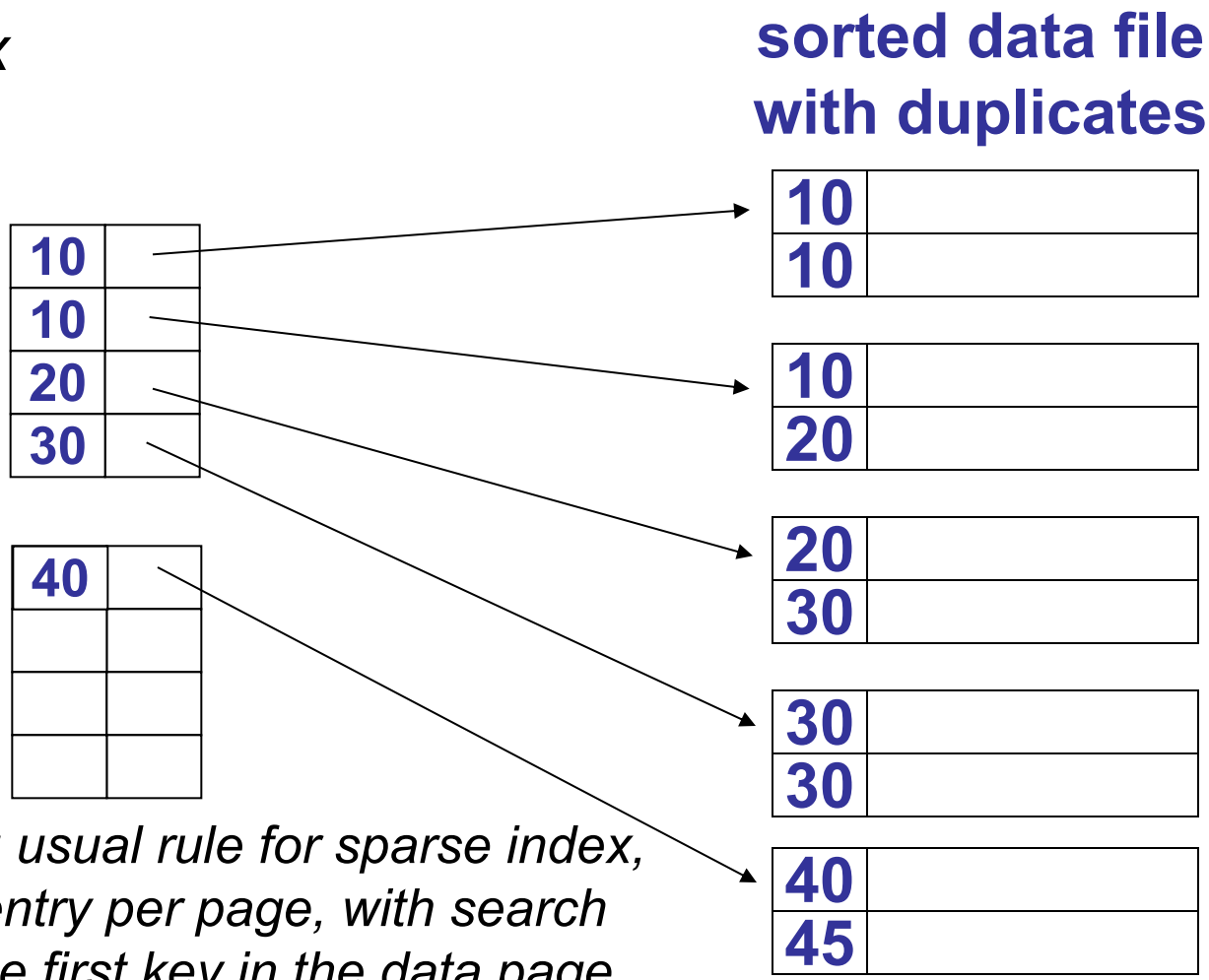
Finding all the records with value K (example: find 20 in the case of previous slide) for the search key can be done as follows:

- look for K in the index file, and follow the pointer to the first data records with value K for the search key
- find all the other data records by moving forward in the data file (taking advantage of the sorting)

Note that moving forward in the data file means either going forward sequentially if the data file is a sequential file, or following the pointers in the data file if its pages are in a linked list.

Clustering, secondary non-unique sorted index

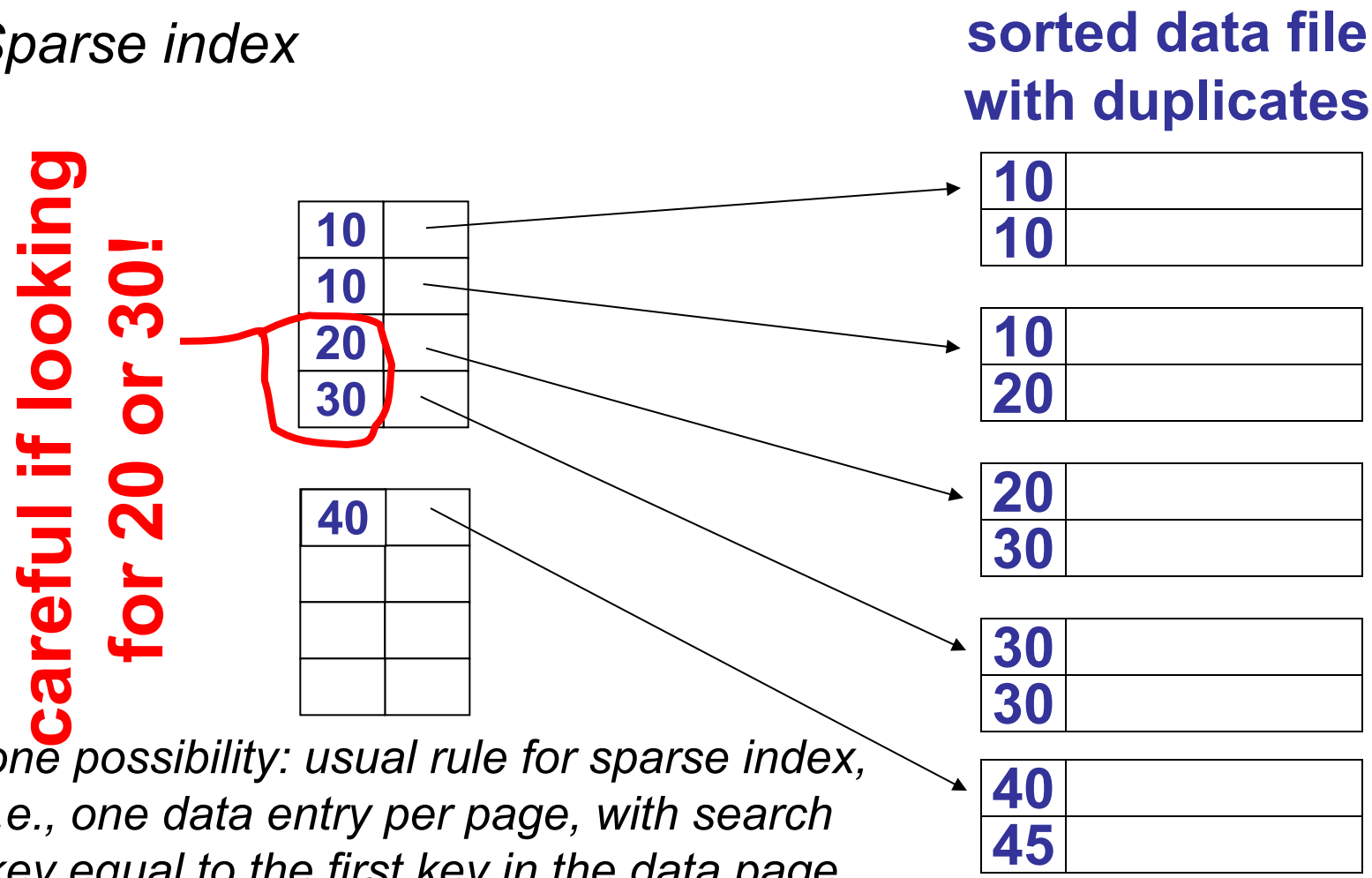
Sparse index



one possibility: usual rule for sparse index, i.e., one data entry per page, with search key equal to the first key in the data page (duplicates may occur in the index)

Clustering, secondary non-unique sorted index

Sparse index



one possibility: usual rule for sparse index, i.e., one data entry per page, with search key equal to the first key in the data page (duplicates may occur in the index)

Clustering, secondary non-unique sorted index

Sparse index with usual rule

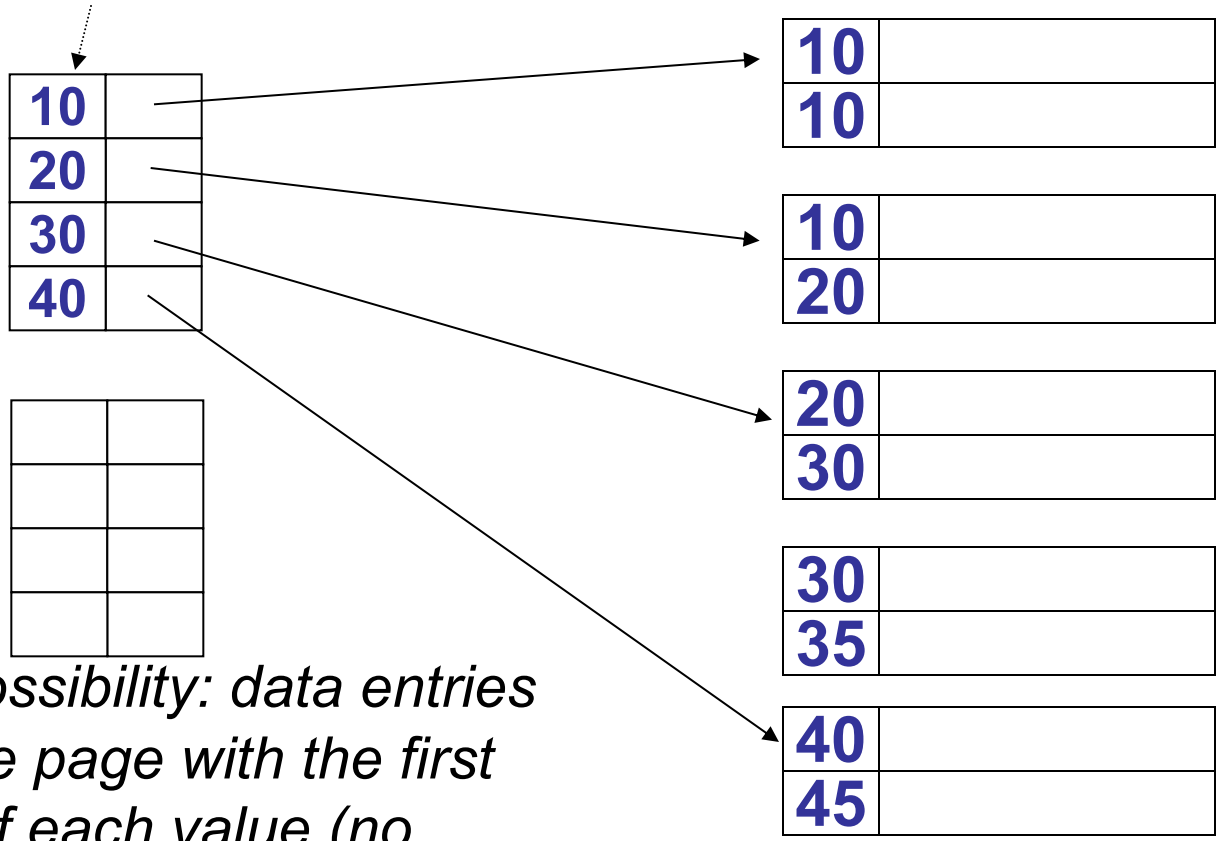
The index has data entries corresponding to the first search key on each page of the data file.

To find the records with search key K in the data file, we find the last entry E_1 in the index that has the key value less than or equal to K. We then move backward in the index until we either come to the very first data entry of the index, or we come to an entry E_2 with a key value that is strictly less than K. All the data pages that might have a record with search key K are pointed to by the data entries from E_2 to E_1 inclusive.

Example: find 20 in the case of previous slide

Clustering, secondary non-unique sorted index

Sparse index



another possibility: data entries point to the page with the first instance of each value (no duplicates in the sparse index)

Clustering, secondary non-unique sorted index

Sparse index where data entries point to the page with first instance of each value

Each data entry points to a data page. The data entry for a data file page holds the smallest search key in that page that is new, i.e., did not appear in a previous page. If a data file page does not contain a new value, there is no data entry for that data file page.

To find the records with search key K in the data file, we look in the index for the first data entry whose key is either

- equal to K, or
- less than K, but with the next key greater than K.

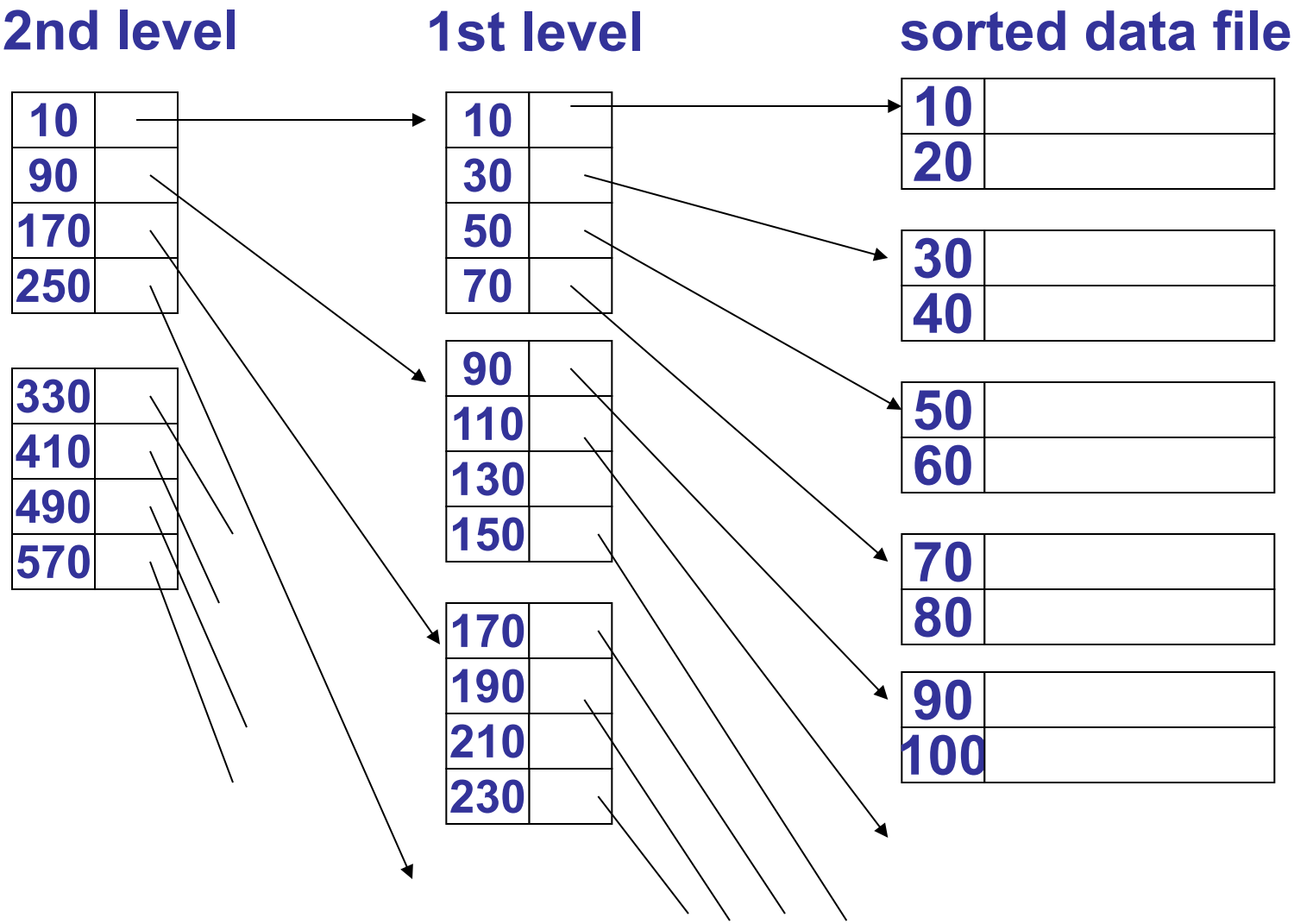
We follow the pointer in this data entry, and if we find at least one data record with key K in that page, then we search forward through additional pages until we find all the data records with search key K.

