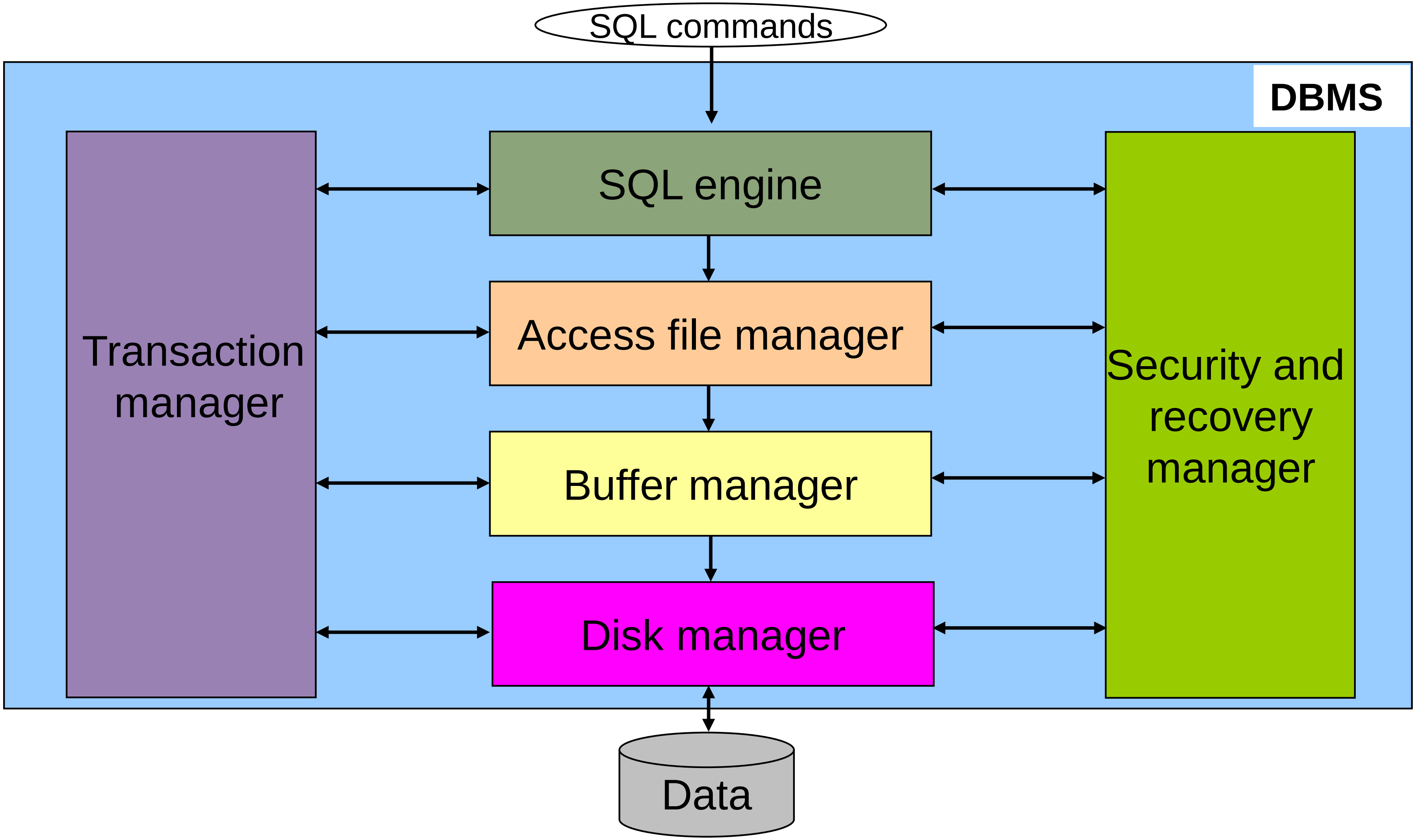
Database Management Systems: Access file manager

Maurizio Lenzerini, Riccardo Rosati

Dipartimento di Ingegneria informatica automatica e gestionale
Università di Roma “La Sapienza”

2018/2019

Architecture of a DBMS



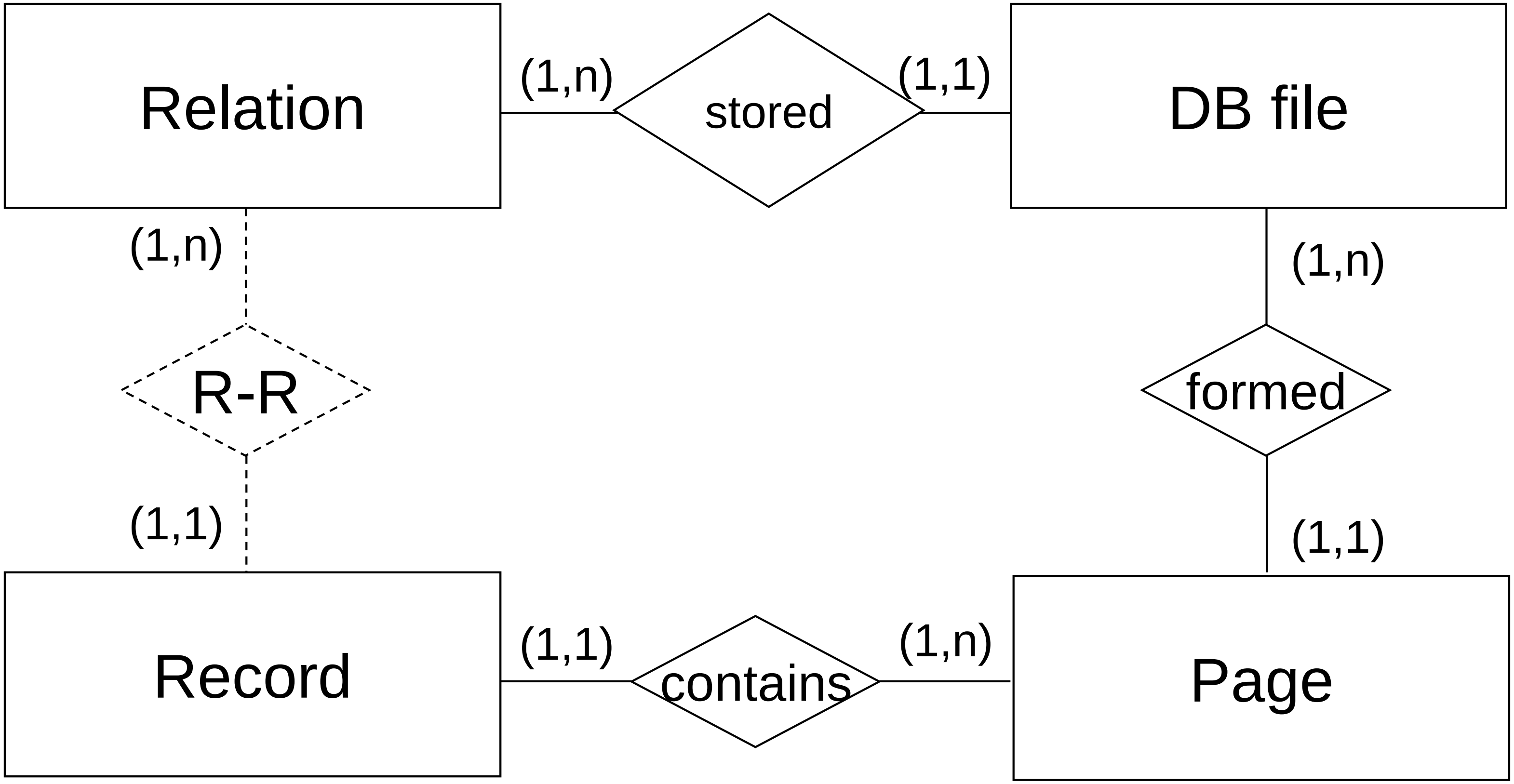
3. Access file manager

3.1 Pages and records
3.2 Simple file organizations
3.3 Index organizations

3. Access file manager

3.1 Pages and records
3.2 Simple file organizations
3.3 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

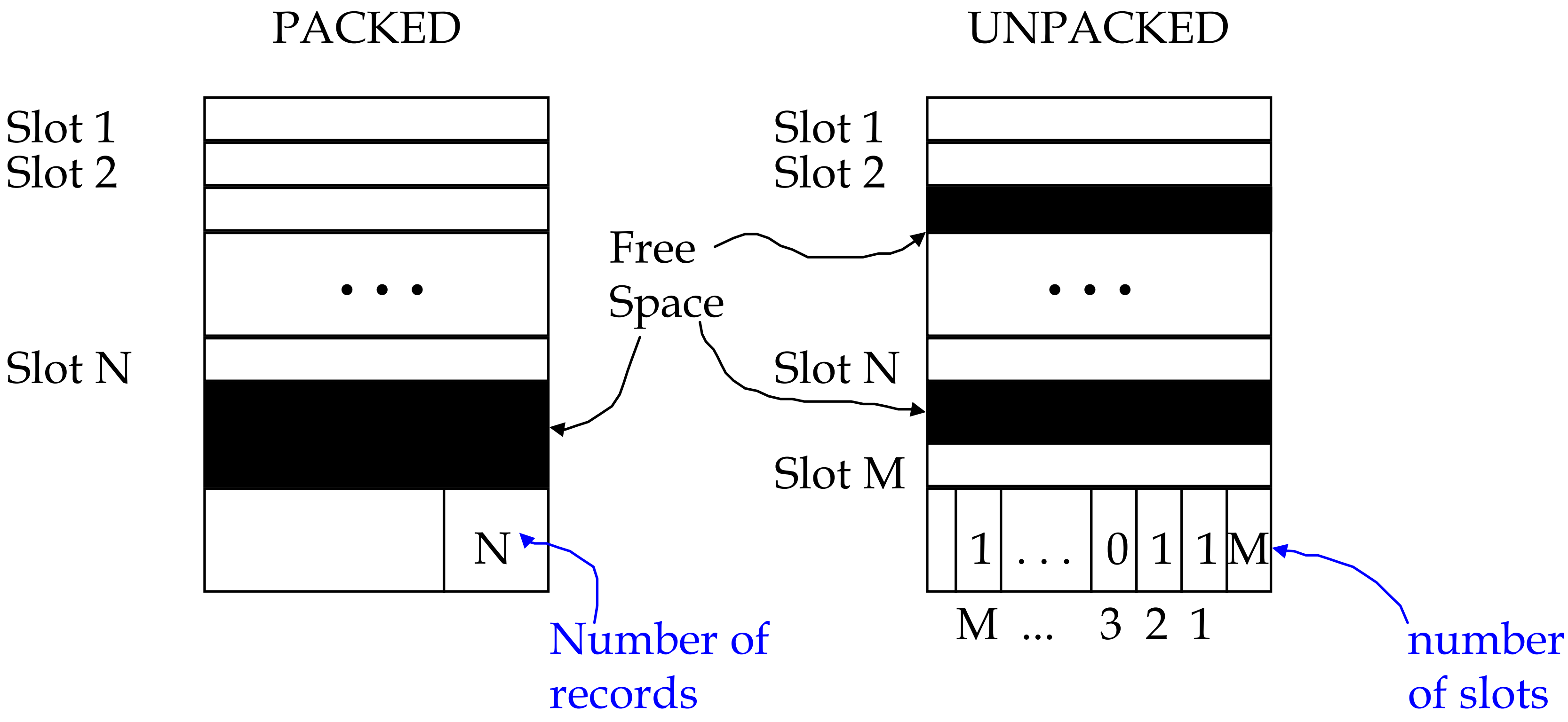
- A page is physically constituted by a set of slots

- A slot is a memory space that may contain one record (typically, all slots of a page contain records of one relation) and has a number that identifies it in the context of the page

- Each record has an identifier (record id, o rid)