Example: find 20 and then 35 in the case of previous slide

Multiple levels of clustering sorted index

- We still may need to do many page access to search a sorted index.
- By putting a sorted index on the sorted index we can make use of the first level more efficient.
- The idea is valid also in the case of non-clustering index (see later)
- The idea can even be iterated, to form a 3rd level sorted index, and so on (the notion of tree-based index pushes this idea to the limit)

Multiple levels of the sorted index (clustering)



Multiple levels of clustering sorted index

- The first level index can be either dense, or sparse (as we saw before)
- However, the second and the higher levels must be sparse, because a dense index on an index would have essentially as many data entries as the first level index and therefore would give no advantages.
- Since the second level index probably fits in main memory, we only need two page accesses to find a data record given the value for the search key (if we have three levels, we will need just two or three page accesses)

Multiple levels of the sorted index: example

We have a relation with 1.000.000 tuples, ten of which fit in a 4096-byte data page. Since there are 100.000 data pages, and 100 key-pointer pairs fit in a page (assuming the fields are 30 bytes and pointers are 8 bytes), we need only 1.000 index pages if the index is sparse. This is the first level index.

If we want to build a second level index, we need 10 pages for such second level index (since 100 key-pointer pairs fit in a page), and it is very likely that we have room for such 10 pages in the buffer.

Clustering sorted index: insertion and deletion

We leave to the students the task of thinking about algorithms for insertions and deletions (assuming alternative 2) in the case of clustering sorted index organization.

Non-clustering sorted index: two cases

Recall that a non-clustering sorted index (also called secondary sorted index in some texts) is an organization in which the data file is either unsorted (for example, a heap), or is a sorted file, but sorted on attributes different from the search key of the sorted index.

1. Non-clustering, primary (or unique) sorted index (i.e., on a relation key)

2. Non-clustering, secondary non-unique sorted index (i.e., on non-key attributes)

Non-clustering sorted index: first case

1. Non-clustering, primary (or unique) sorted index (i.e., on a relation key)

2. Non-clustering, secondary non-unique sorted index (i.e., on non-key attributes)

Non-clustering sorted index: on key

We start our analysis with the case of primary (or unique) index.

Recall that, if not otherwise stated, we assume to use alternative 2, and the difference between dense and sparse is as follows:

- Dense

every value of the search key that appears in the data file appears also in at least one data entry of the index (or, strongly dense: there is one data entry in the index file for every record in the data file)

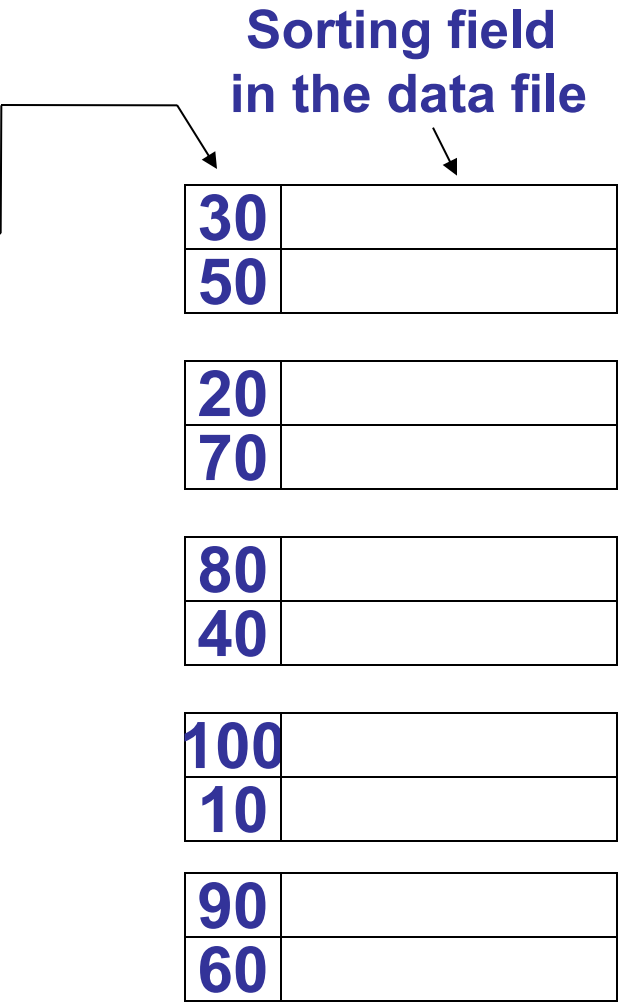
- Sparse

only some of the data records have a corresponding data entry in the index file. Often, there is one data entry per page in the data file.

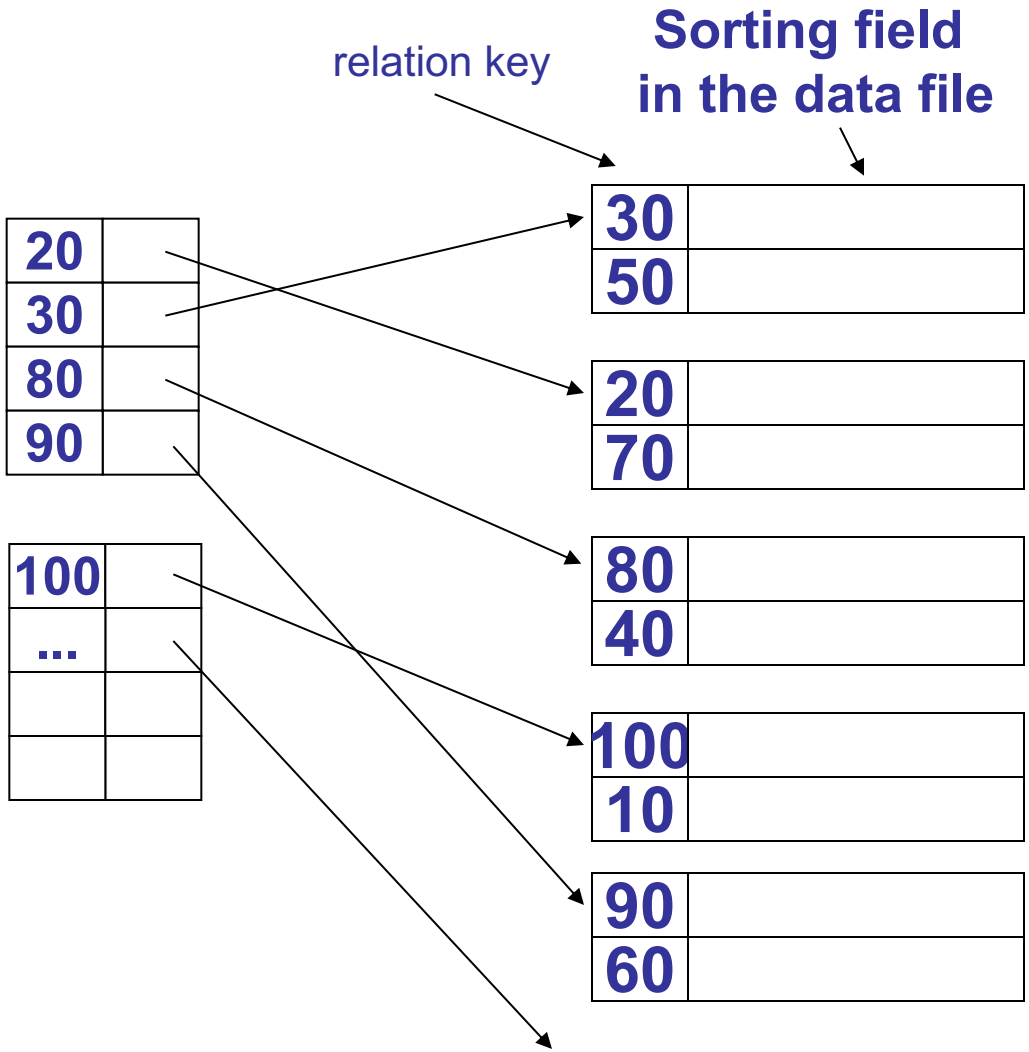
Non-clustering sorted index: on key

We build a sorted index on this relation key:

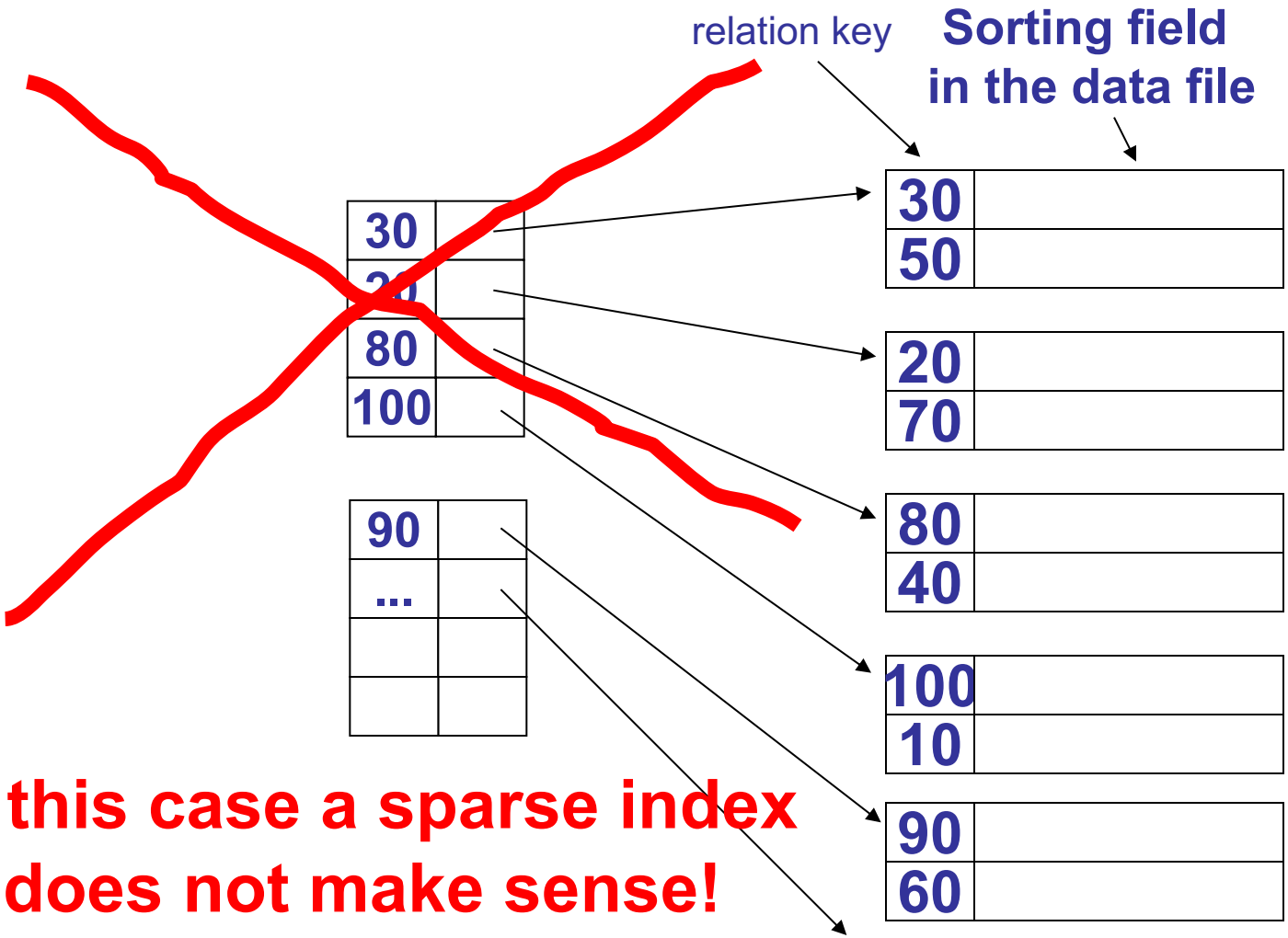
Since we are dealing with a primary key index, we do not have duplicate values in the index



Non-clustering sorted index: on key, sparse



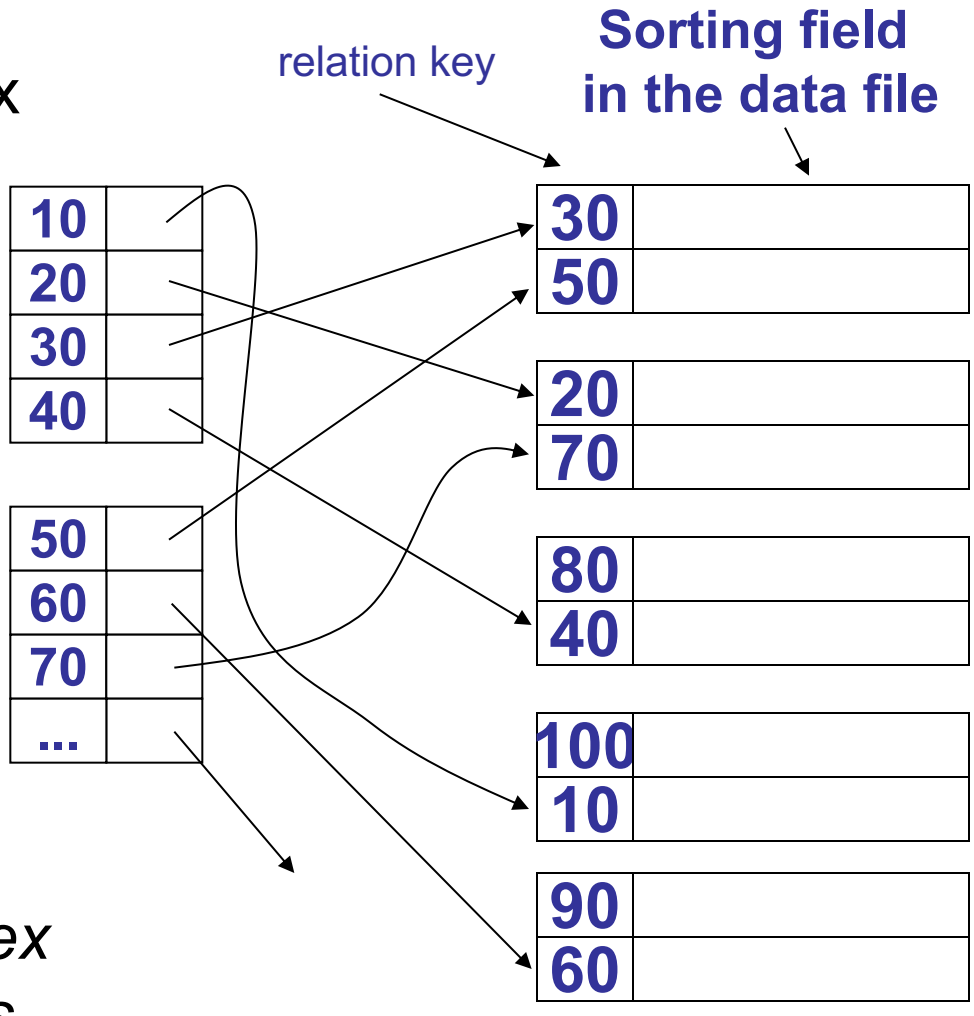
Non-clustering sorted index: on key, sparse



in this case a sparse index does not make sense!

Non-clustering sorted index: on key, dense

A non-clustering dense sorted index makes sense!



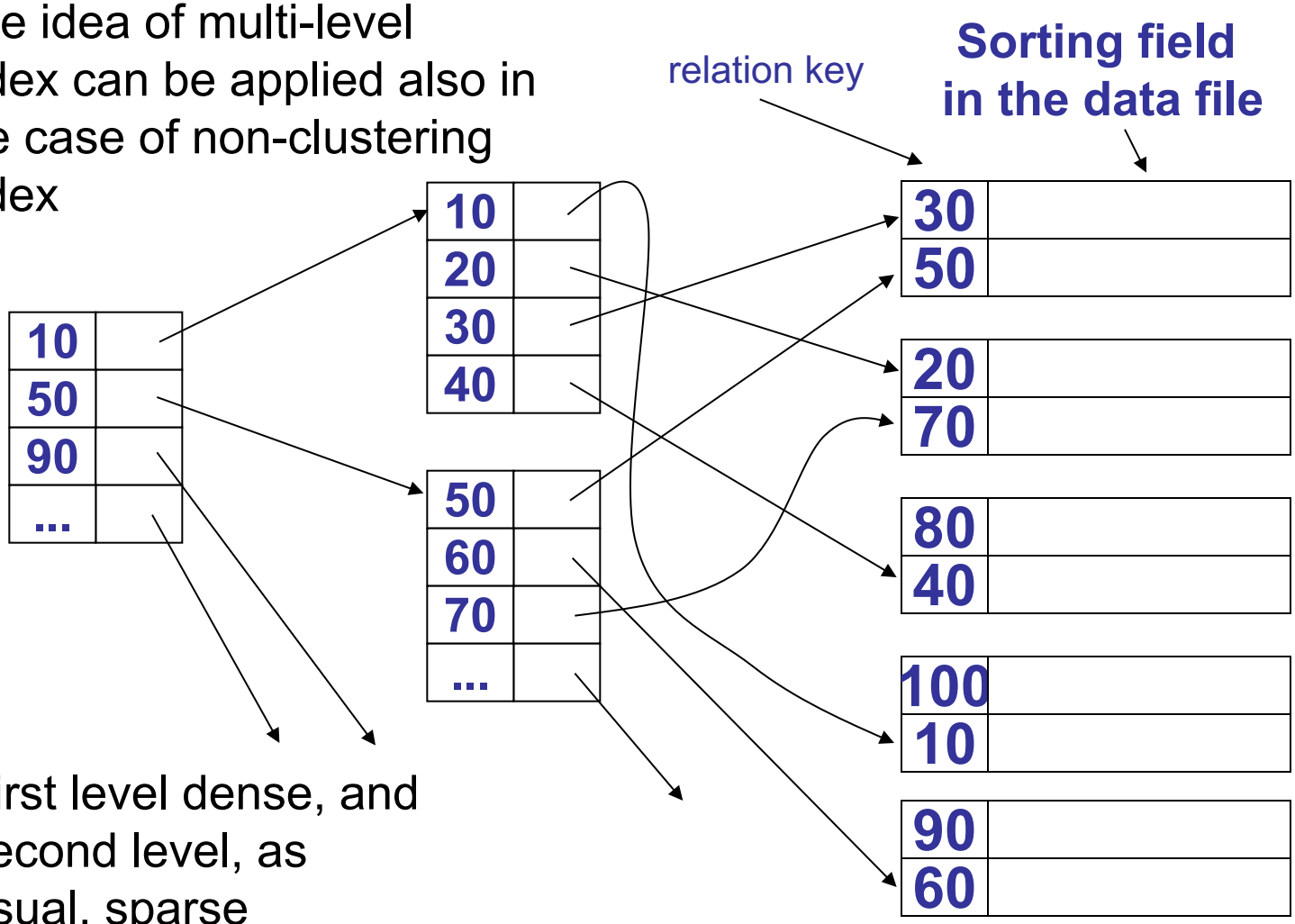
Note: pointers in a non-clustering index are record pointers

Non-clustering sorted index: important comment

- A non-clustering sorted index supports reasonably well the operation of equality search on the search key
- While we have seen that a clustering sorted index also supports the range selection (on the search key) operation (also called interval-based search), a non-clustering sorted index support well the range selection operation only in the case where we do not have to access to the data file (index-only access)
- Like in every non-clustering index organization, the range selection operation in the case where we have to access the data file, is much more problematic, because the number of page accesses to the data file might grow considerably.

Multiple level non-clustering sorted index

The idea of multi-level index can be applied also in the case of non-clustering index



First level dense, and second level, as usual, sparse

Non-clustering sorted index: non-key

1. Non-clustering, primary (or unique) sorted index (i.e., on a relation key)

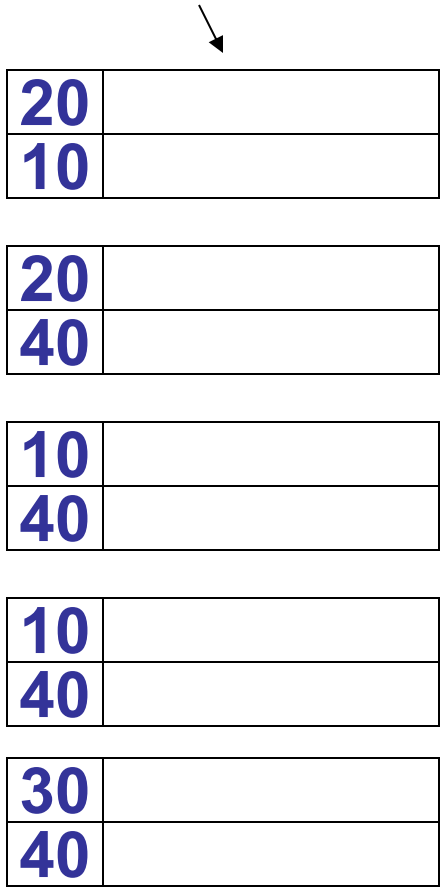
2. Non-clustering, secondary non-unique sorted index (i.e., on non-key attributes)

Non-clustering sorted index (with duplicates)

So far, we have assumed that the search key of the non-clustering index was a key in the data file, and therefore there were no duplicates in the data file in the search key.

Now, we analyze the case of secondary index, i.e., the case where there may be duplicate values of the search key in our non-clustering index.

Sorting field in the data file

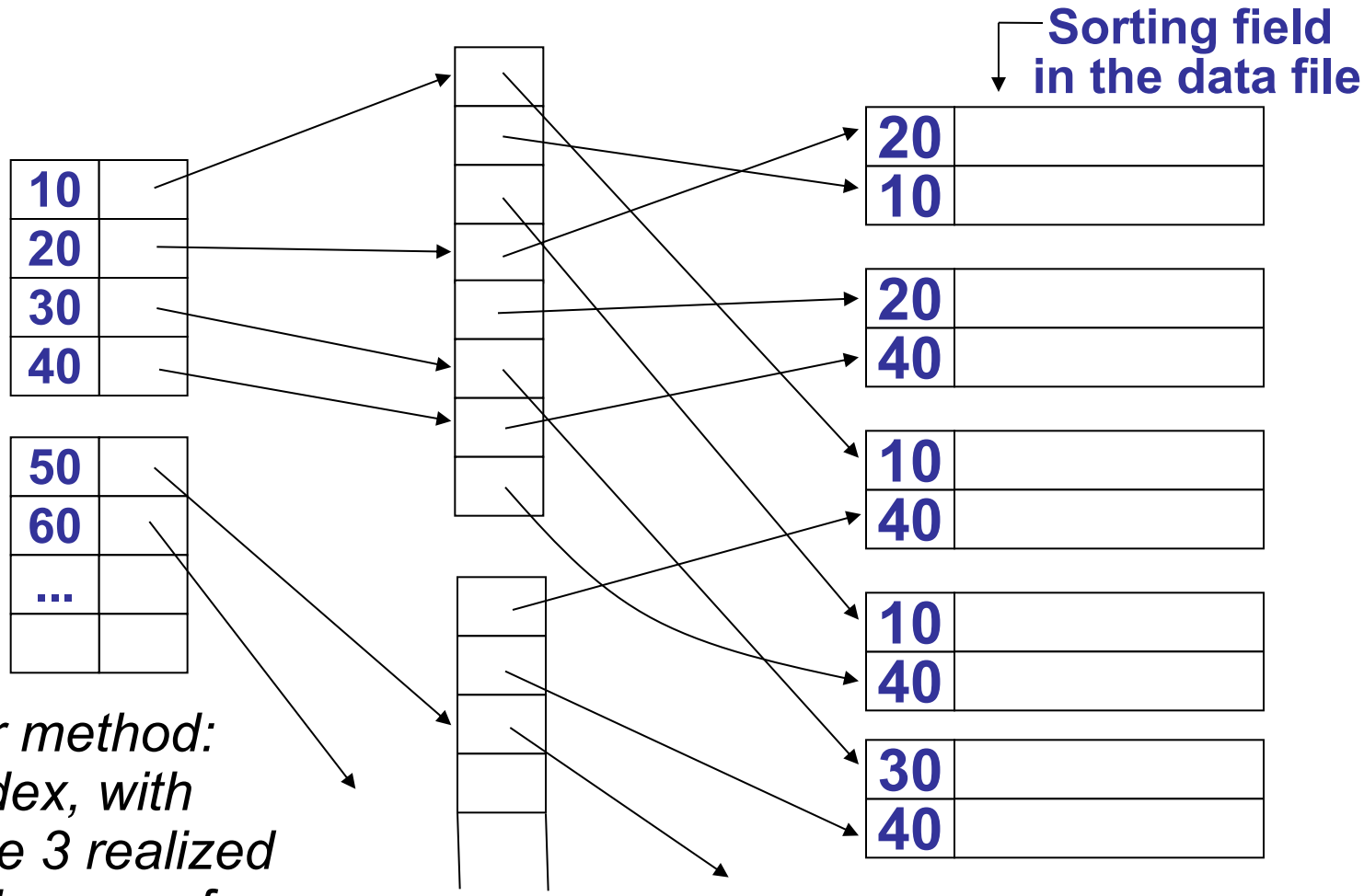


Non-clustering sorted index (with duplicates)

- From the picture in the previous slide, we can observe that the pointers in one index page can go to many different data pages, instead of one or a few consecutive pages (as in the case of clustering index).

- For example, to retrieve all the records with search key 10 in the picture of the previous slides, we not only may have to look at several index pages (this could happen in the case of clustering index too), but we are sent by their pointers to three different pages. Thus, using a non-clustering index may result in many more data page accesses than if we get the same number of records via a clustering index.

Non-clustering sorted index (with duplicates)



A popular method: dense index, with alternative 3 realized through the use of "buckets"

Why the idea of “bucket” is useful

The “buckets” organization (also called “inverted list”) in the context of a non-clustering index with duplicates are used to link the values of the search key with all the data records holding such values, thus avoiding duplicating the values of the search key in the index.

But buckets are also useful during query answering. Let us consider an example:

Indexes

Name: primary

Dept: secondary

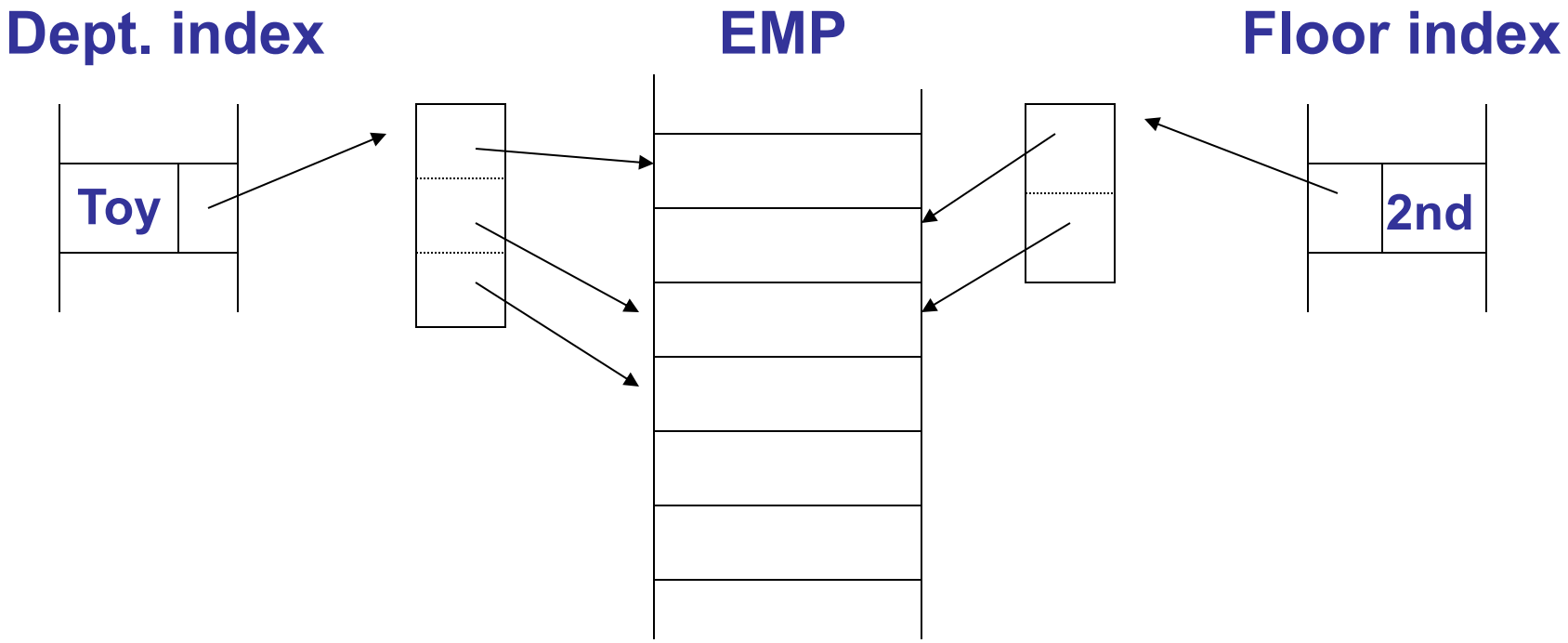
Floor: secondary

Data records

EMP(name,dept,floor,...)

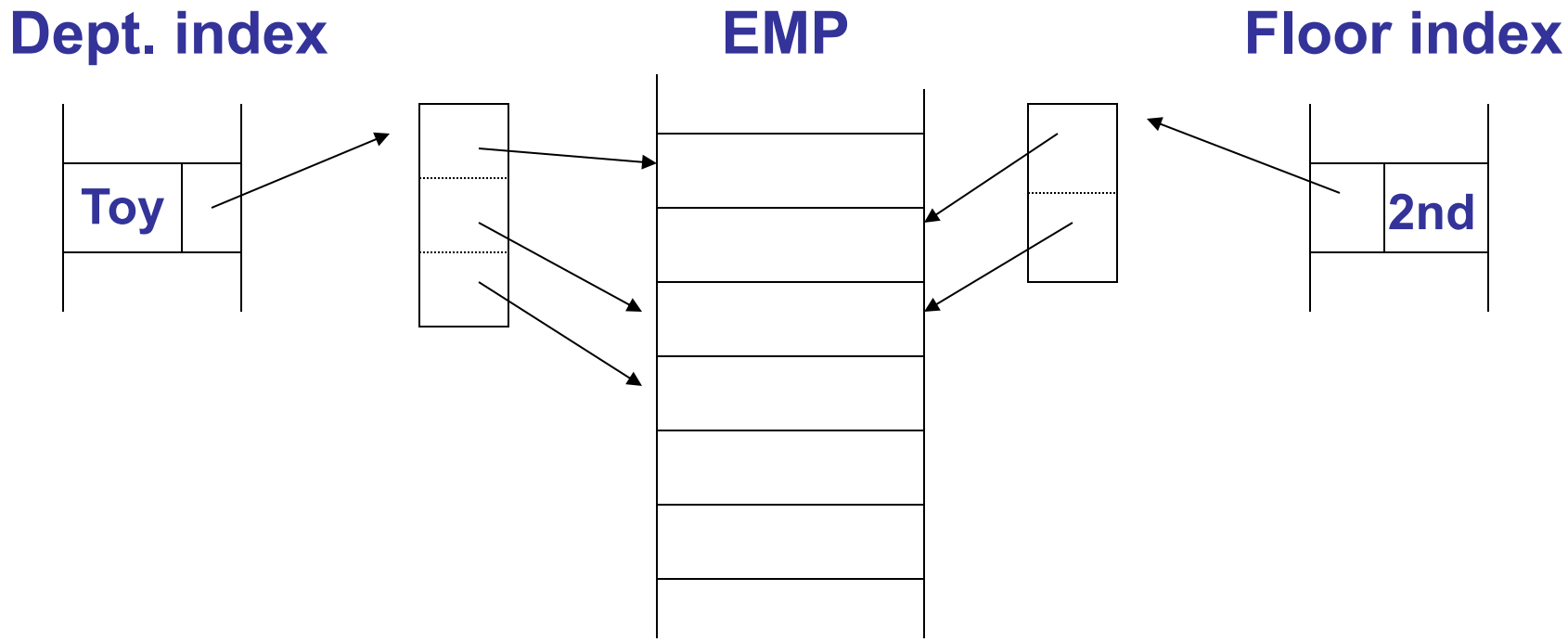
Why the idea of “bucket” is useful

Query: Get employees in the “Toy” department working in the second floor.



Why the idea of “bucket” is useful

Query: Get employees in the “Toy” department working in the second floor.



We answer the query by computing the intersection between the set of pointers in the “Toy” bucket and the set of pointers in the “2nd” bucket!