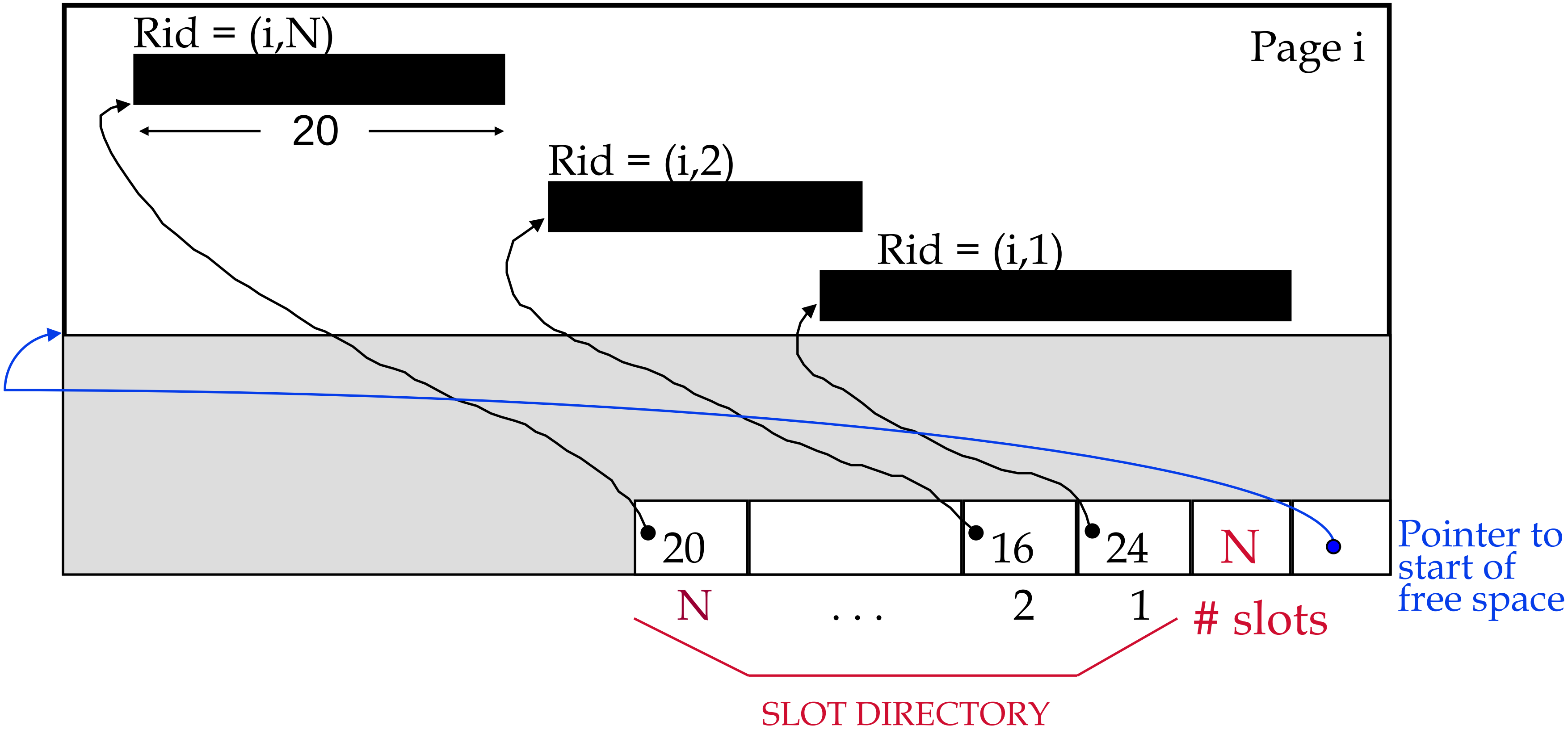
  rid = <page id, slot number>

Page with fixed length records



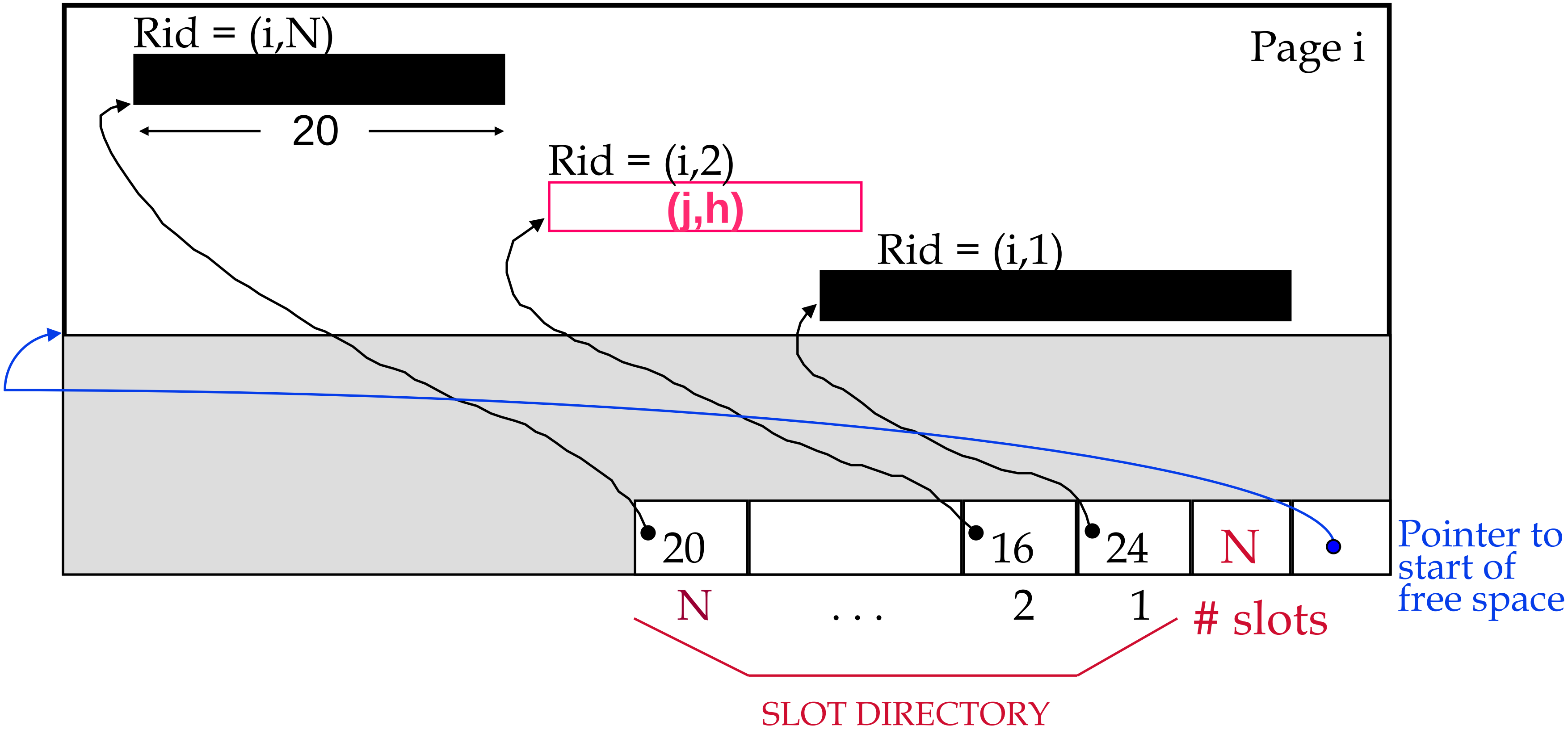
- Packed organization: moving a record changes its rid, and this is a problem, when records are referred to by other pages
- Unpacked organization: identifying a record requires to scan the bit array to check whether the slot is free (0) or not (1)

Page with variable length records



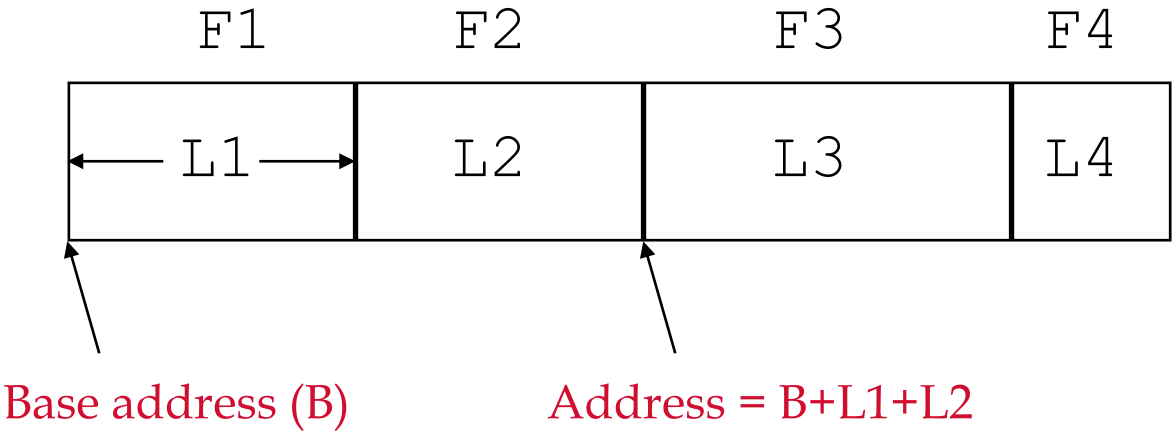
No problem in moving records to the same page! Deleting a record means to set to -1 the value of the corresponding slot, and move the record space to the free space (re-organizing the data area and the free space area when needed).

Page with variable length records



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

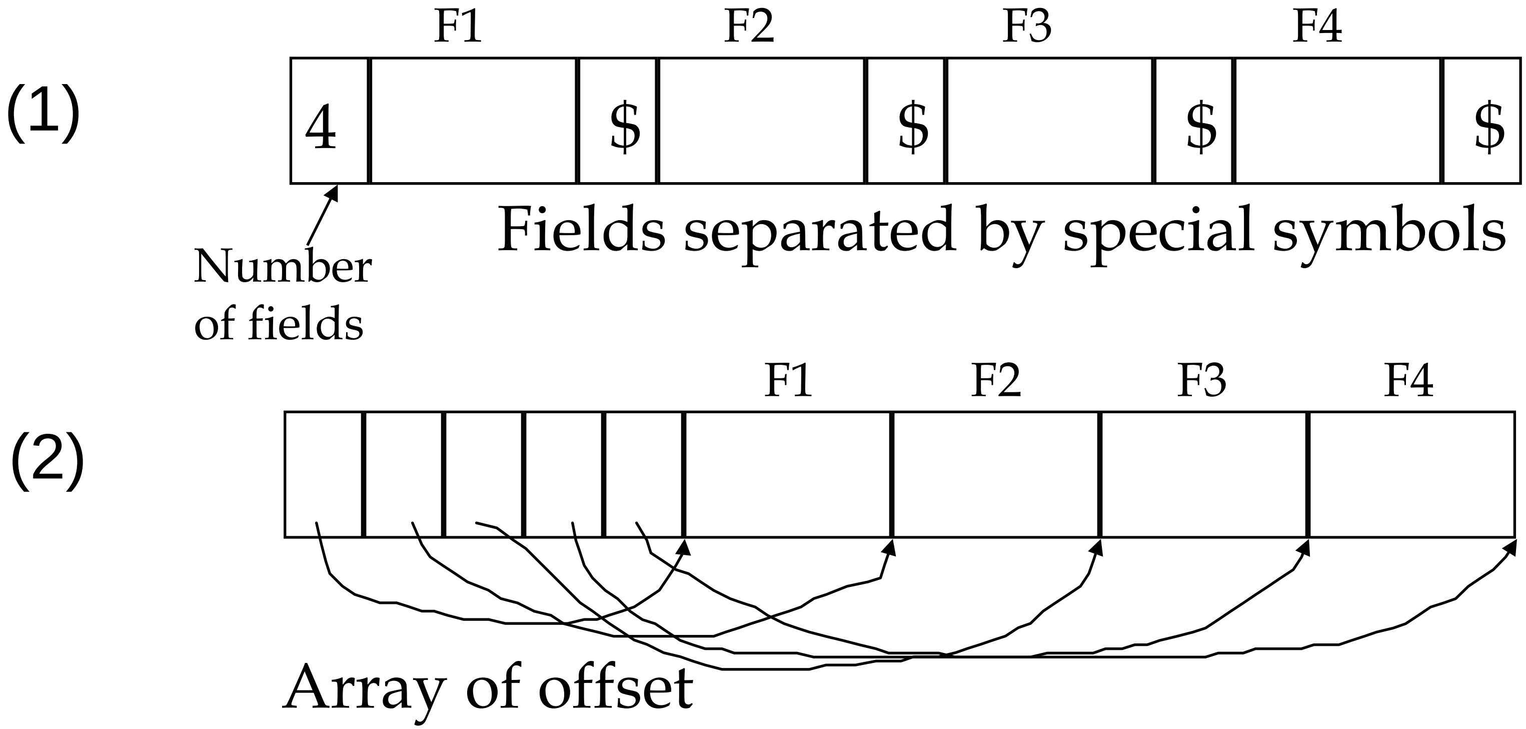
Format of a fixed length record



- Information on fields are the same for all records of the type, and are stored in the system catalog