This idea used in text information retrieval

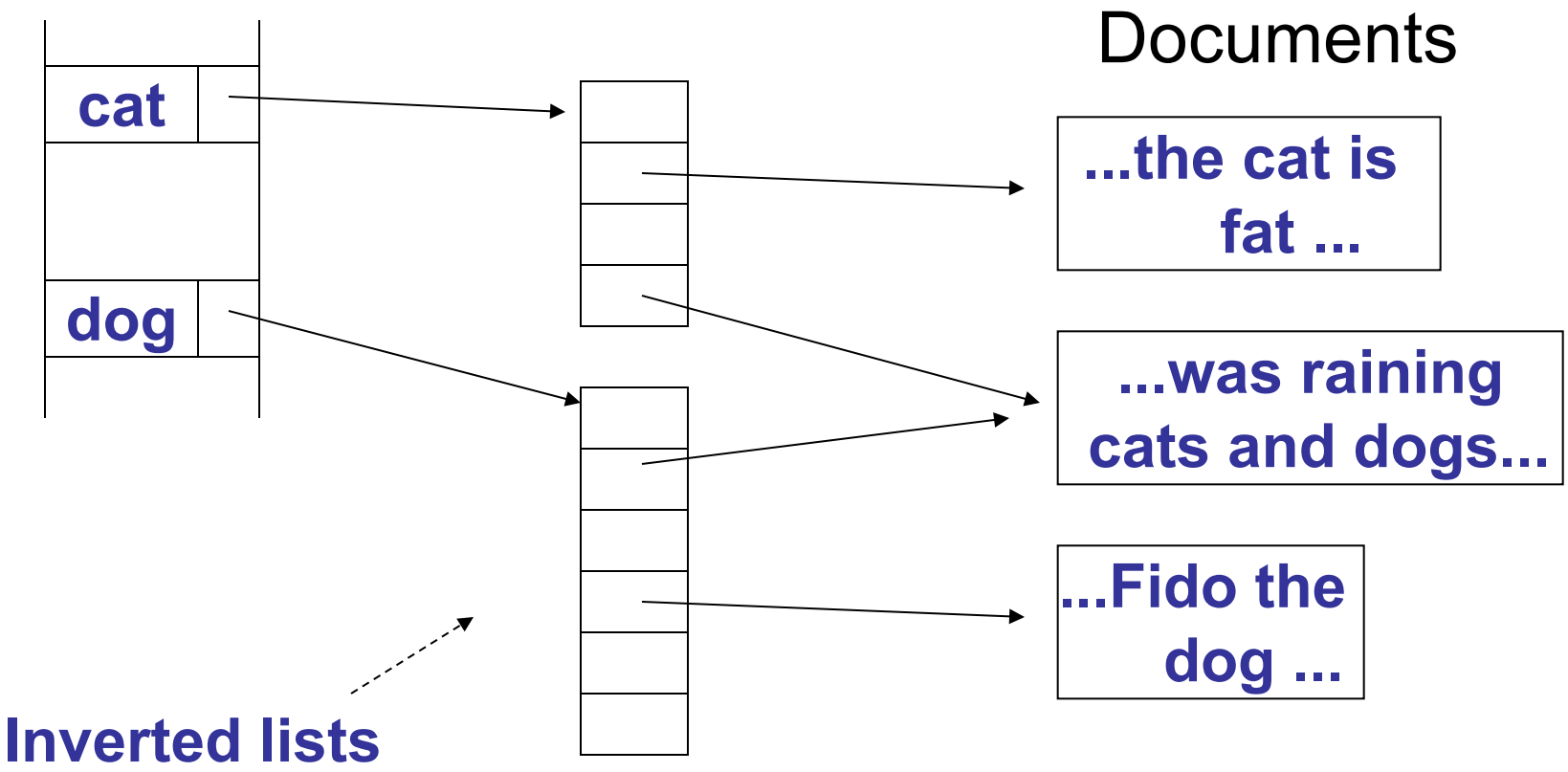
Documents

...the cat is fat ...

...was raining cats and dogs...

...Fido the dog ...

This idea used in “Information Retrieval”



Typical “Information Retrieval” queries

- Find articles with “cat” and “dog”
- Find articles with “cat” or “dog”
- Find articles with “cat” and not “dog”

These queries can be answered by means of operations on the sets of pointers in the inverted lists.

Another application of unclustered sorted index

- We said at the beginning of this part of the course that one DB file is tipically used for just one relation. This rule has exceptions. In particular, there are special cases where the DBMS stores two relations in the same DB file, that is called clustered file (not to be confused with the notion of clustered, or clustering, index), in the sense that the file “clusters” two relations together.

- We now discuss the usefulness of non-clustering sorted indexes in indexing relations stored in a clustered file.

Another application of unclustered sorted index

Consider the following example. We have a relation schema constituted by

- Movie(title, year, length, studioname)
- Studio(name, address, president)

Attributes title and year form the key for Movie, and name is the key for Studio. Attribute studioname is a foreign key referencing Studio. Now suppose that a common query is query 1:

```
select title, year
from Movie
where studioname = ZZZ
```

To support this query reasonably, we can order the tuples of Movie by studioname and build a clustering secondary sorted index (with duplicates) on Movies.studioname.

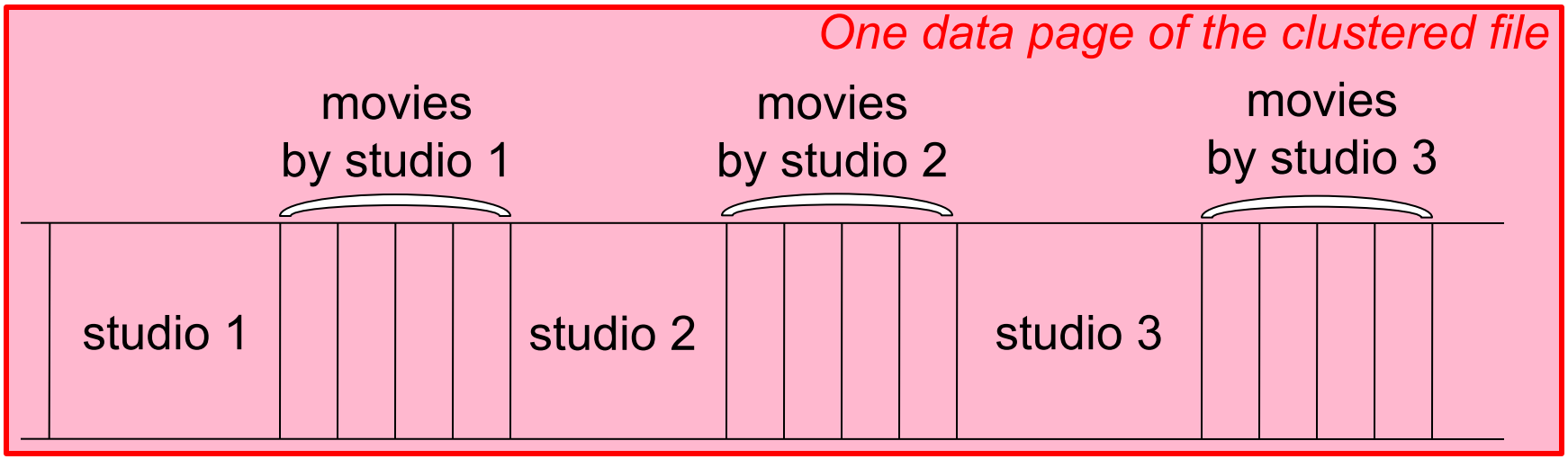
Another application of unclustered sorted index

Now suppose that another important query to execute is query 2:

```
select Studio.president
from Movie, Studio
where Movie.title = YYY and Movie.studioname=Studio.name
```

which requires a join between the two relations.

To support this query, we can decide to store the two relations in a clustered file, where we include in each Studio record all the Movie records of movies made by that studio (obviously, forgetting about the attribute studioname of all the movies), with no redundancy, since each movie is produced in one studio.



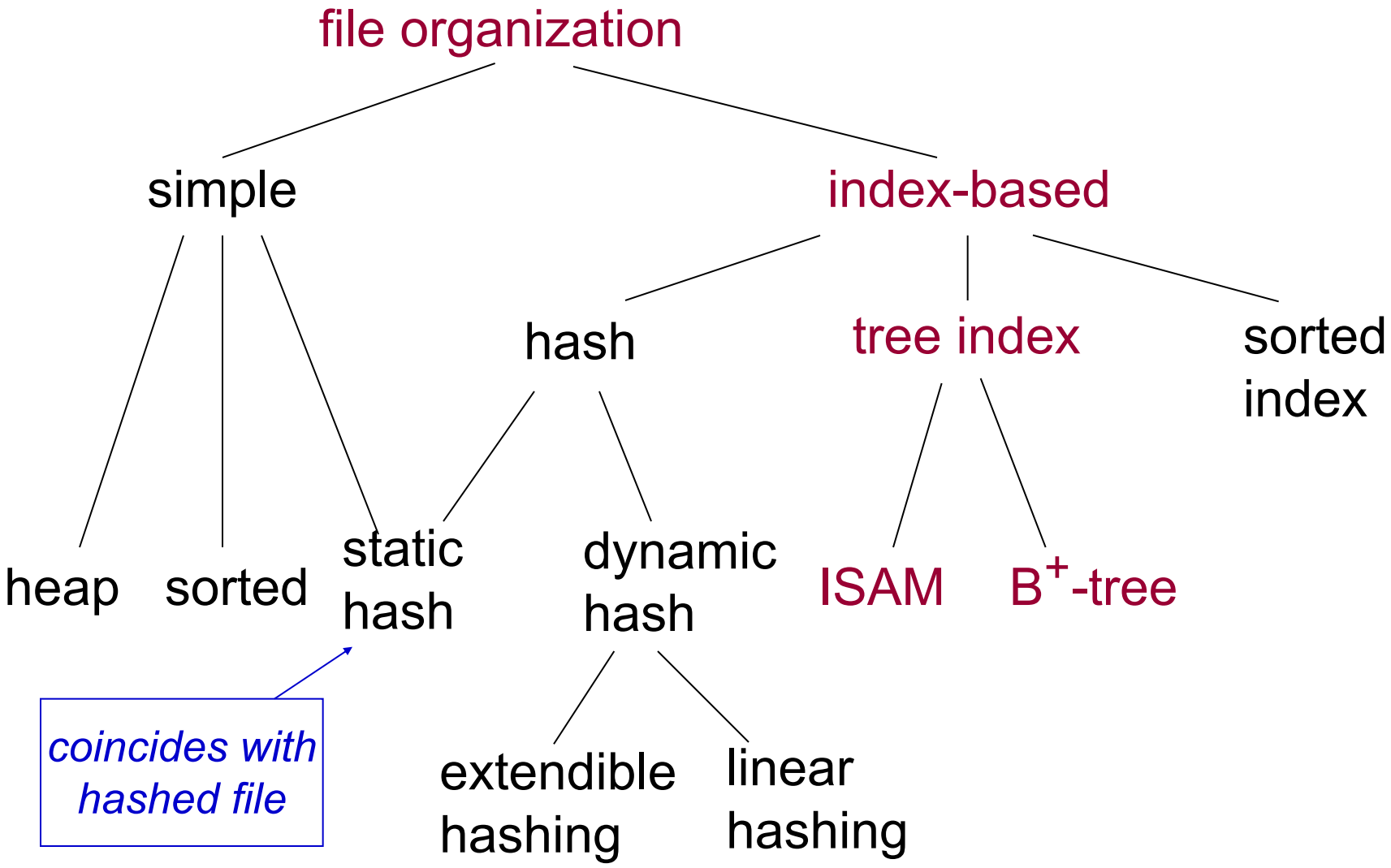
Another application of unclustered sorted index

If we want the president of the studio that made a particular movie (e.g., the movie with a given title YYY), we have a good chance of finding the record with the movie and the corresponding studio in the same page, saving at least one page access. Also, if we want the movies made by a certain studio, we will again tend to find the movies in the same page as the studio.

Obviously, in order to make the queries really efficient, we need to find the given movie or the given studio efficiently. For finding the studio with a given name efficiently, we can replace the index defined before with a clustering, primary sorted index on Studio.name (so as to support query 1). For finding the movies with a given title efficiently, we can then define a non-clustering secondary sorted index on Movie.title to find the movies with that title (so as to support query 2).

The price we pay is that some operations (e.g., scan of each relation) are more costly than with the usual representation with two files.

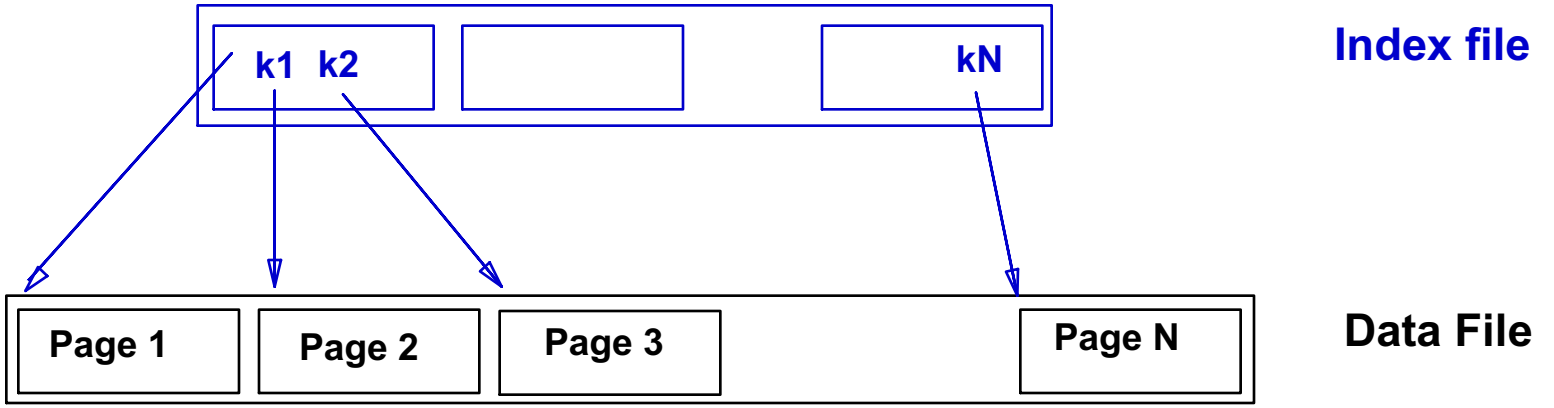
Tree-based index organization



Remember the idea of sorted index ...

We remind here the basic idea of the sorted index.

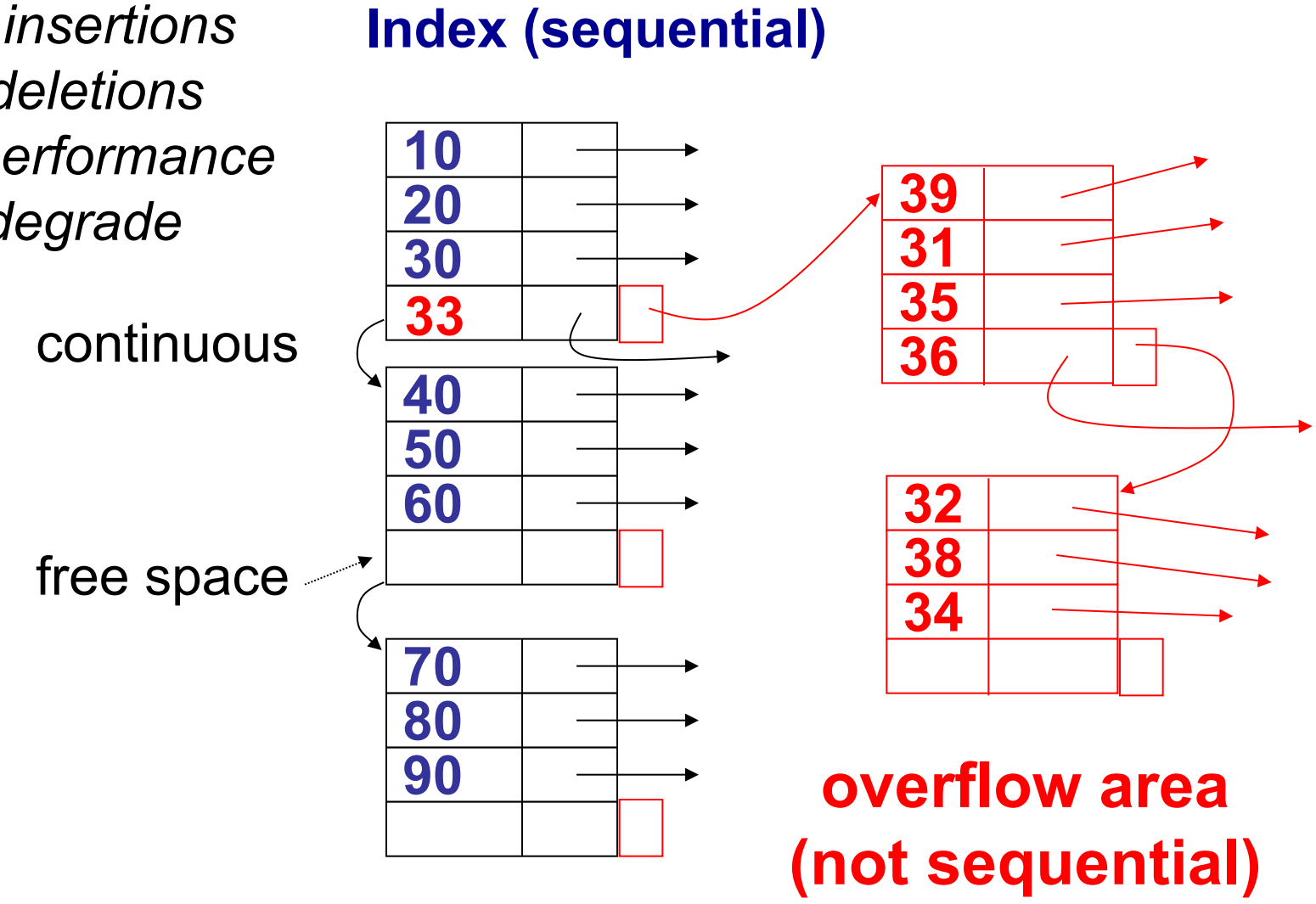
- Find all students with avg-grade > *27*
  - If students are ordered based on avg-grade, we can search through binary search
  - However, the cost of binary search may become high for large files
- Simple idea: create an auxiliary sorted file (index), which contains the values for the search key and pointers to records in the data file



The binary search can now be carried out on a smaller file (data entries are smaller than data records, and we can even think of a sparse index)

One problem with the sorted index

With insertions and deletions the performance can degrade



Tree index: general characteristics

- If we iterate as much as we can the process behind the simple idea described above (until the auxiliary structure produced fits in one page), we obtain a tree index, with also the possibility of solving the problems of perfomance degradation that we face with insertions and deletions (see later)
- In a tree index, the data entries are organized according to a tree structure, based on the value of the search key
- Searching means looking for the correct page (the page with the desired data entry), with the help of the tree, which is a hierachical data structure where
  - every node coincides with a page
  - pages with the data entries are the leaves of the tree
  - any search starts from the root and ends on a leaf (therefore it is important to minimize the height of the tree)
  - the links between nodes corresponds to pointers between pages
- In the following slides, when we talk about index, we mean tree index, and we distinguish between ISAM, and B^+-tree index

Tree index: general characteristics

The typical structure of an intermediate node (including the root) is as follows:

- Sequence of m+1 pointers P separated by different values K ordered according to the search key
- Pointer P_{i-1} on the left of value K_i ($1 \leq i \leq m$) points to the subtree containing only data entries with values that are less than K_i
- Pointer P_i on the right of value K_i points to the subtree containing only data entries with values that are greater than or equal to K_i (and, obviously less than K_{i+1},if K_{i+1} exists)

Note that this implies that $K_1 \leq K_2 \leq ... \leq K_m$.

Tree index: two types

- ISAM

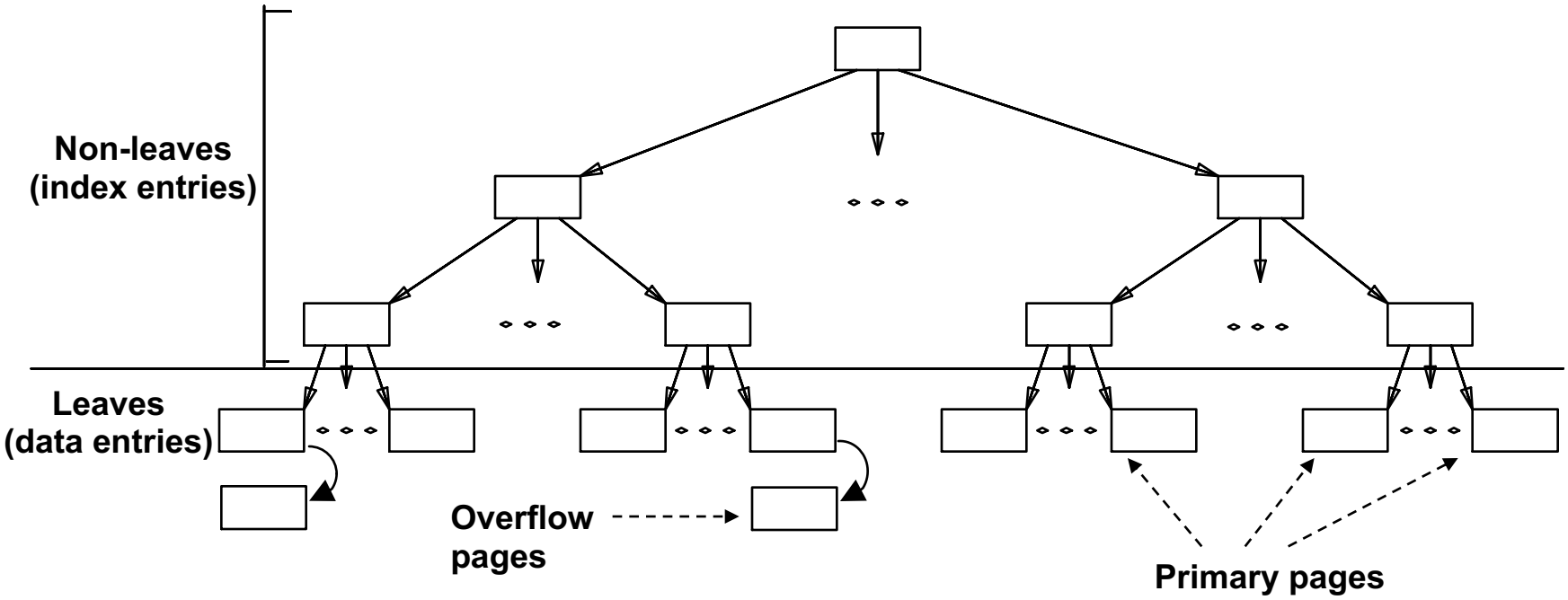
  used when the relation (or, the index) is static (essentially no insertion or deletion on the tree)
- B+-tree

  effective in dynamic situations (i.e., with insertions and deletions)

In what follows, we assume that there are no duplicates of the search key values in the index. All the observations can be generalized to the case with duplicates.

ISAM

The name derives from Indexed Sequential Access Method



The leaves contain the data entries, and they can be scanned sequentially. The structure is static: although updates are possible in the leaves, we have no update on the intermediate nodes of the tree!

Comments on ISAM

- An ISAM is a balanced tree -- i.e., the path from the root to a leaf has the same length for all leaves (except for the last one)
- The **height** of a balanced tree is the length of the path from root to leaf
- In ISAM, every non-leaf nodes have the same number of children; such number is called the **fan-out** of the tree.
- If every non-leaf node has F children, a tree of height h has F^h leaf pages.
- In practice, F is at least 100, so that a tree of height 4 contains 100 million leaf pages
- We will see that this implies that we can find the page we want using 4 I/Os (or 3, if the root is in the buffer). This has to be contrasted with the fact that a binary search of the same file would take $\log_2 100.000.000$ (>25) I/Os.

Comments on ISAM

- *Creation of the index*: the leaves are allocated sequentially, and the intermediate nodes are then created
- *Search*: We start from the root and compare the key we are looking for with the keys in the tree, until we arrive at a leaf. The cost is

$$\log_F N$$

or $(1 + \log_F N)$ under the assumption that the root is not in the buffer.

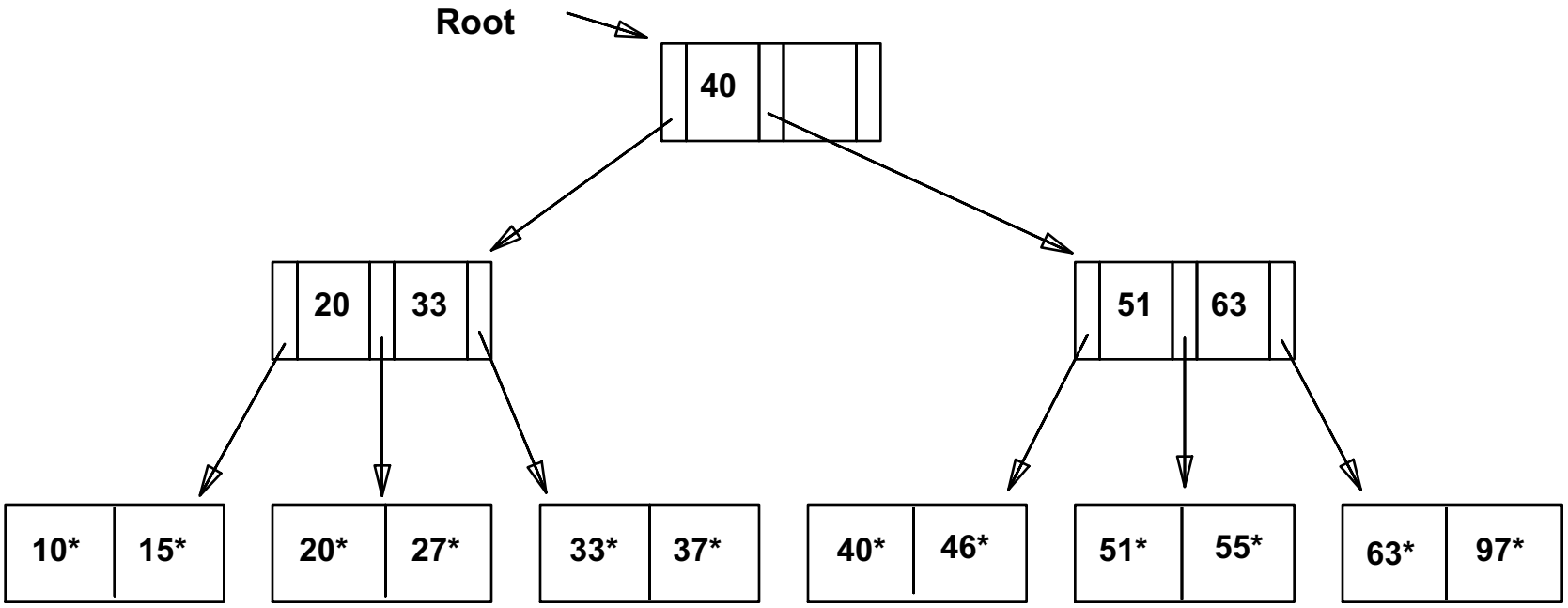
where F is the fan-out of the tree, and N is the number of leaves (typically, the value of N depends on several factors, including the size of the data file, and whether the index is dense or sparse)

- *Insert*: We find the correct leaf where to insert, allocating an overflow page, if needed; we then insert the data record in the data file
- *Delete*: We find and delete from the leaves; if the page is an overflow page, and is empty, then we deallocate the page; in any case, we delete the correct data record from the data file

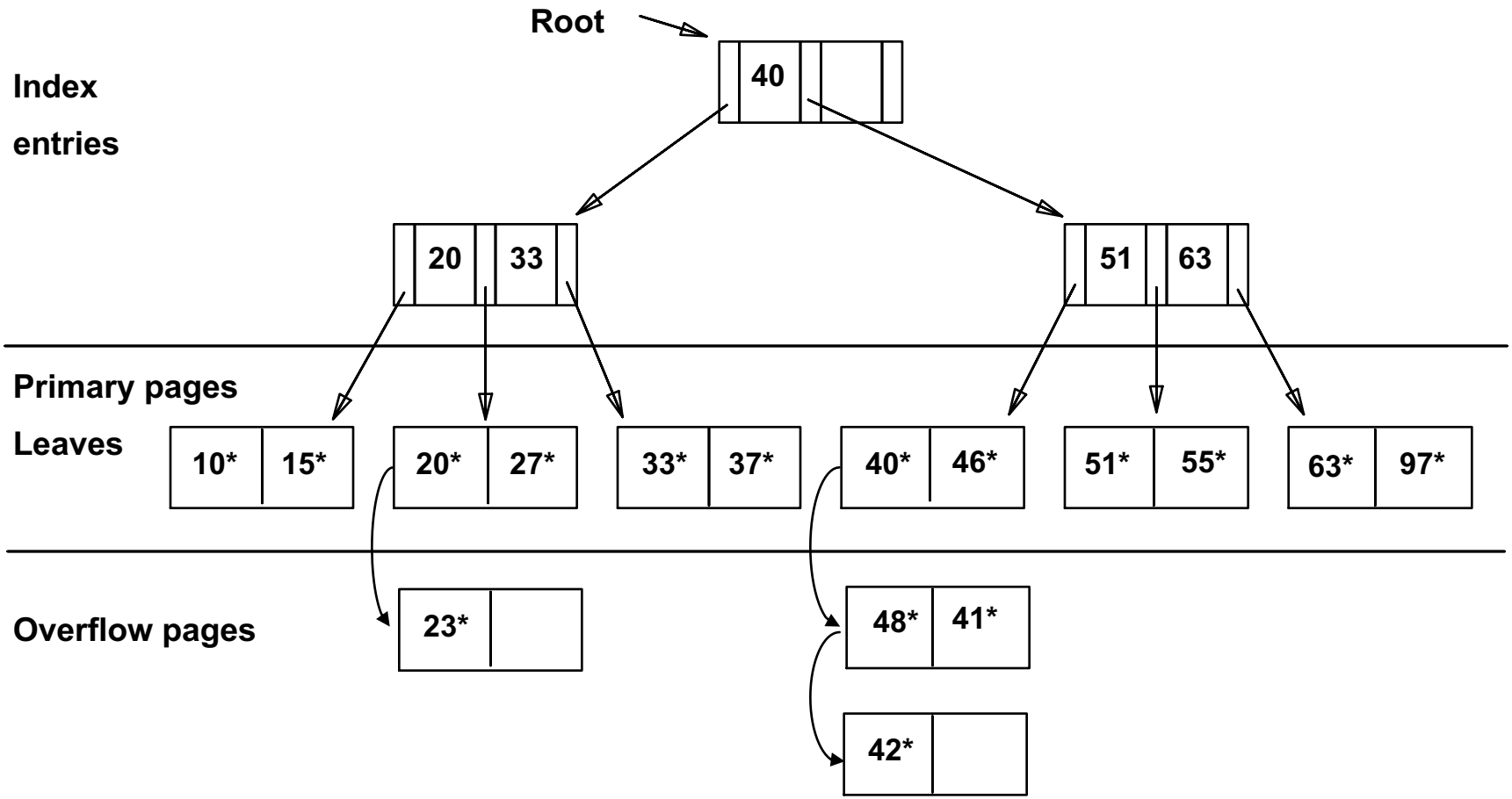
Static structure: *insert/delete are rare, and involve only leaves*

ISAM: example

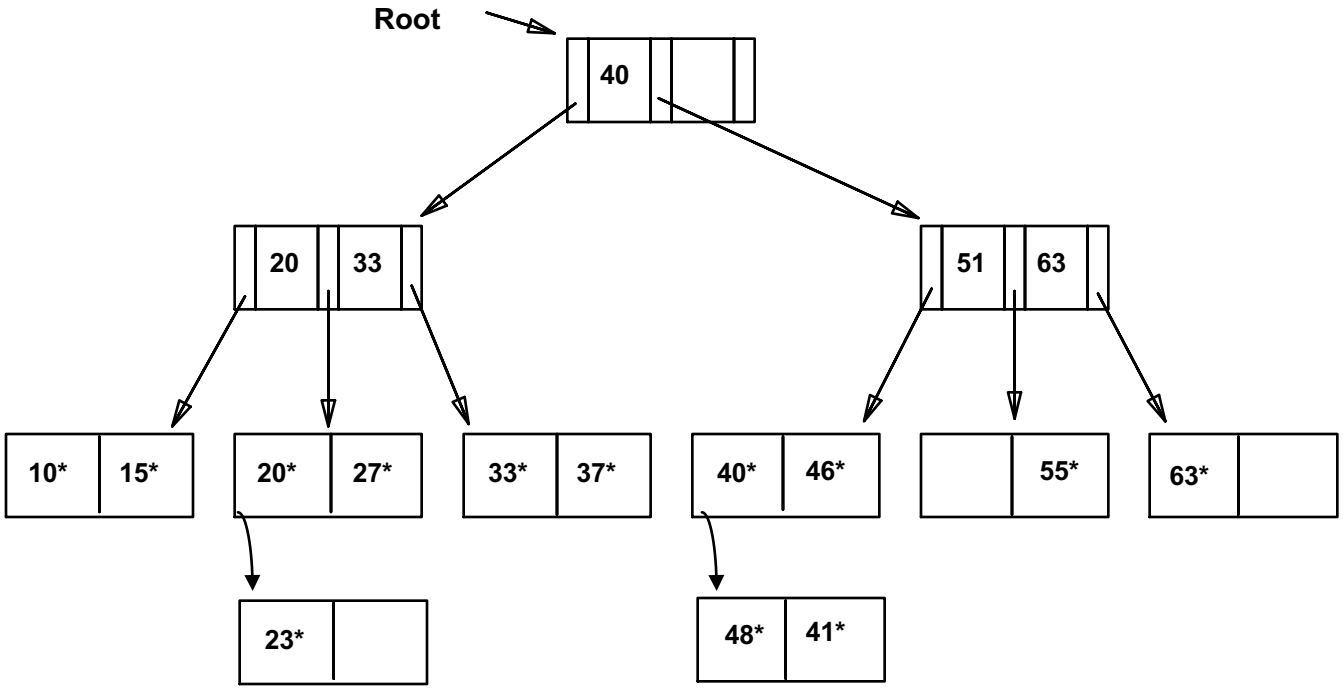
We assume that every node contains 2 entries (three pointers), except the root that may contain less than 2 entries)



Insertion of 23*, 48*, 41*, 42* ...