Format of variable length records

Two alternatives (the number of fields is fixed):



Alternative (2) allows direct access to the fields, and efficient storage of nulls (two consecutive pointers that are equal correspond to a null)

3. Access file manager

3.1 Pages and records

3.2 Simple file organizations

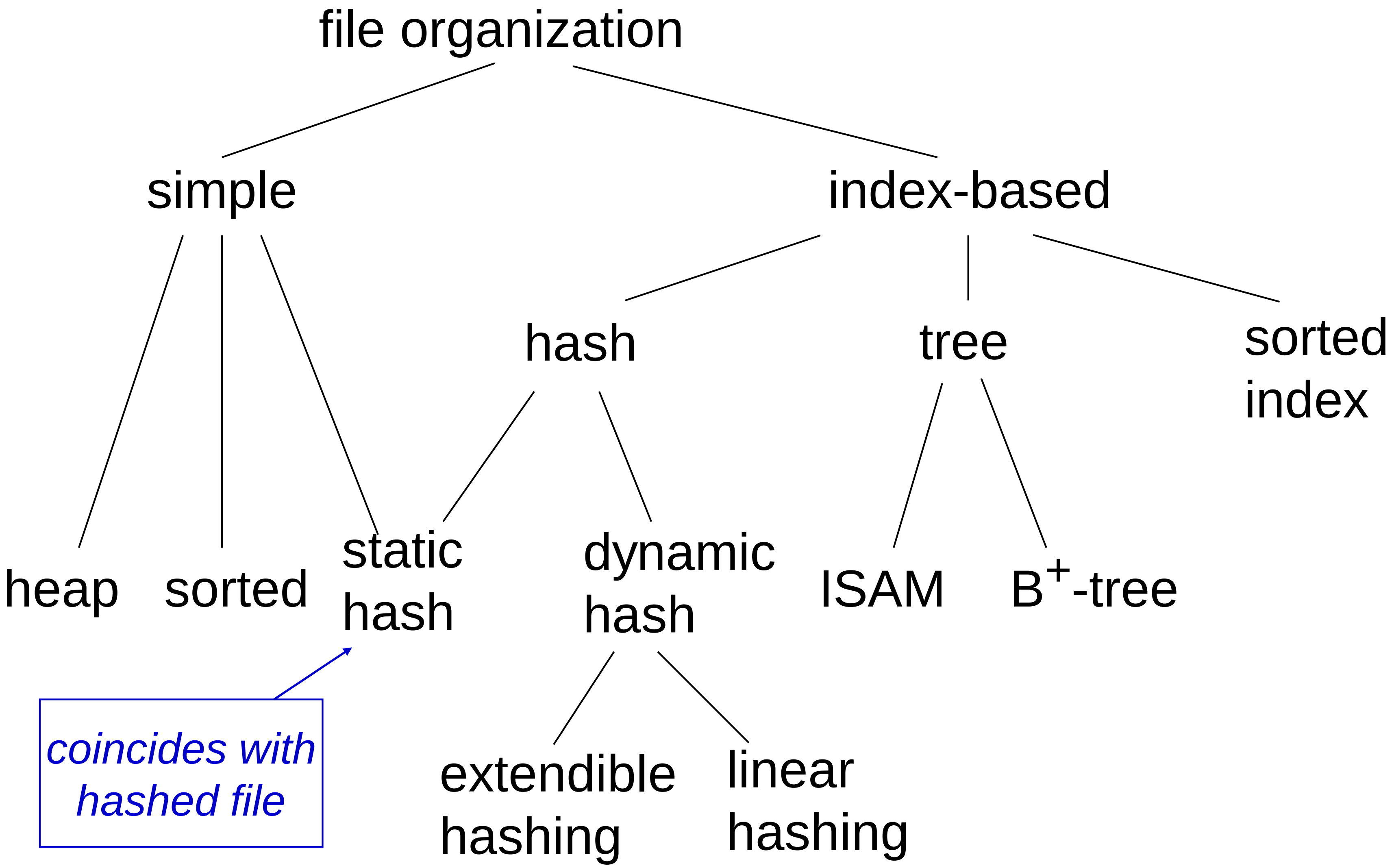
3.3 Index organizations

File

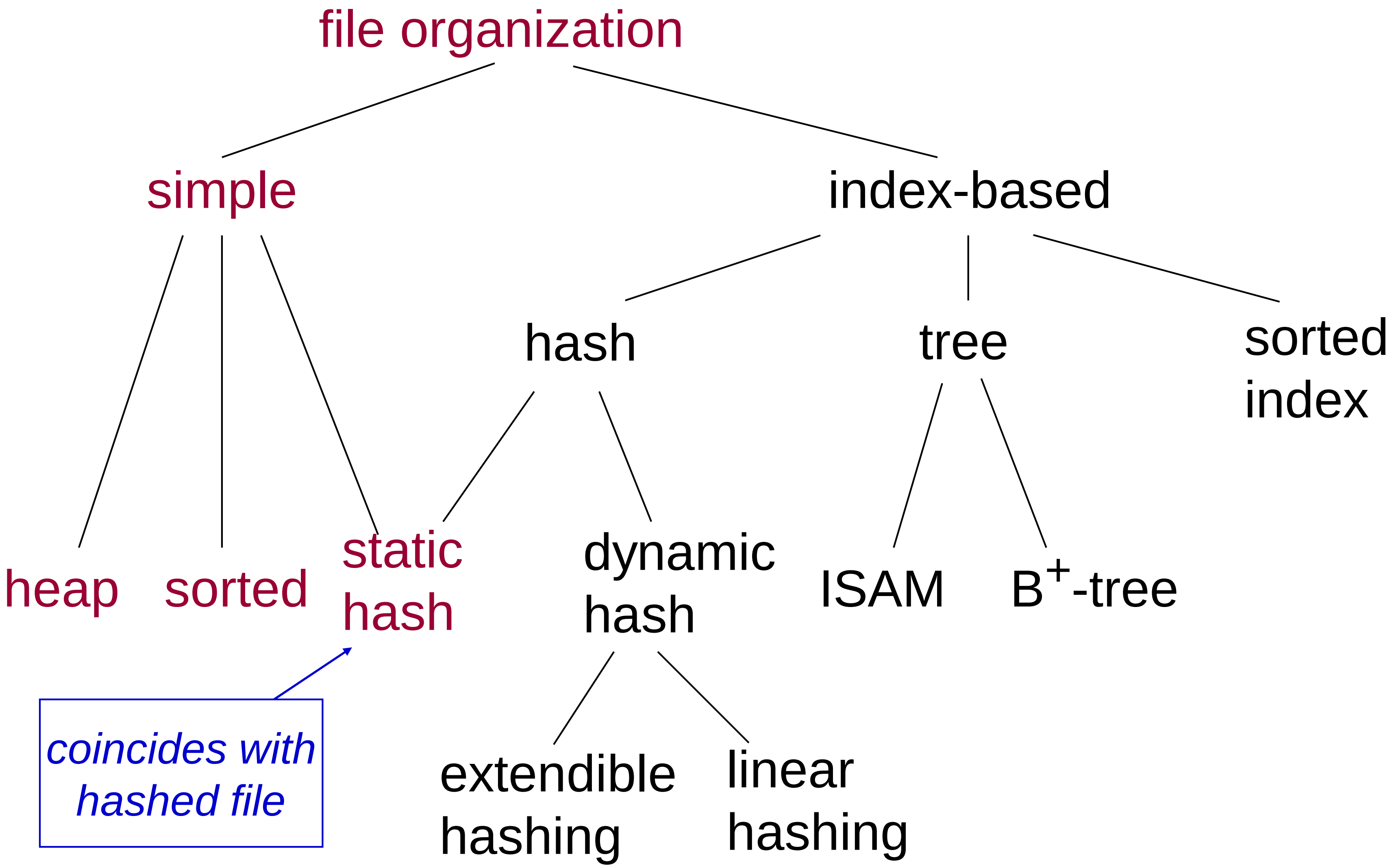
- A file is a collection of pages, each one containing a collection of records (as we saw before)

- A file organization should support the following operations:
  - insert/delete/update a record
  - read the record specified by its rid
  - scan all the records, possibly focusing on the records satisfying some given condition

File organizations



Simple file organizations



file organization

simple

index-based

hash

tree

sorted index

heap

sorted

static hash

dynamic hash

ISAM

B$^+$-tree

extendible hashing

linear hashing

coincides with hashed file

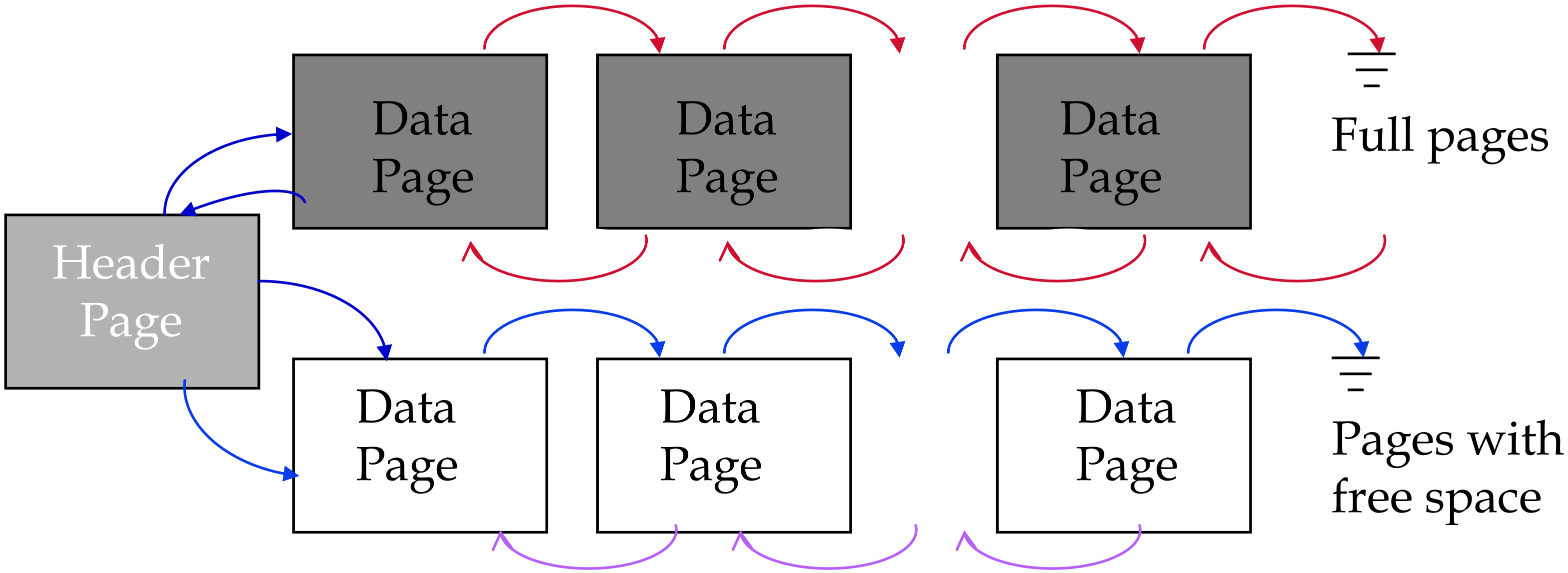
Heap File

- 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

- When the relation grows or shrinks, pages are allocated or de-allocated

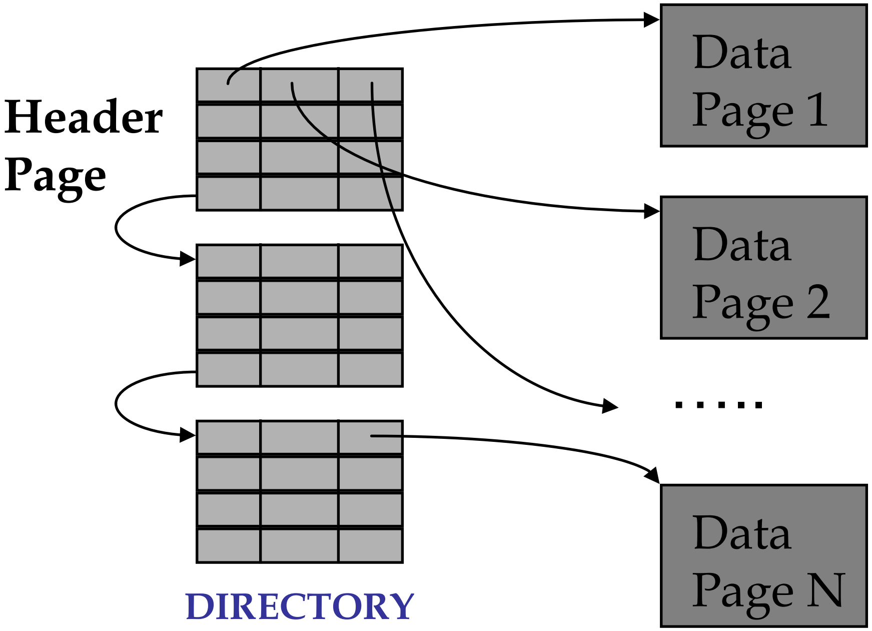
- To support the operations, it is necessary:
  - To keep track of the pages belonging to the file
  - To keep track of free space in the pages of the file
  - To keep track of the record in the pages of the file