Deletion of 42*, 51*, 97*



Note that 51* appears in the index entries, but not in the leaves (except for the leaves, we cannot change the tree!)

B^+-tree index

- A B^+-tree is again a *balanced* tree, where the length of the path from the root to a leaf is the same for all leaves
- B^+-trees overcome the limitations/problems that ISAM has with insertion/deletion
- If each page has space for **d** search key values and **d**+1 pointers, then **d** is called the *rank* of the tree
- Every node n_i contains m_i data entries (search key value + pointer), with (**d**+1)/2 <= m_i <= **d**. The only exception is the root, that may have less index entries (at least one)
- The leaves are the pages with the data entries, and are linked through a list based on the order on the search key

  → Such list is useful for "range" queries over the search key values:
  - we look for the first value in the range, and we access the "correct" leaf L
  - we scan the list from the leaf L to the leaf with the last value in the range

Comments on B$^+$-tree

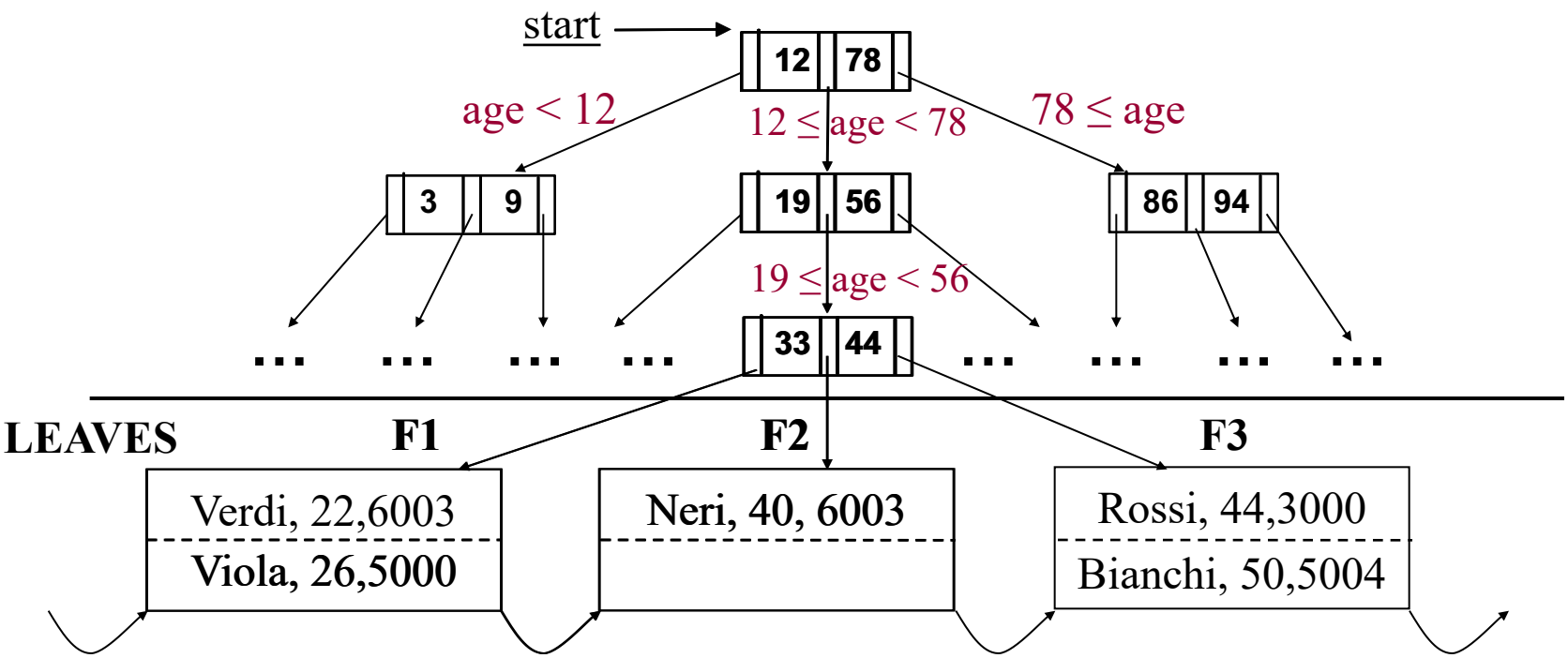
- We remind the reader that, for trees where the various non-leaf nodes have the same number of children, such number is called the **fan-out** of the tree. Also, if every node has n children, a tree of height h has n^h leaf pages. In other words, if the tree has M leaves, then the height h is $\log_n M$.
- If different non-leaf nodes may have different numbers of children, then using the average value F for the number of children of non-leaf nodes, we get F^h as a good approximation to the number of leaf pages (where h is the height), and, knowing that there are M leaves, we get $\log_F M$ as a good approximation of the height h. Such average value is often considered to be (d+d/2)/2 = 3d / 4.

Search through B$^+$-tree: example

We look for data entries with $24 < age \leq 44$

→ We search for the leaf with the first value in the range

→ We reach F1: we start a scan from F1 until F3 (where we find the first record with the first value outside the range)



Search through B^+-tree: observations

- The number of page accesses needed in a search for equality operation (assuming the search key to be the primary key of the relation and the root in the buffer) is at most the height of the tree (F is the fan-out of the tree, which is the average number of children per node):

  $\log_F N$ (where N is the number of leaves)

- The aim is to have F as large as possible (note that F depends on the size of the block):

  →Typically, the fan-out is at least 100, and by default we will assume that is exactly 100; we already noticed that with F=100, and 1.000.000 pages, the cost of the search is 4 (or 3, if the root is in the buffer)

  →Majority of pages occupied by the leaves

Search through B^+-tree: observations

- B^+-trees (in particular, when they realize a clustering index) are the ideal method for efficiently accessing data on the basis of a range
- They are also very effective (but no ideal) for accessing data on the basis of an equality condition
- We will now address the issue of insertions/deletions in a B^+-tree

Insertion in a B^+-tree

We only deal with insertion in the index (insertion in the data file is orthogonal)

Recursive algorithm

- We search for the appropriate leaf, and put the new key there, if there is space
- If there is no room, we split the leaf into two, and divide into equal parts the keys between the two new nodes
- After splitting, there is a new pointer to insert at the higher level; do that recursively
- If we try to insert into the root, and there is no room, we split the root into two nodes and create the new root at higher level, which has the two nodes resulting from the split as its children.

Insertion in a B^+-tree

Splitting a leaf node

- Suppose that N is a leaf node with n keys (which is the maximum allowed) and we want to insert the (n+1)-th key K
- Let S be the new (sorted) set of key values (i.e., the set of key values in N plus K)
- We create a new node M as a "right" sibling of N, and the first (n+1)/2 key-pointer pairs in S remain in N, and the other key-pointer pairs go to M
- The value in the middle in the order (the minimum value in the "right" sibling M) among the sorted values in S go to the higher level together with the appropriate pointer

Insertion in a B^+-tree

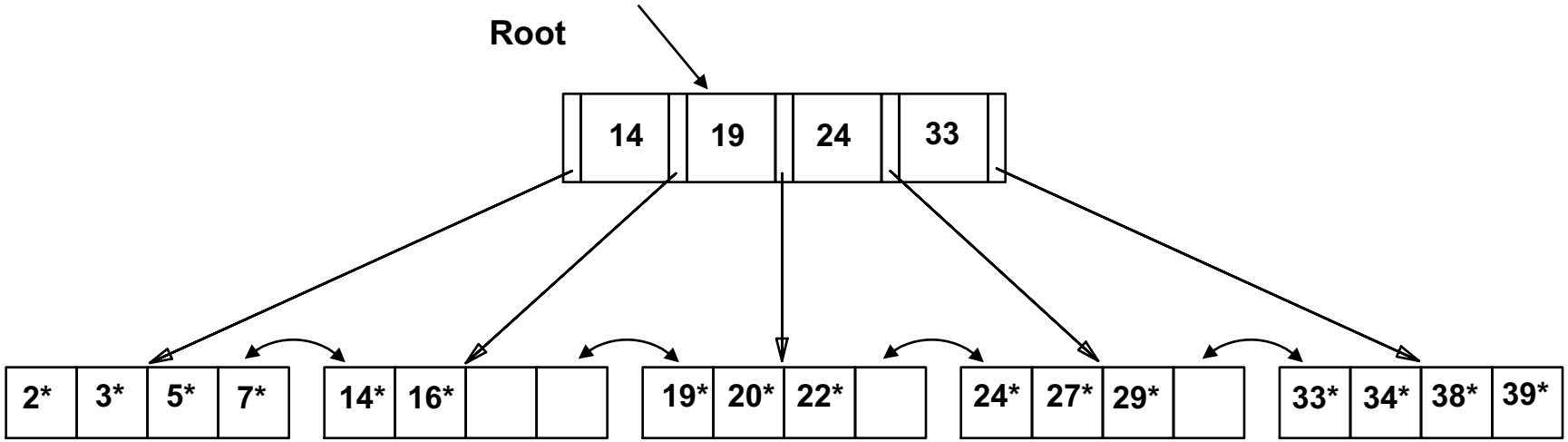
Splitting a non-leaf node

Suppose that N is a non-leaf node with n keys and n+1 pointers, suppose that another pointer arrives because of a split in the lowest level, and suppose that (n+1) was the maximum value of pointers allowed in the node.

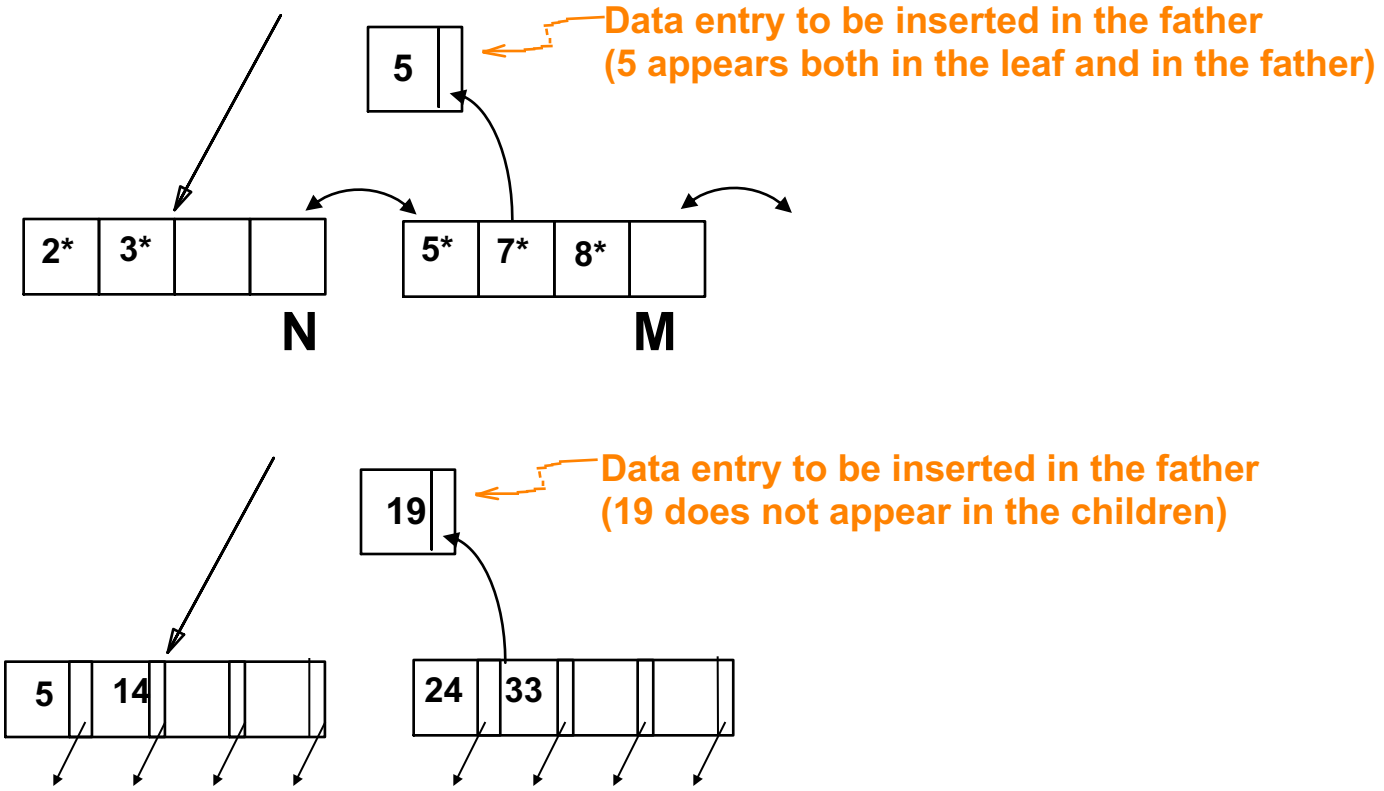
- We leave the first (n+2)/2 pointers in N, in sorted order, and move the remaining (n+2)/2 pointers to a new node M, sibling of N
- The first n/2 keys stay in N, and the last n/2 keys move to M. There is one key in the middle left over that goes with neither N nor M. The leftover key K is the closest value that is equal or smaller to the smallest key reachable in the tree via the first of the M's children. In other words, K will be sent to the parent of N and M and will be used to divide searches between those two nodes.

Insertion in a B$^+$-tree: example

Insertion of a data record with search key value 8

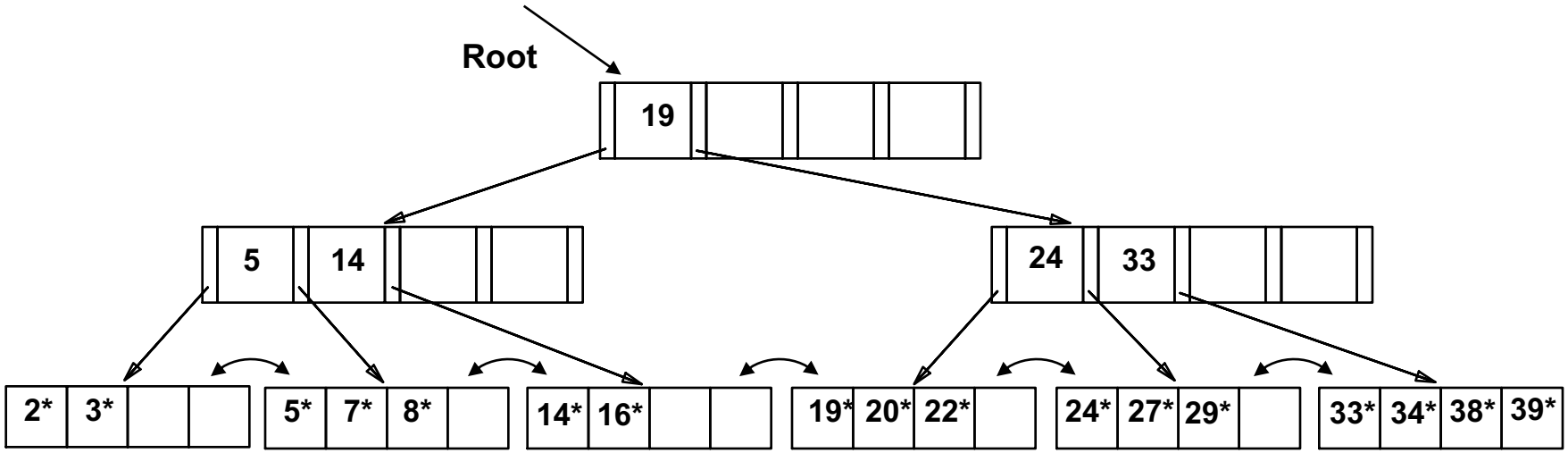


Insertion in a B$^+$-tree: example



Note: every node (except for the root) has a number of data entries greater than or equal to **d**/2, where **d** is the rank of the tree (here **d**=4)

Insertion in a B$^+$-tree: example



→The height of the tree has increased

Typically, the tree increases in breadth. The only case where the tree increases in depth is when we need to insert into a full root.

Deletion in a B^+-tree

We only deal with deletion in the index (deletion in the data file is orthogonal)

Deletion algorithm

If the node N after the deletion has still at least the minimum number of keys, then there is nothing to do

Otherwise, we need to do one the following things:

1. If one of the adjacent siblings of node N has more than the minimum number of keys, then one key-pointer pair can be moved to N (key redistribution). Possibly, the keys at the parent of N must be adjusted: for instance, if the right sibling of N, say node M, provides and extra key and pointer, then it must be the smallest key that is moved from M to N. At the parent of N and M, there is a key that represents the smallest key accessible via M: such key must be changed!

Deletion in a B$^+$-tree

2. If neither of adjacent nodes of N can provide an extra key for N, then we choose one of them, and "merge" it with N (this operation is called coalesce), because together they have no more keys and pointers than are allowed in a single node. After merging, we need to adjust the keys at the parent, and then delete a key and a pointer at the parent. If the parent is full enough, we are done, otherwise, we recursively apply the deletion algorithm at the parent. Note that this may result in lowering the depth of the tree.

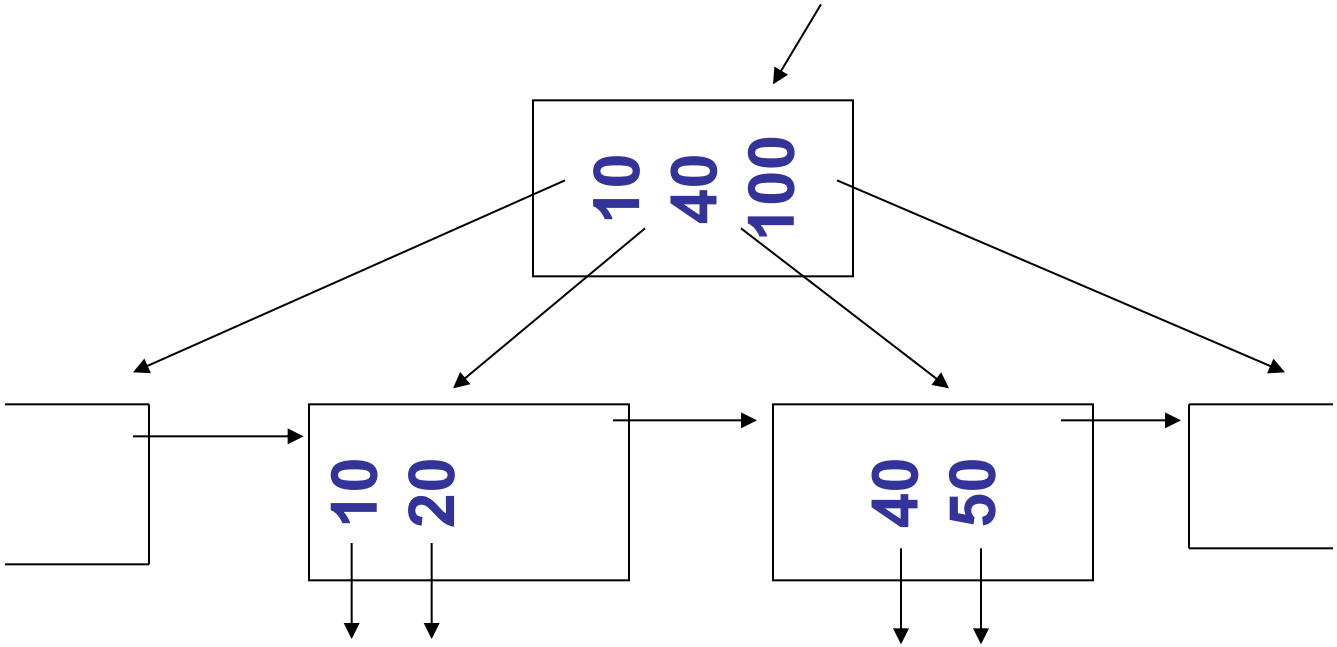
Note: sometimes, coalesce is not implemented, and deletion does nothing, keeping free space in the leaves for future insertions.