Heap File represented through lists



- When a 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

Heap File represented through directory



- The directory is a list of pages, where each entry contains a pointer to a data page
- Each entry in the directory refers to a page, and tells whether the page has free space (for example, through a counter)
- Searching for a page with free space is more efficient, because the number of page accesses is linear with respect to the size of the directory, and not to the size of the relation (as in the case of the list-based representation)

File with sorted pages

- Records are sorted within each page on a set of fields (called the 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

Hashed file

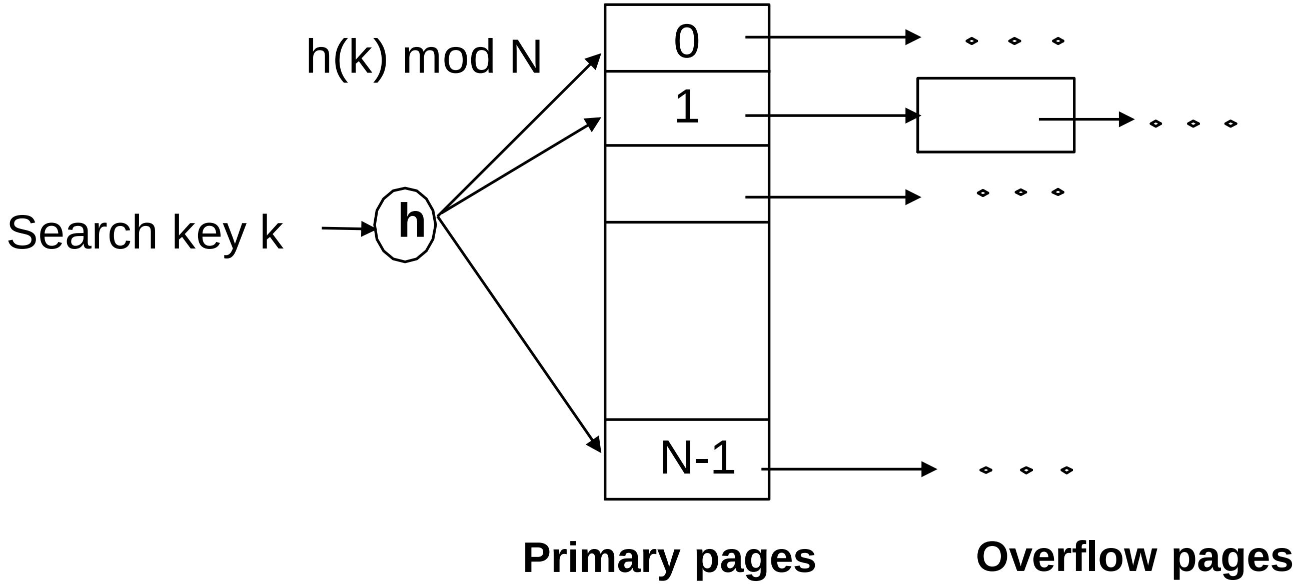
- The page of the relation are organized into groups, called *bucket*

- 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 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

- 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 often concentrate only on the number of page accesses 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 key (search based on key)
- Selection based on range
  - 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 occurs
  - 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
  - Like insertion

Heap file - cost of operations

We will ignore the cost of locating and managing the pages with free space

- Scan:

$$B(D + RC)$$

  – For each of the B pages of the file
    - load it (D)
    - for each of the R records of the page: process it (C)

- Equality selection:

$$B(D + RC)$$

  → 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 if the probability that the record is in the i-th page is 1/B, then the average cost is $B/2(D + RC)$

Heap file - cost of operations

- Range selection:

  $$B(D + RC)$$

- Insertion: $D + C + D$
  - Loading of the (last) page
  - Insertion in the page
  - Write the page
- Deletion
  - If the record is identified by rid: $D + C + D$
  - If the record is specified through an equality or range selection: $B(D + RC) + XC + YD$
    - X number of records to delete
    - Y number of pages with records to be deleted

Sorted file: search based on key

- The simplest method to perform the equality selection on the key (search based on the key) is by scanning the file. The average cost is B/2(D + RC), 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 is given by invoking the following algorithm with range (a_1, a_B).

Sorted file: search based on key

- To search the record with search key K 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 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, p_{i-1}), else repeat the algorithm using the range (p_{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

  $$p_k = (K - K_{min}) / (K_{max} - K_{min})$$

  is the probability that a record have a search key value less than or equal to K. This implies that

  $$K = K_{min} + p_k \times (K_{max} - K_{min})$$

  and therefore, assuming that the distance between addresses is analogous to the distance between key values, we can choose as tentative address

  $$i = a_1 + p_k \times (a_B - a_1)$$

  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

Sorted file - cost of operations

- Scan: B(D + RC)

- Equality selection on search key (with binary search)

  D $\log_2 B$ + C $\log_2 R$ (worst case)

  - binary search for locating the page with the (first) relevant record
    - ❿ $\log_2 B$ steps for locating the page
    - ❿ at each step, 1 I/O operation + 2 comparisons (that can be ignored)
  - binary search for locating the (first) relevant record in the page: C $\log_2 R$

- Equality selection on search key (with interpolation search)
  - in the average case: D ($\log_2 \log_2$ B) + C $\log_2 R$
  - in the worst case: B(D + RC)

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:

    $$D \log_2 B + C \log_2 R + (f_s \times B – 1)(D + RC)$$

    where $(D \log_2 B + C \log_2 R)$ is the cost of locating the first record with the value K_1, and $(f_s \times B – 1)(D + RC)$ is the cost of searching for the other records.

Sorted file - cost of operations

- Insertion:

  $$D \log_2 B + C \log_2 R + C + 2B(D + RC)$$

  - Worst case: first page, first position
  - Cost of searching the page to insert: $D \log_2 B + C \log_2 R$
  - Insertion of record: C
  - If we decide not to use overflow pages, the we must add the cost of loading and writing the other pages: 2B(D + RC)
  - Average case (insert in the middle of the file):

    $D \log_2 B + C \log_2 R + C + B(D + RC)$

- 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(D + RC)

  We assume (as usual) that pages are kept at about 80% occupancy, to minimize overflows as the file expands.

- Equality selection on search key:

  (D + RC) ´ (number of relevant records)

  We assume direct access through the hash function

- Range selection on search key: 1.25 B(D + RC)
- Insertion:

  2D + RC

- Deletion:

  Cost of search + D + RC

See later for a more detailed analysis

Comparison

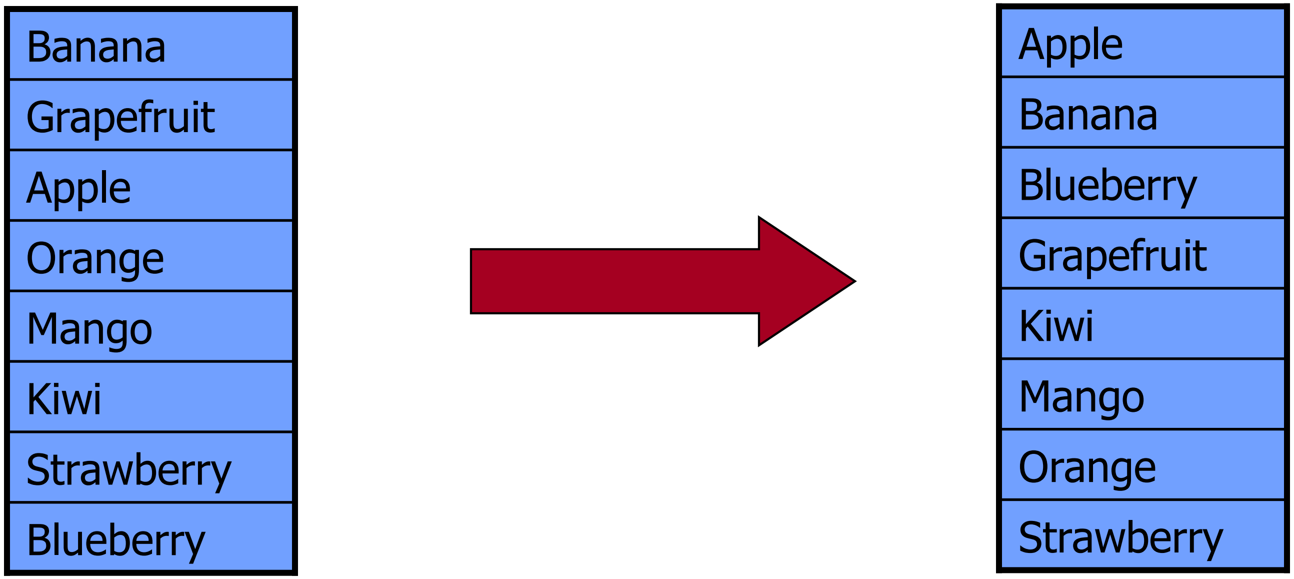
Organization	*Scan*	*Equality selection*	*Range selection*	*Insertion*	*Deletion*
Heap file	**BD**	**BD**	**BD**	**2D**	**Cost of search + D**
Sorted file	**BD**	**(search based on key)** **D $\log_2 B$**	**(on key)** **D $\log_2 B$ + number of relevant pages**	**Cost of search + 2BD**	**Cost of search + 2BD**
Hashed file	**1.25 BD**	**D (1 + number of relevant records)**	**1.25 BD**	**2D**	**Cost of search + D**

In the above table, we have only considered the cost of I/O operations, and we have assumed the use of binary search for sorted 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
- Sorting is useful for eliminating duplicates
- Sort-merge join algorithm involves sorting (see later)



Banana
Grapefruit
Apple
Orange
Mango
Kiwi
Strawberry
Blueberry

Apple
Banana
Blueberry
Grapefruit
Kiwi
Mango
Orange
Strawberry

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 these work for databases?

Sorting in secondary storage

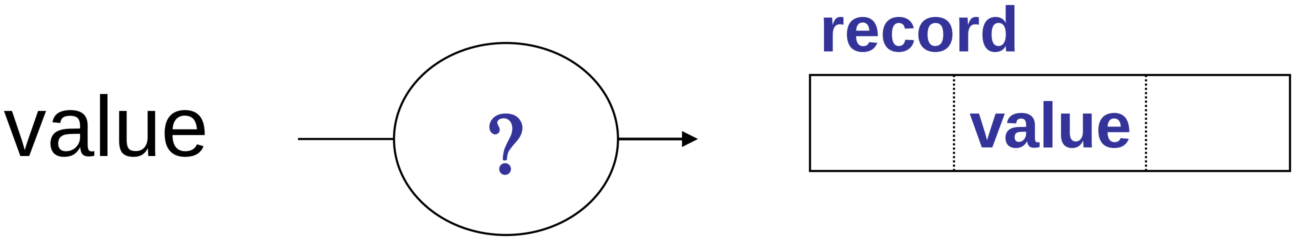
- The problem is how to sort data that do not fit in main memory
- Sorting of data in secondary storage (called external sorting) is very different from sorting an array in main memory (called internal sorting).
- We will consider an «external sorting» version of the merge-sort algorithm
- The complexity of sorting a data file of N pages through this algorithm is $\mathbf{N \log_F N}$, where F is the number of buffer slots that are available for this operation