Deletion in a B$^+$-tree: examples

n=4

Coalesce with sibling

– Delete 50

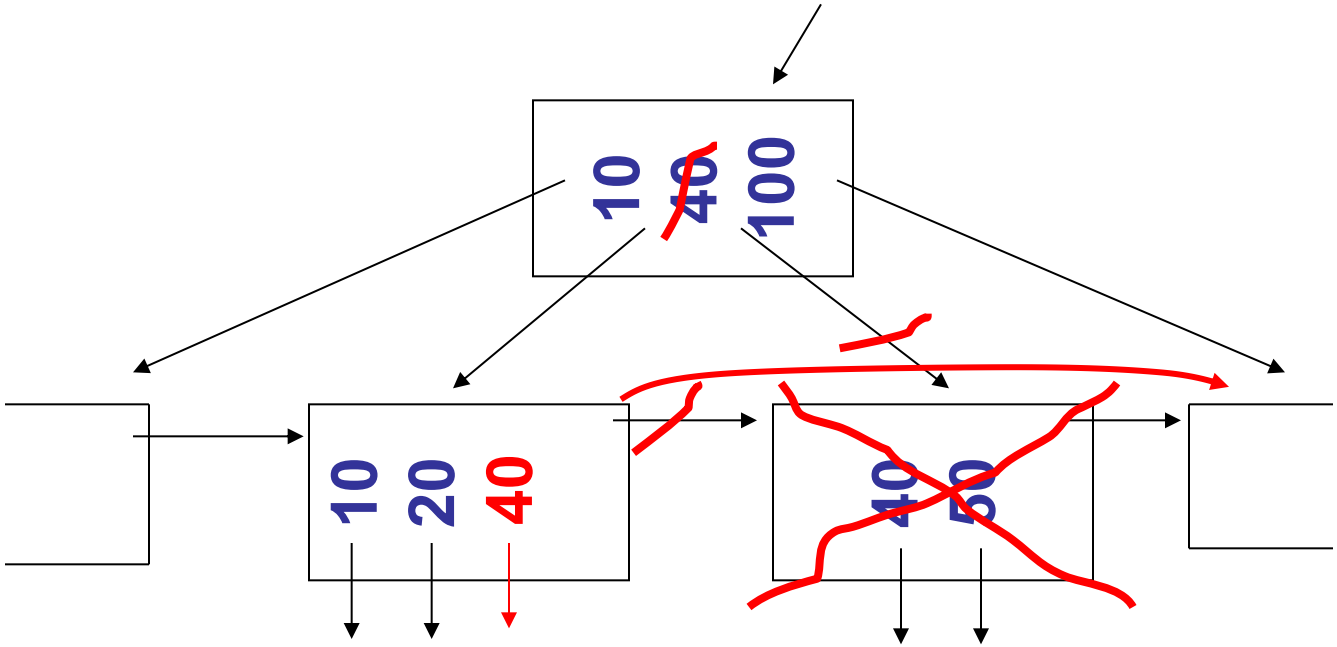


Deletion in a B$^+$-tree: examples

(b) Leaf-coalesce with sibling

– Delete 50

n=4

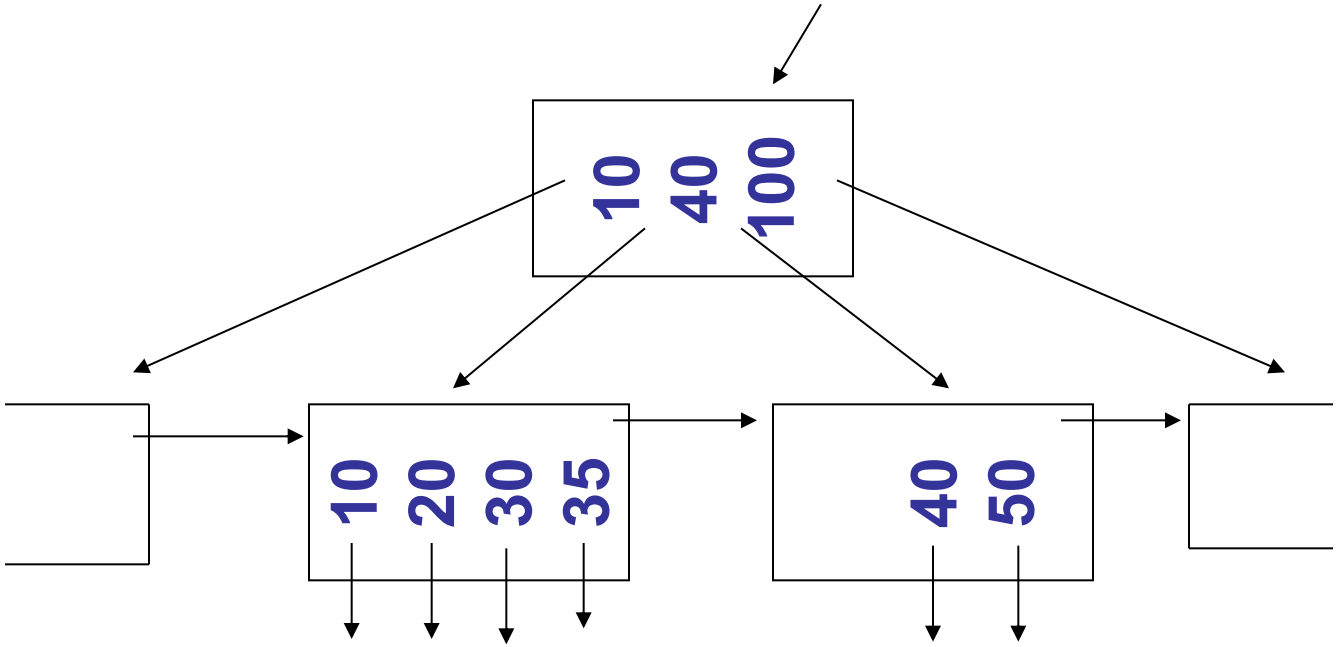


Deletion in a B$^+$-tree: examples

n=4

(c) Redistribute keys

– Delete 50

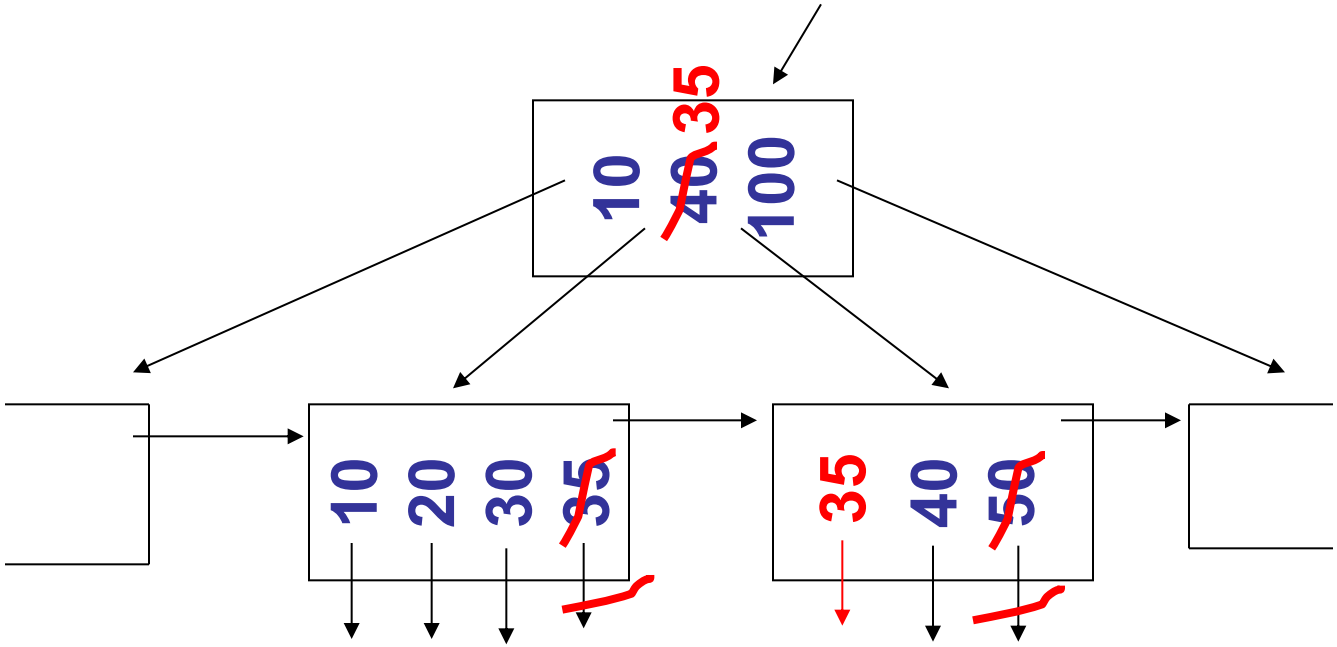


Deletion in a B$^+$-tree: examples

(c) Redistribute keys

– Delete 50

n=4

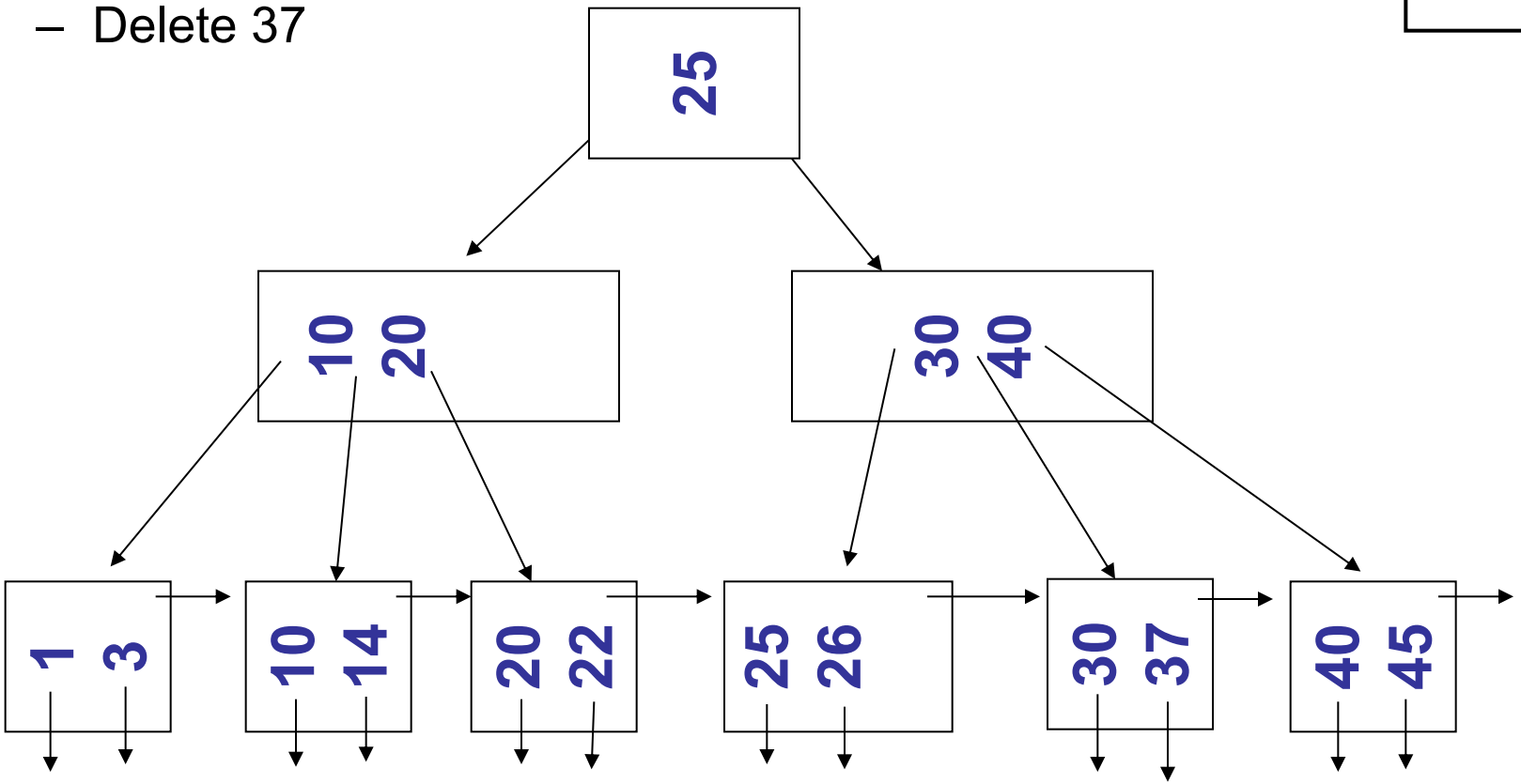


Deletion in a B$^+$-tree: examples

n=4

(d) Non-leaf coalesce

– Delete 37

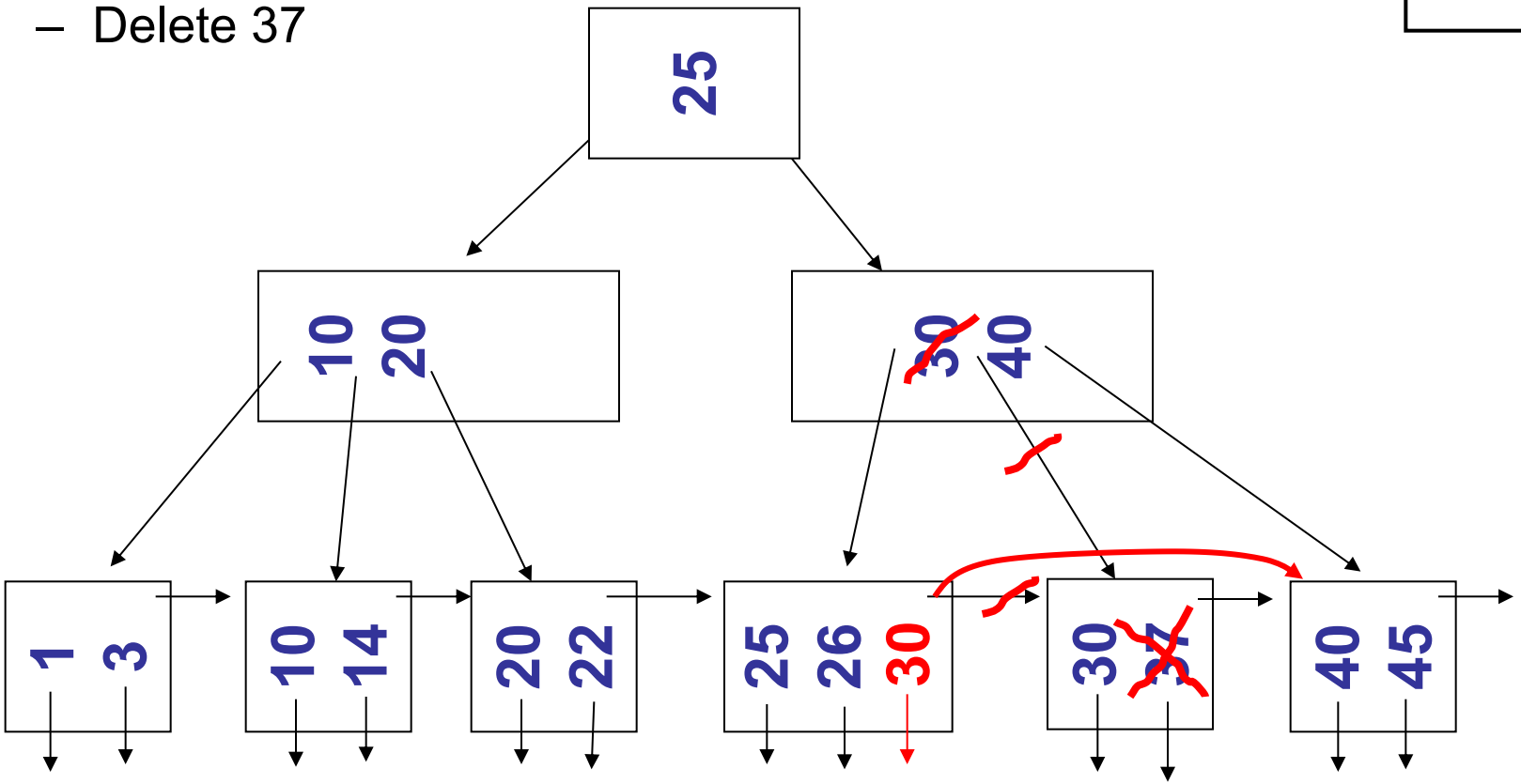


Deletion in a B$^+$-tree: examples

n=4

(d) Non-leaf coalesce

– Delete 37

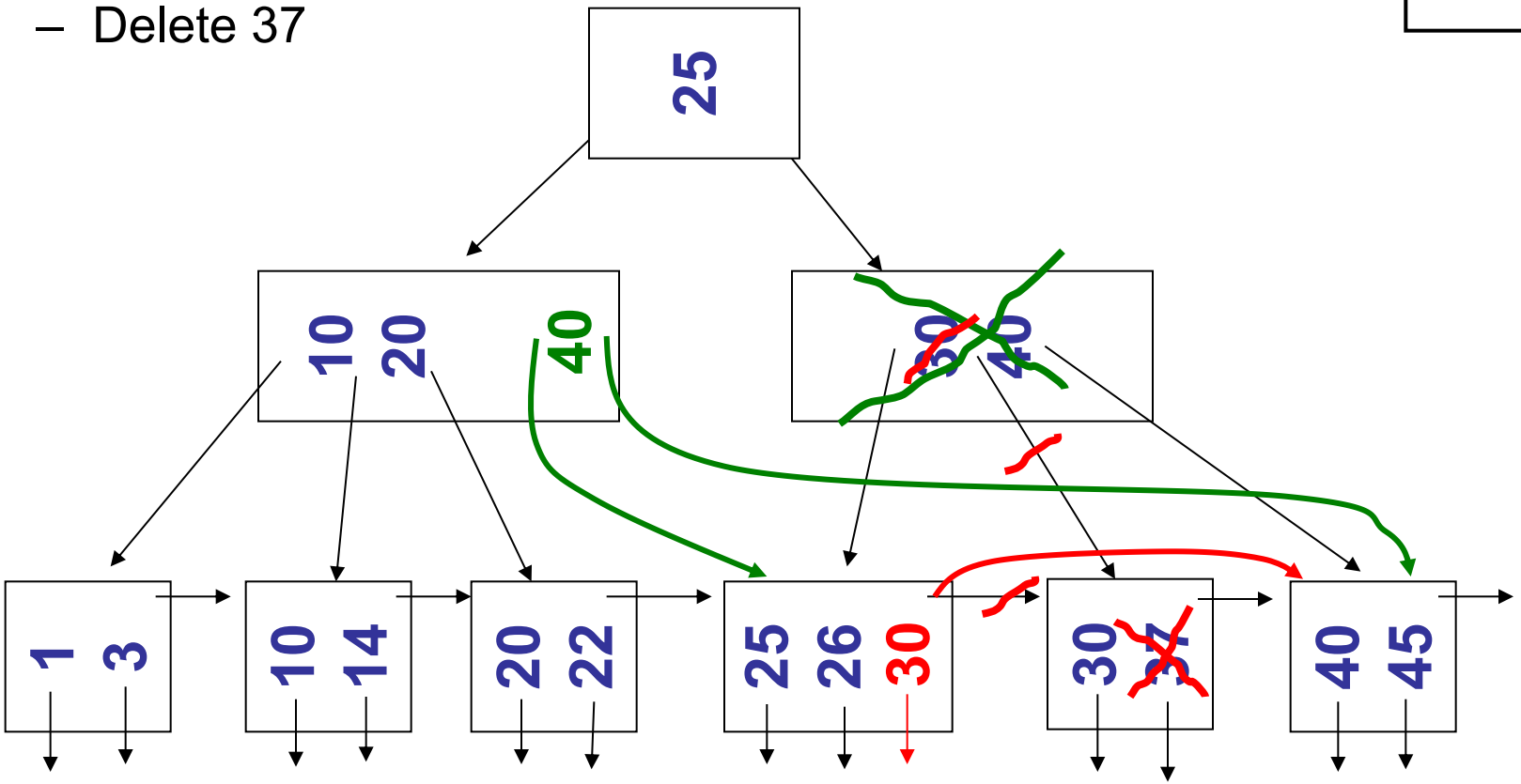


Deletion in a B$^+$-tree: examples

n=4

(d) Non-leaf coalesce

– Delete 37

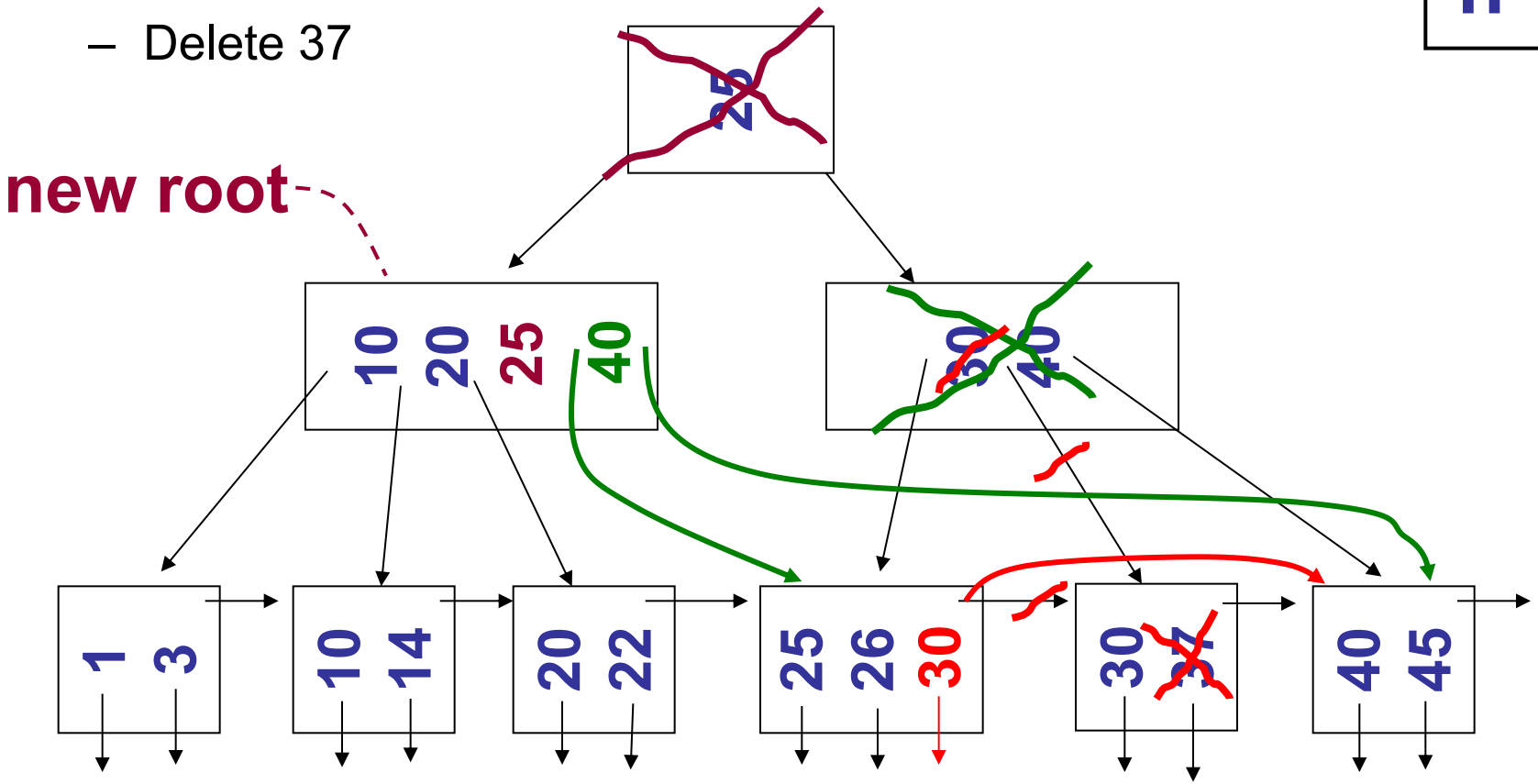


Deletion in a B$^+$-tree: examples

n=4

(d) Non-leaf coalesce

– Delete 37



Clustering B$^+$-tree index: an analysis

Note that:

- We assume alternative (1) and we assume that F is the fan-out of the tree
- Empirical studies prove that in a B+-tree, in the average, the pages in the leaves are filled at 67%, and therefore the number of pages with data entries is about 1.5B, where B is the minimum number of pages required for storing the data entries (in case of alternative 1, this coincides with the pages for the data records). So, the number of physical pages forming the leaves of the tree is B'=1.5B

- Scan: 1.5 B → O(B)
- Selection based on equality:

  $\log_F(1.5\ B) \rightarrow O(\log_F(B))$

  - Search for the first page with a data record of interest
  - Search for the first data record, through binary search in the page
  - Typically, the data records of interest appear in one page (otherwise we have to count other page accesses)
- **N.B.** In practice, the root is in the buffer, so that we can avoid one page access
- **N.B.** If alternative (2) is used, then we compute the number B of leaf pages required to store the data entries, and we count again B'= 1.5B.

Clustered B$^+$-tree index

- Selection based on range:
  - same as selection based on equality
  - but with further I/O operations, if the data records of interest are spread in several (linked) leaves
- Insertion: $\log_F(1.5B) \rightarrow O(\log_F(B))$
  - cost of search + insertion + write
  - we ignore additional costs arising when the insertion is on a full page
- Deletion
  - similar to insertion
  - we ignore further costs arising when the deletion leaves the page empty

Exercise 3

We have carried out our cost analysis for the clustered B^+-tree index under the assumption of alternative 1.

Characterize the cost of the various operations for the clustered B^+-tree index under the assumption that alternative 2 is used, both in the case of dense index, and in the case of sparse index.

Do the same under the assumption of alternative 3.

Unclustered B$^+$-tree index

- We assume that the data file is a heap file.
- We assume a dense index using alternative (2), and we suppose that the size of a data entry in a leaf is 1/10 of the size of a data record; this means that, if B is the number of pages in the data file, 0.1B is the minimum number of pages required to store the leaves of the tree.
- Number of leaves in the index: 1.5(0.1 B)=0.15B
- Number of data entries in a leaf page (recall that a data entry page is 67% full), where R is the number of records per page in the data file: 10(0.67R)=6.7R
- F is the fan-out of the tree

➢ Scan: 0.15B(6.7R) + BR
  – Scan of all index data entries: 0.15B(6.7RC)
  – Every data entry in a leaf can point to a different data file page: BR

  →High cost: sometime it is better to ignore the index! If we want the records ordered on the search key, we can scan the data file and sort it -- the I/O cost of sorting a file with B pages can be assumed to be 4B (with a 2-pass algorithm in which at each pass we read and write the entire file and with sufficient space in the buffer), which is much less than the cost of scanning the unclustered index.

Unclustered B$^+$-tree index

- Selection based on equality:

$$\log_F(0.15B) + X$$

  - locate the first page of the index with the data entry of interest: $\log_F(0.15B)$
  - X: number of data records satisfying the equality condition

    → we may need one I/O operation for each of these data records

- Selection based on range:
  - Similar to selection based on equality
  - Note that

    → the cost depends on the number of data records (may be high)

Both for the scan and the selection operator, if we need to go to the data file, sometimes it might be convenient to avoid using the index. Rather, we can simply sort the file, and operate directly on the sorted file.

Unclustered B^+-tree index

- Insertion (of a single data record):

  $\log_F(0.15B)$

  - insert in the data file (unordered)
  - search for the correct position in the index to insert the data entry: $\log_F(0.15B)$
  - write the corresponding index page

- Deletion (of a single data record):

  $\log_F(0.15B)$

  - search of the data entry: $\log_F(0.15B)$
  - load the page with the data record to be deleted
  - modify/write the pages that have been modified in the index and in the file

Estimating the number of leaves

In a tree index, the analysis of performance of the various operations depends primarily by the number of physical pages stored as leaves. Here are some observations related to the issue of figuring out the number of leaves.

- The pages in the leaves are occupied at the 67%. This means that the physical pages are 50% more than the number of "required leaves", where the required leaves are the pages strictly **required** to store the data entries.
- When we use alternative 1, the number of **required** leaves is the number of pages in the data file (in this case, we accept that the physical pages in the data file is full at 67% of occupancy).
- When the index is dense and is on a key of the relation (primary index, or secondary, unique index), we have one data entry per data record. If we know how many data entries fit in one page, we can compute the number of **required** leaves (or express this number in terms of the number of data pages)
- When the index is dense, secondary non unique, and with duplicates then we must estimate the average number of data records having the same value for the search key. Using this information, and knowing how many data entries fit in one page, we can again compute the number of **required** leaves.
- When the index is sparse, then we know that the number of data entries in the required leaves is essentially equal to the number of pages in the data file, and again we should be able to compute the number of **required** leaves.

Discussion

- Heap file
  - Efficient in terms of space occupancy
  - Efficient for scan and insertion
  - Inefficient for search and deletion
- Sorted file
  - Efficient in terms of space occupancy
  - Inefficient for insertion and deletion
  - More efficient search with respect to heap file

Discussion

- Clustered tree index
  - Limited overhead in space occupancy
  - Efficient insertion and deletion
  - Efficient search
  - Optimal support for search based on range

- Static hash index
  - Efficient search based on equality, insertion and deletion
  - Optimal support for search based on equality
  - Inefficient scan and search based on range

→ No file organization is uniformly superior to the other ones in every situation