3. Access file manager

3.1 Pages and records
3.2 Simple file organizations
3.3 Index organizations

The notion of index

An index is any method that that takes as input a property 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. e.g., in the case of hash-based index), containing
  - Data entry, 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 entry (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

  the index is a tree

- Hash-based

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

Possible structures of a data entry

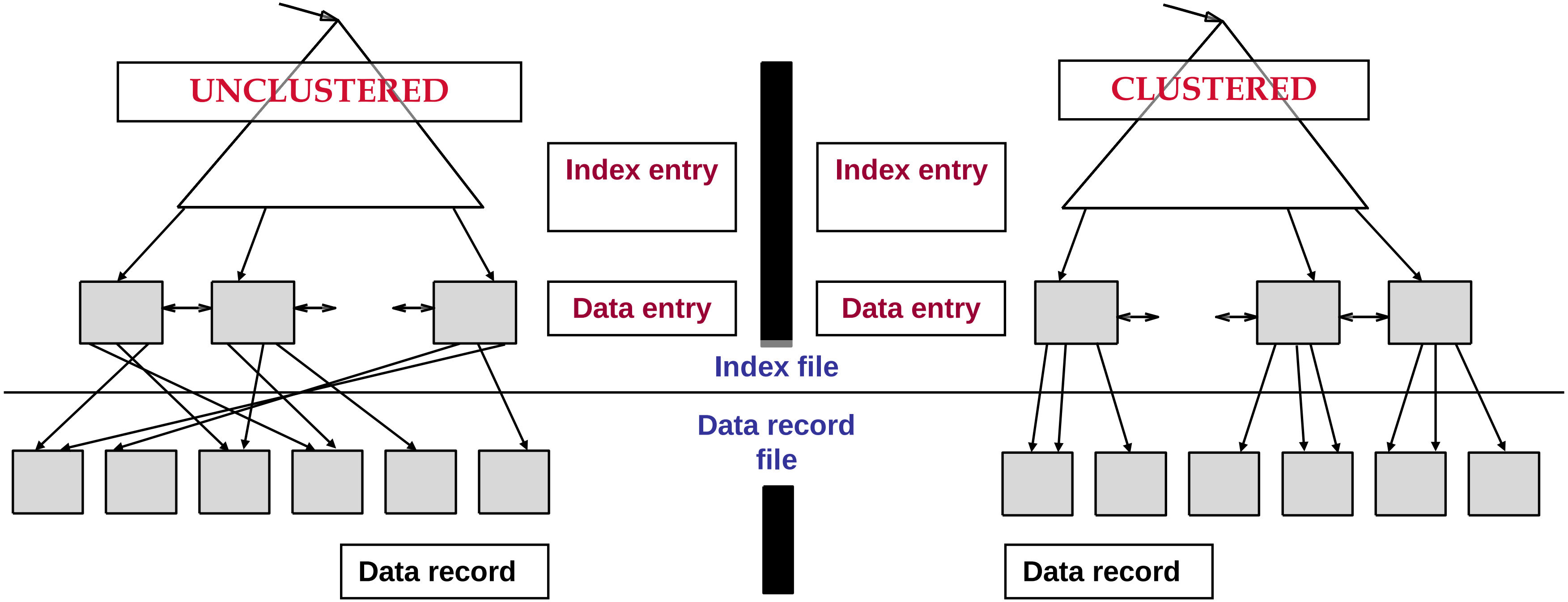
There are three main alternative techniques for storing a data entry whose search key value is k (such a data entry is denoted with **k***):

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 (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) to a data record with search key equal **k**
   - the index file is independent 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 independent 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 *clustering (also called 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-clustering (or, unclustered)*.



Clustering/non-clustering

- An index 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 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.

Clustering/non-clustering

In the case where the data file is not store in any order, or in the case where it is ordered according to a different ordering key, the index may still be clustering: indeed, in this case, we say that the index is 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 *I* appears in the page of the data file (or, on roughly as few pages as can hold them).

Indeed, some authors define an index *I* to be clustering if all the tuples of the indexed data file with a fixed value for the search key used in *I* appear on roughly as few pages as can hold them. Note that our definition of clustering implies this alternative definition.

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)*.

- In some text, the term “primary index” is used with the same meaning that we assign to the term “clustering index”, and the term “secondary index” is used with the same meaning that we assign to the term “non-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. Note that a unique secondary index does not contain duplicates.

Primary and secondary indexes

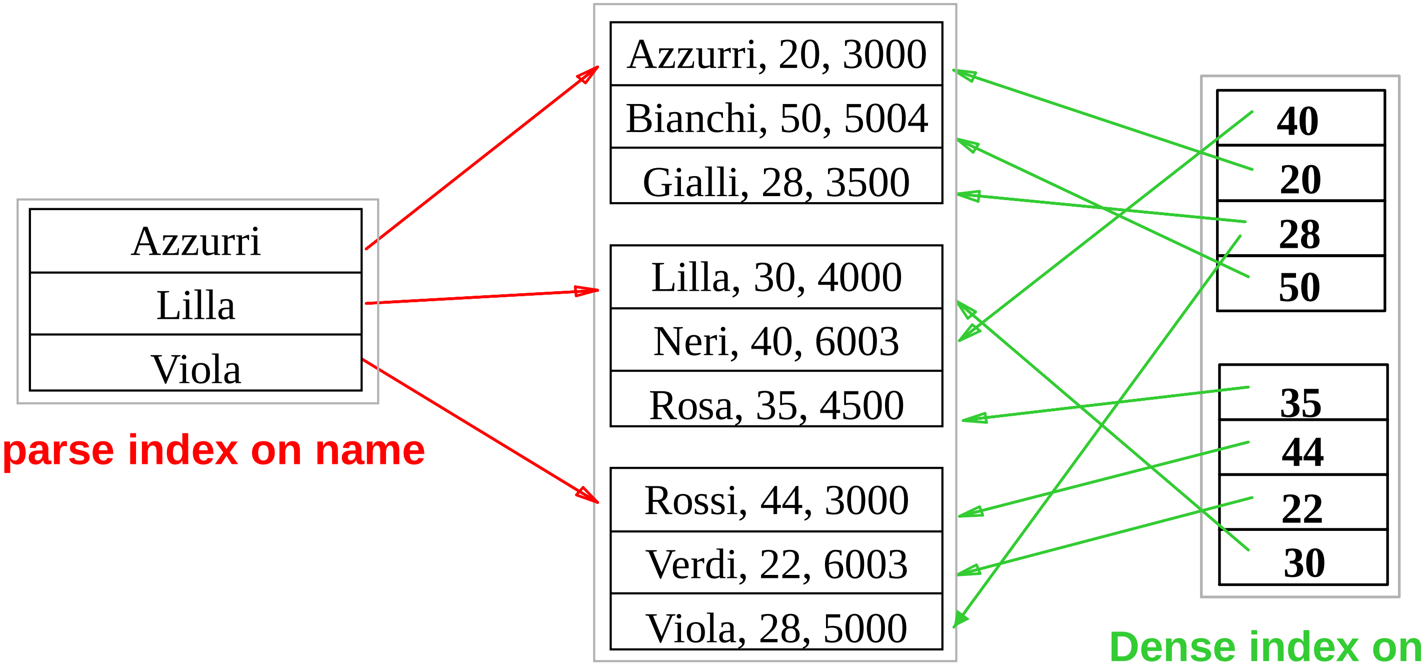
If a *secondary non-unique index* does not contain duplicates, then there are two possibilities:

– 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 in this case the index is certainly clustered. Indeed, 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 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* (typically, a dense index keeps a search key value for each data block)



Azzurri
Lilla
Viola

Azzurri, 20, 3000
Bianchi, 50, 5004
Gialli, 28, 3500

Lilla, 30, 4000
Neri, 40, 6003
Rosa, 35, 4500

Rossi, 44, 3000
Verdi, 22, 6003
Viola, 28, 5000

40
20
28
50

35
44
22
30

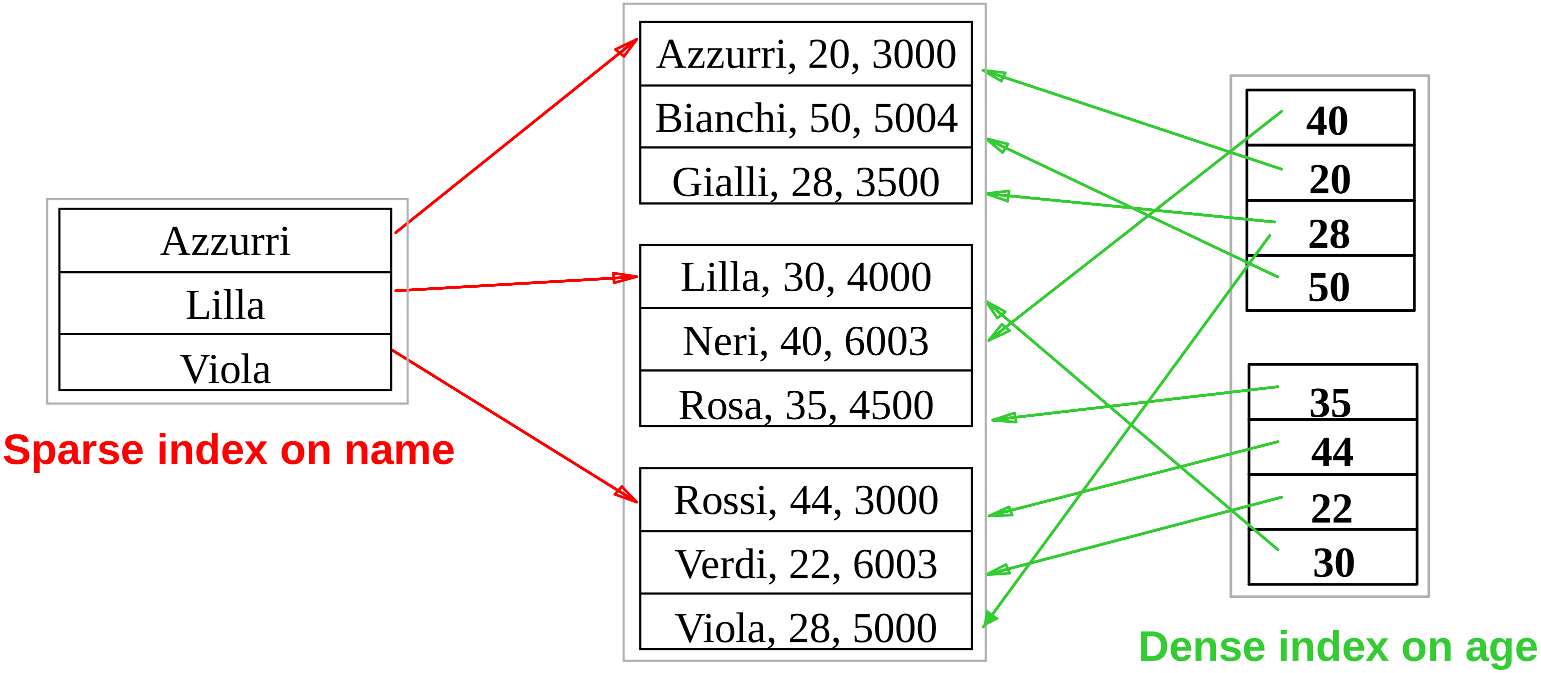
Sparse index on name

Dense index on age

- 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

Typically, in a *dense* index, we have one data entry per data record, where the value of the search key of the 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. In the figure, the red arrows denote record identifiers.



Azzurri
Lilla
Viola

Azzurri, 20, 3000
Bianchi, 50, 5004
Gialli, 28, 3500

Lilla, 30, 4000
Neri, 40, 6003
Rosa, 35, 4500

Rossi, 44, 3000
Verdi, 22, 6003
Viola, 28, 5000

40
20
28
50

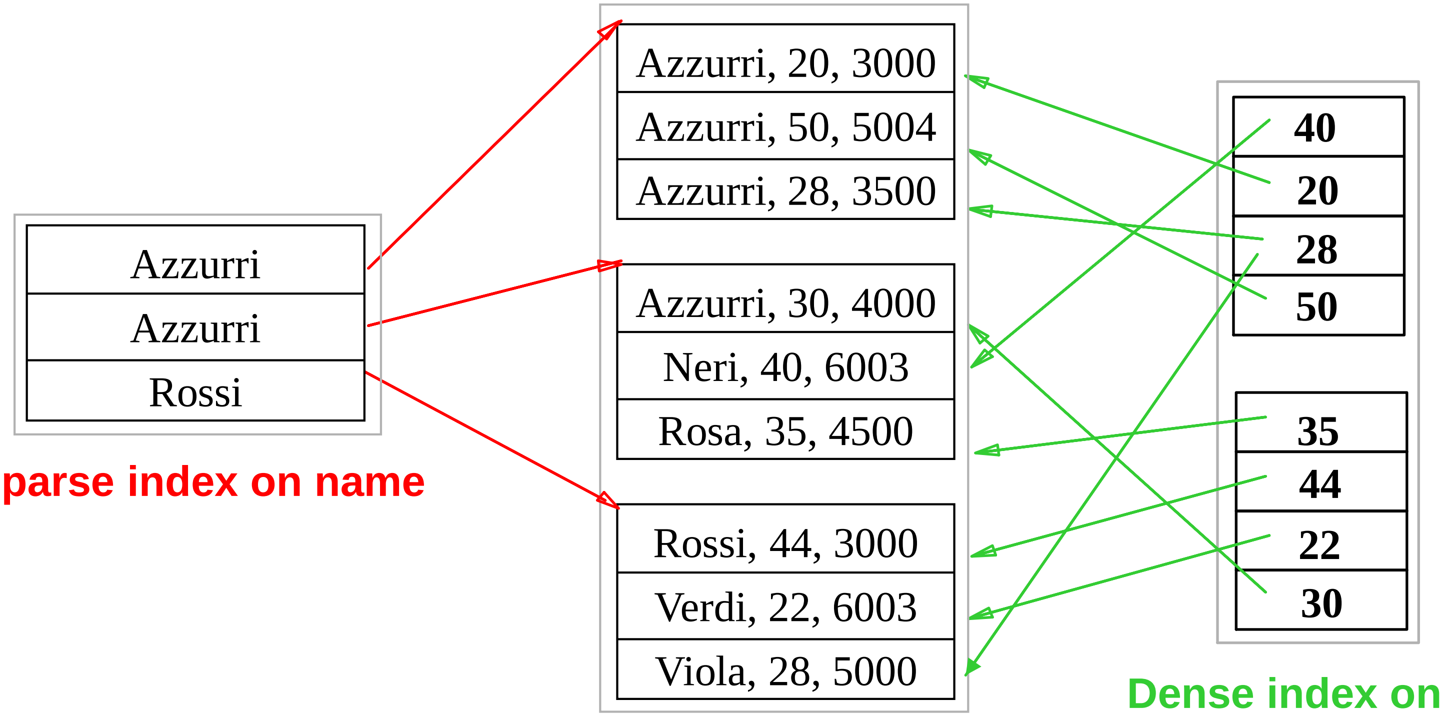
35
44
22
30

Sparse index on name

Dense index on age

Sparse vs. dense

Typically, in a *sparse* index, 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).



Azzurri
Azzurri
Rossi

Azzurri, 20, 3000
Azzurri, 50, 5004
Azzurri, 28, 3500

Azzurri, 30, 4000
Neri, 40, 6003
Rosa, 35, 4500

Rossi, 44, 3000
Verdi, 22, 6003
Viola, 28, 5000

40
20
28
50

35
44
22
30

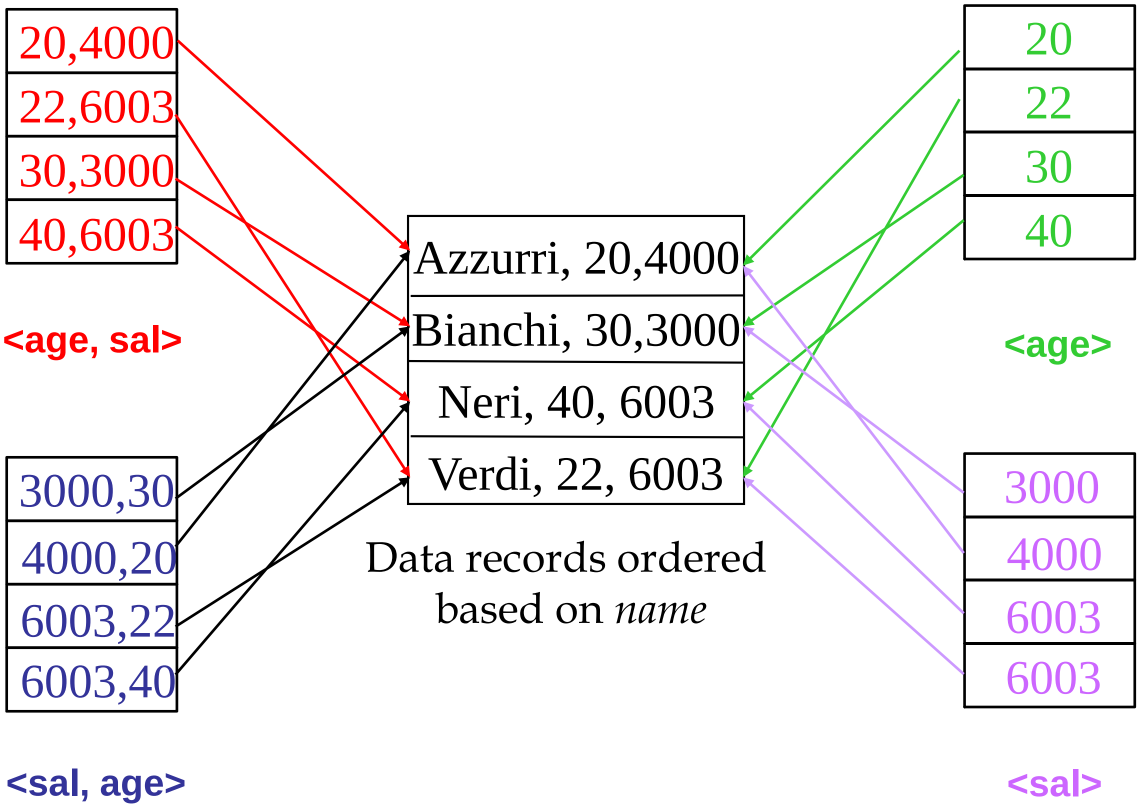
Sparse index on name

Dense index on age

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 some of the fields 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, we can even avoid accessing the data records, i.e., we can carry out an *index-only evaluation* (in particular, when all the fields that are relevant in the query are part of the search key)
- On the other hand, a composite index is generally more subject to update than a simple one.

Single vs composite key



20,4000
22,6003
30,3000
40,6003

<age, sal>

3000,30
4000,20
6003,22
6003,40

<sal, age>

Azzurri, 20,4000
Bianchi, 30,3000
Neri, 40, 6003
Verdi, 22, 6003

Data records ordered based on *name*

20
22
30
40

<age>

3000
4000
6003
6003

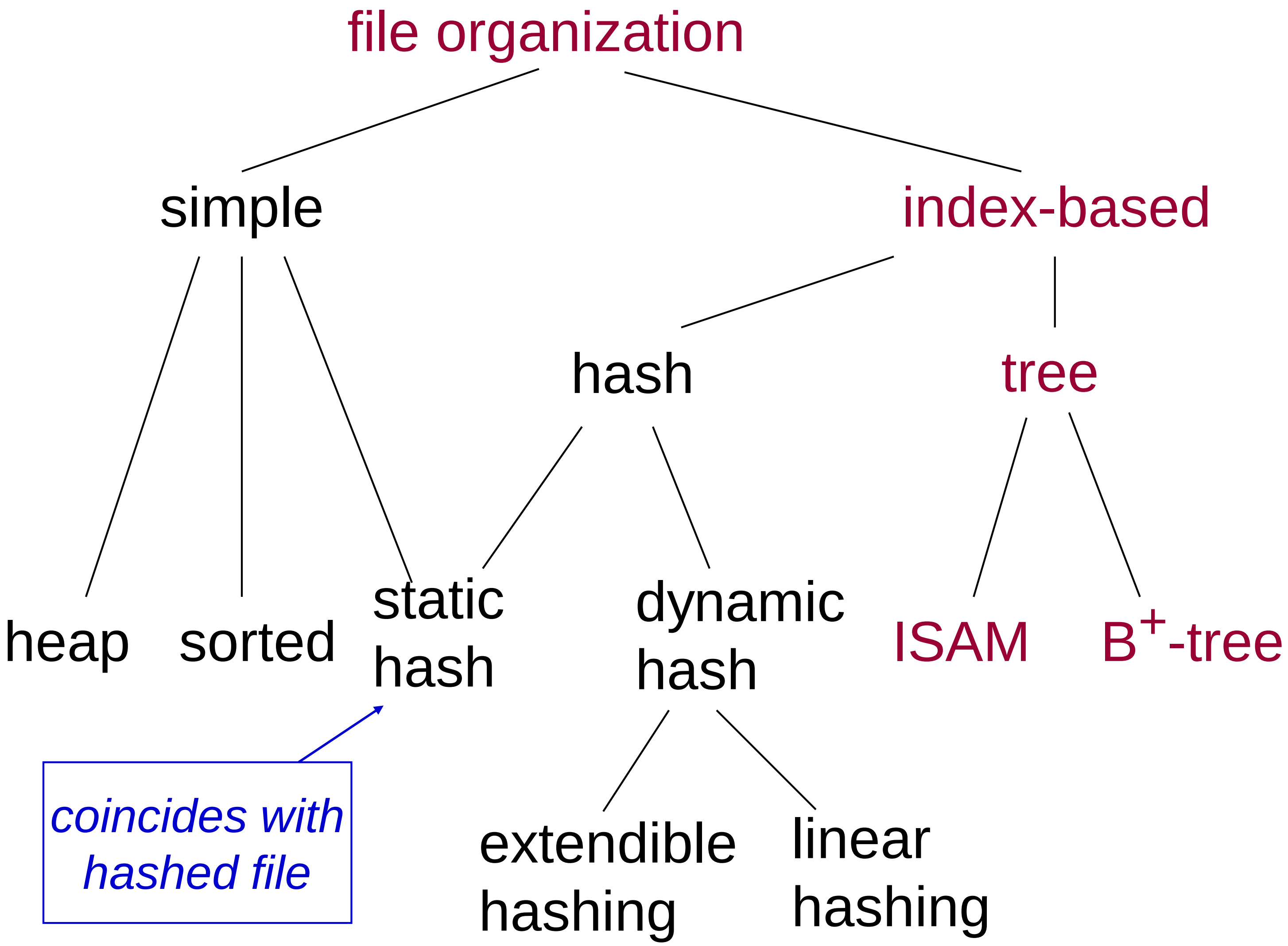
<sal>

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

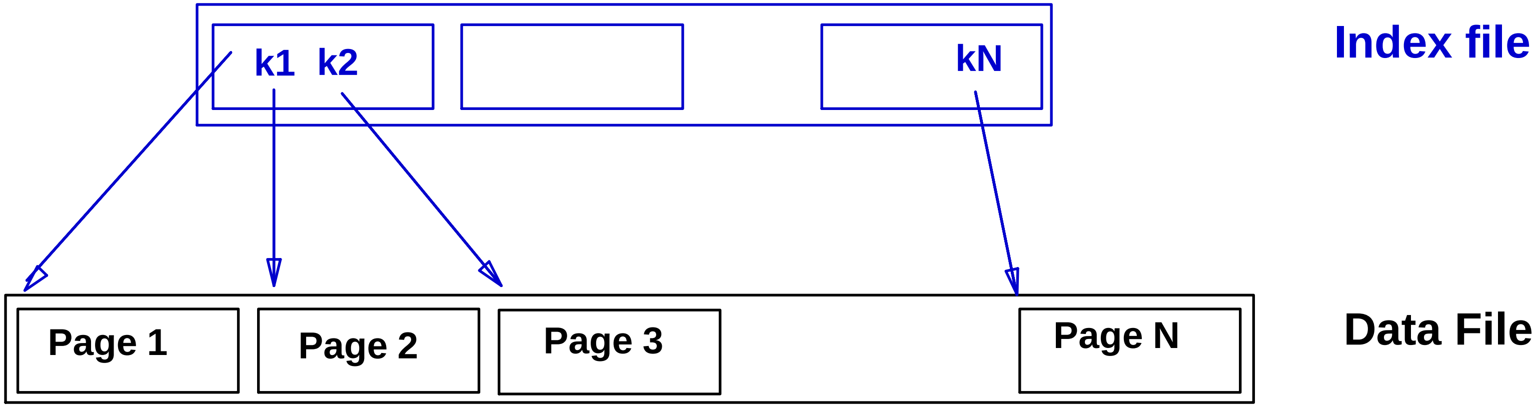
- A multi level index is an index organization where an index is built on a structure that is in turn an index file (for a data file or, recursively, for another index structure).

Tree-based index organization



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



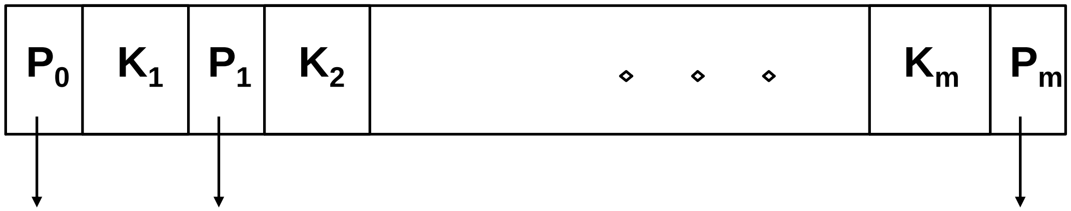
The binary search can now be carried out on a smaller file (data entries are smaller than data records, and we can even think of a sparse index)

Tree index: general characteristics

- 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, when we talk about index, we mean ISAM, or B$^+$-tree index

Tree index: general characteristics

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



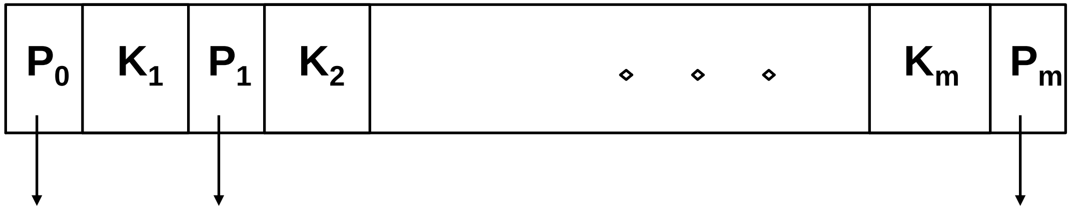
P_0 K_1 P_1 K_2 ⋄ ⋄ ⋄ K_m P_m

- – 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 $(1 \le i \le m)$ 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 it exists)

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

Exercise 4

We just saw the typical structure of an intermediate node (including the root) is as follows:



P_0	K_1	P_1	K_2	⋄ ⋄ ⋄	K_m	P_m

Prove that, if all the key values appearing in the non-leaf nodes of a B+ tree T appear also in the leaf nodes, then every key value K appearing in a non-leaf node of T appears also as the leftmost key value found in the leftmost leaf reachable from the pointer at the “right” of K.

Tree index: two types

- ISAM

  used when the relation is static (no insertion or deletion on the tree)

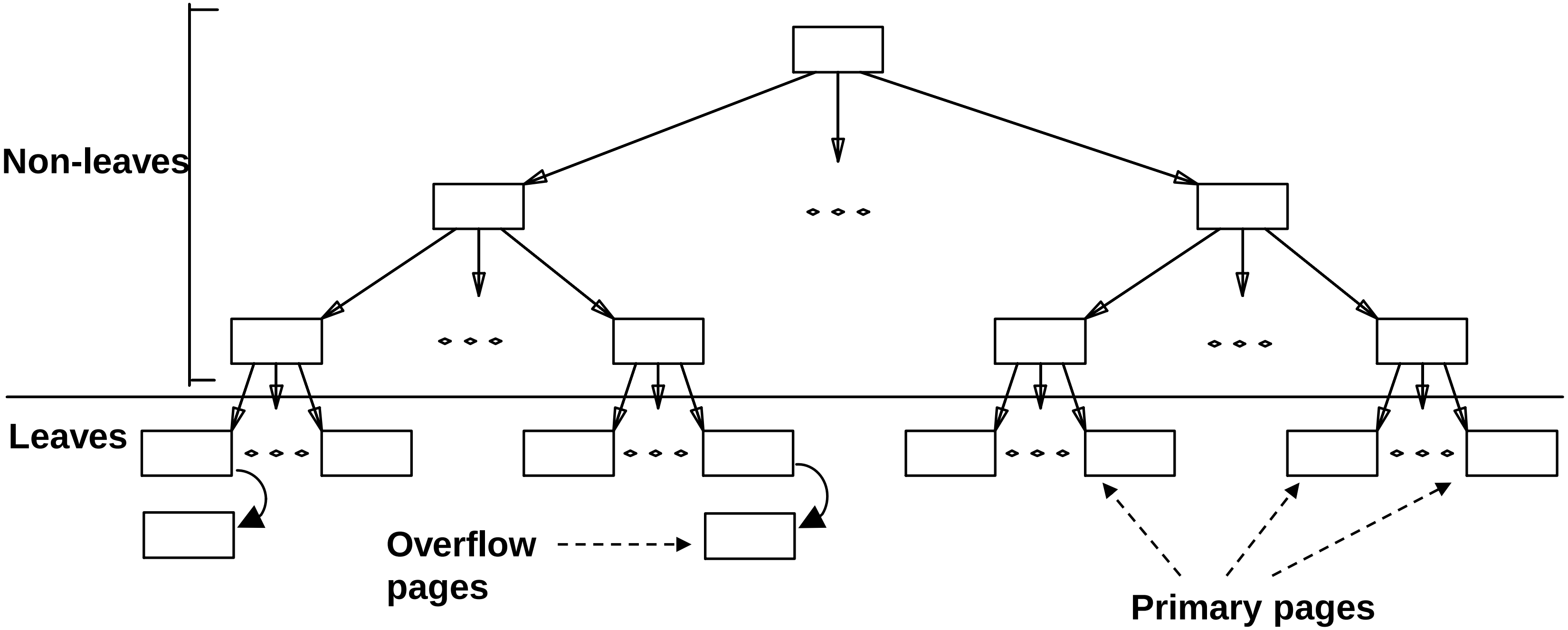
- B+-tree

  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: no update on 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
- 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 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 we compare the key we are looking for with the keys in the tree, until we arrive at a leaf. The cost is

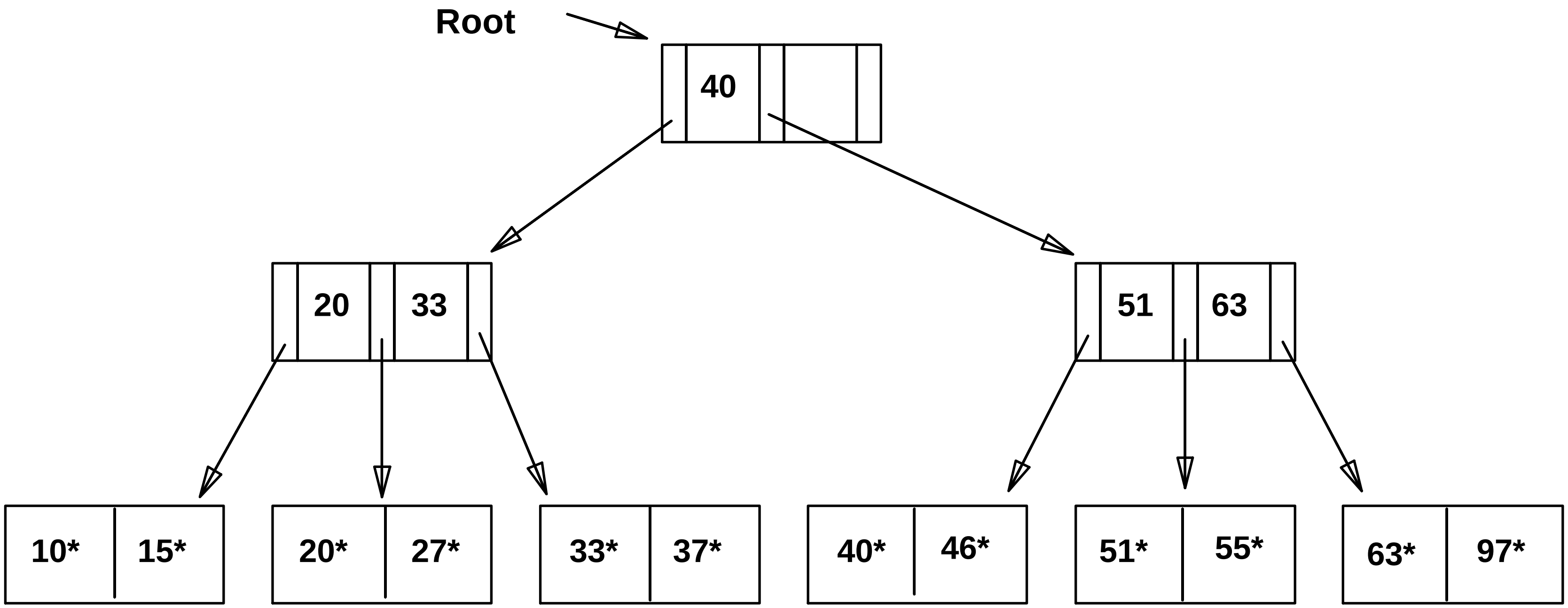
  $$\log_F N$$

  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

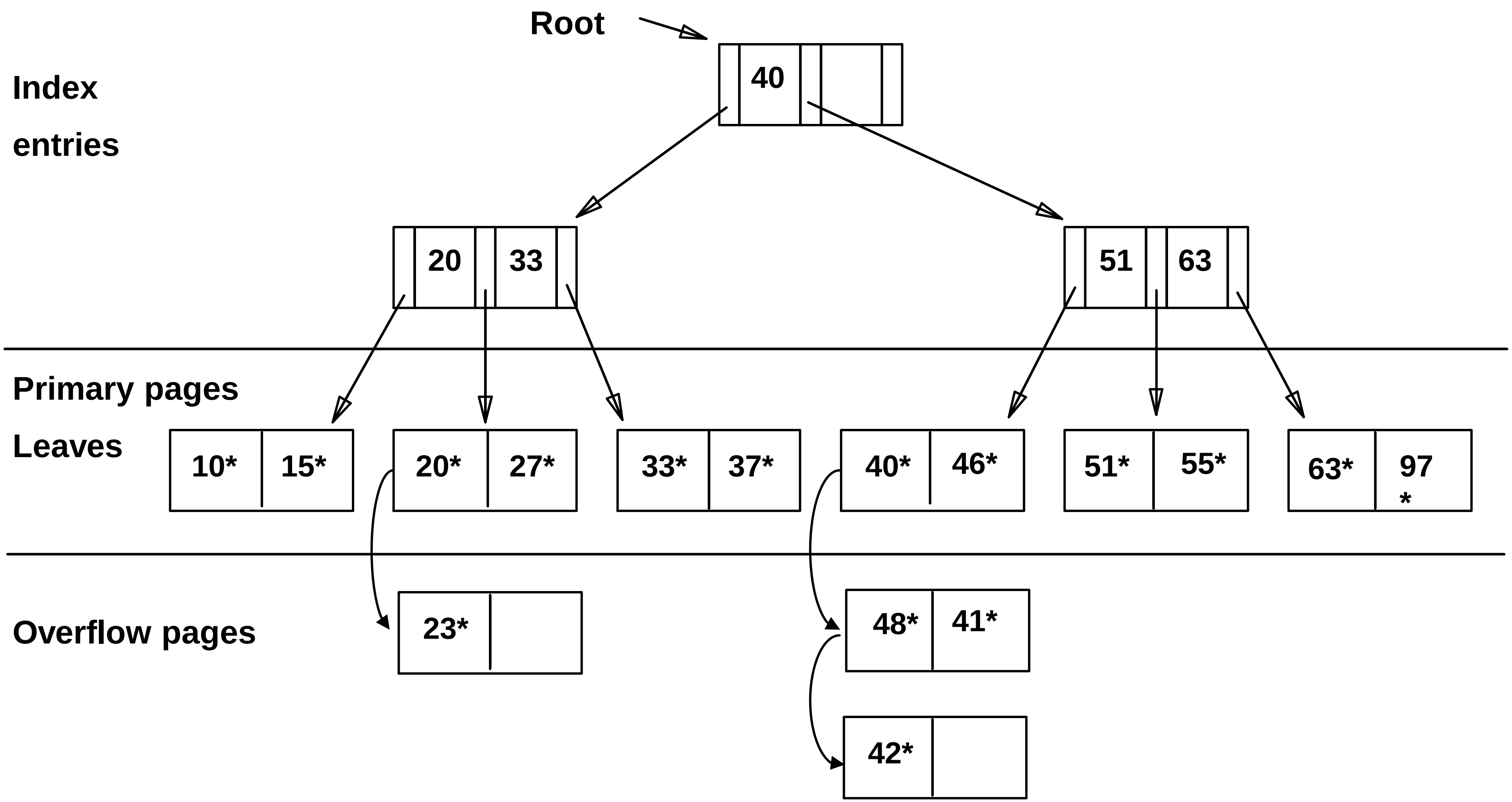
Static structure: *insert/delete are rare, and involve only leaves*

ISAM: example

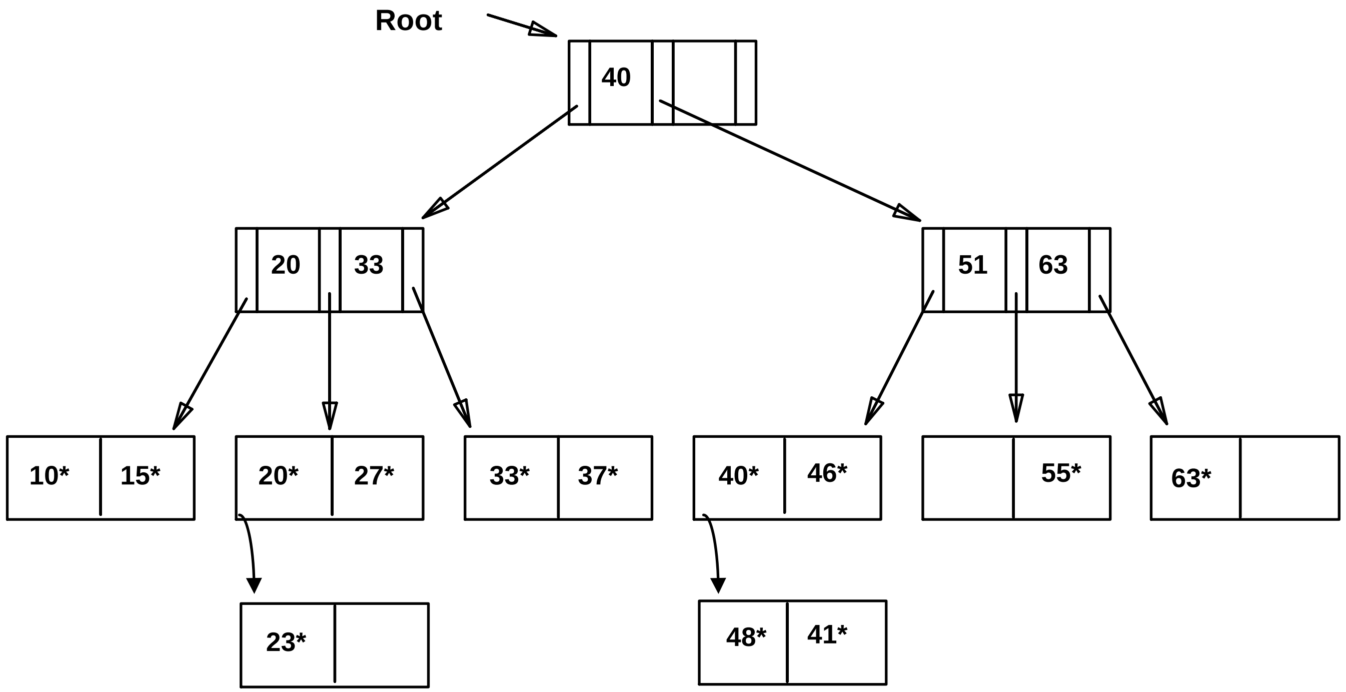
We assume that every node contains 2 entries (three pointers), except the root that may contain less than 2 entries)



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



Deletion of 42*, 51*, 97*



Note that 51* appears in the index entries, but not in the leaves

B^+-tree index

- A B^+-tree is 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 search key values, with (**d**+1)/2 <= m_i <= **d**. The only exception is the root, which may have less search key values (at least one)
- The leaves (the pages with the data entries) are linked through a list based on the order on the search key

  → Such list is useful for “range” queries:

  - 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 $\log_n m$.

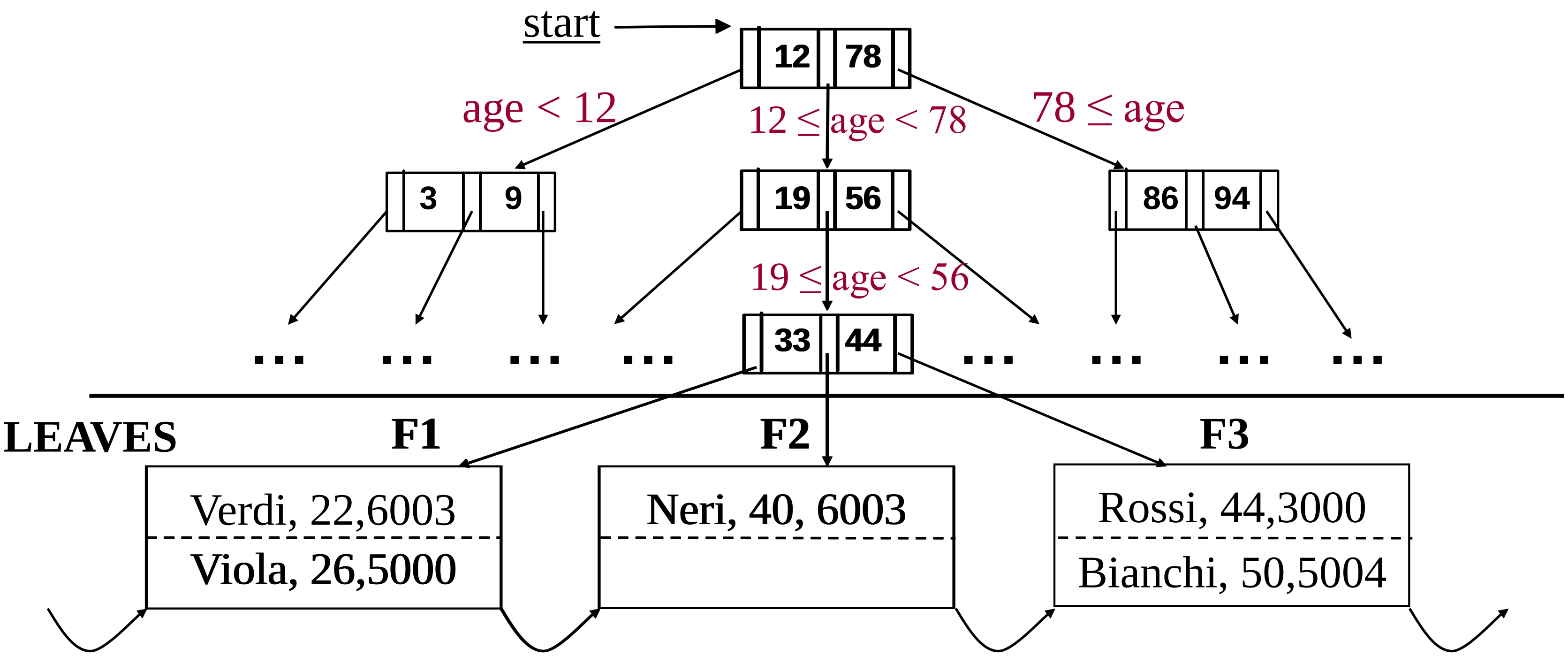
- If different nodes may have different numbers of children, then using the average value F for non-leaf nodes (instead of n), we get F^h as a good approximation to the number of leaf pages, and, knowing that there are m leaves, we get $\log_F m$ as a good approximation of the height h.

Search through B$^+$-tree: example

We look for data entries with 24 < age ≤ 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) is at most the height of the tree (in what follows, 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; note 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

Clustered B^+-tree index

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 for the leaves is B'=1.5B

➢ Scan: 1.5B (D+RC)

➢ Selection based on equality:

$$D \log_F(1.5B) + C \log_2 R$$

- ➢ 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

➢ **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: $D \log_F(1.5B) + C \log_2 R + C + D$
  - 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

Unclustered B^+-tree index

- We assume that the data file is a heap file.
- We assume a sparse 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): 10(0.67R)=6.7R
- F is the fan-out of the tree

➢ Scan: 0.15B(D+6.7RC) + BR(D+C)

– Scan of all index data entries: 0.15B(D+6.7RC)

– Every data entry in a leaf can point to a different data file page: BR(D+C)

→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), which is much less than the cost of scanning the unclustered index.

Unclustered B^+-tree index

- Selection based on equality:

$$D \log_F(0.15B) + C \log_2(6.7R) + XD$$

  - locate the first page of the index with the data entry of interest: $D \log_F(0.15B)$
  - binary search for locating the first data entry in the page: $C \log_2(6.7R)$
  - 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):

  $$2D + C + D \log_F(0.15B) + C \log_2(6.7R) + D$$

  – insert in the data file (unordered): 2D+C

  – search for the correct position in the index to insert the data entry: $D \log_F(0.15B)+C \log_2(6.7R)$

  – write the corresponding index page: D

- Deletion (of a single data record):

  $$D \log_F(0.15B) + C \log_2(6.7R) + D + 2(C+D)$$

  – search of the data entry: $D \log_F(0.15B)+C \log_2(6.7R)$

  – load the page with the data record to be deleted: D

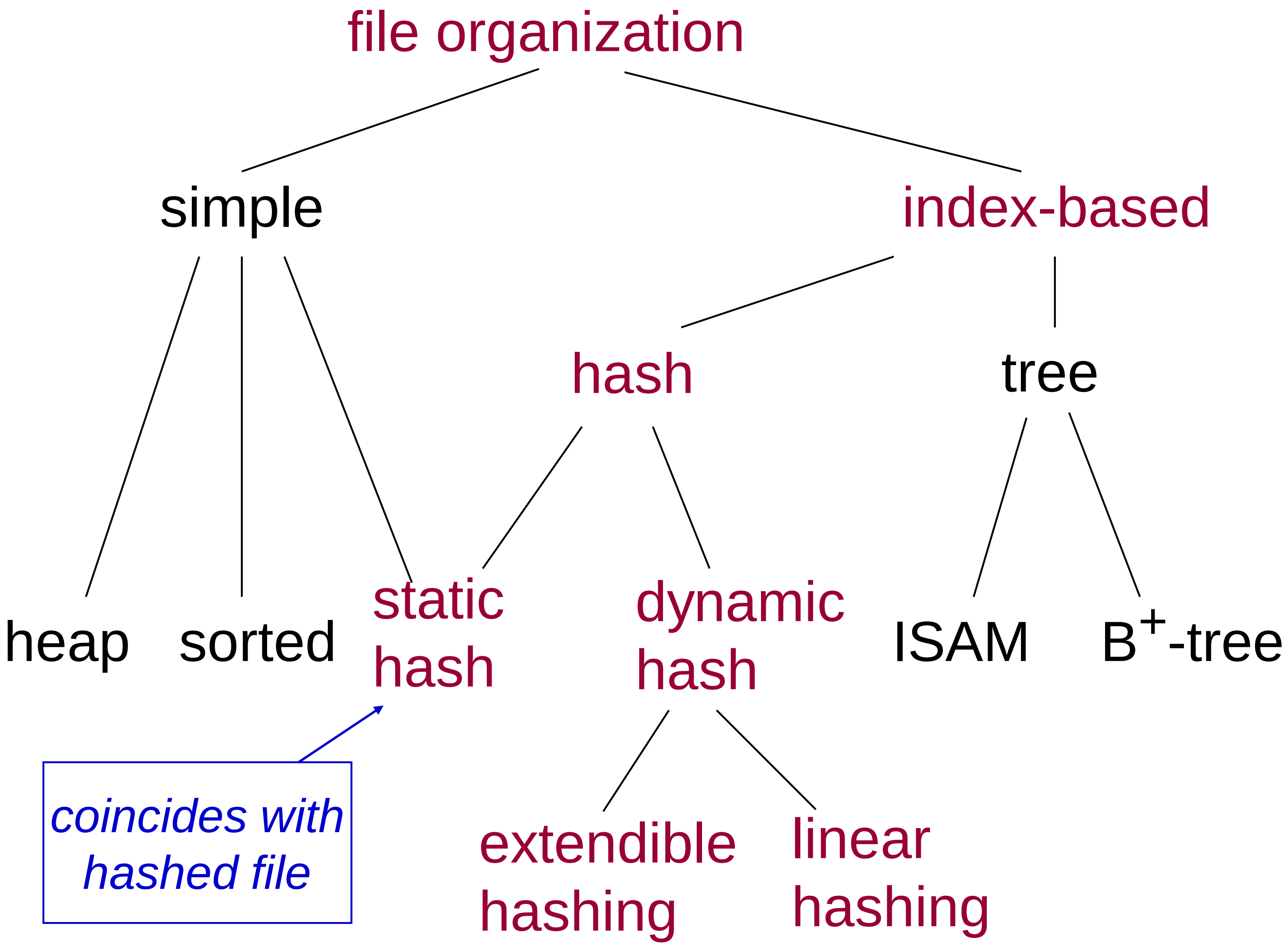
  – modify/write the pages that have been modified in the index and in the file: 2(C+D)

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 leaves. Here are some observations related to the issue of figuring out the number of leaves.

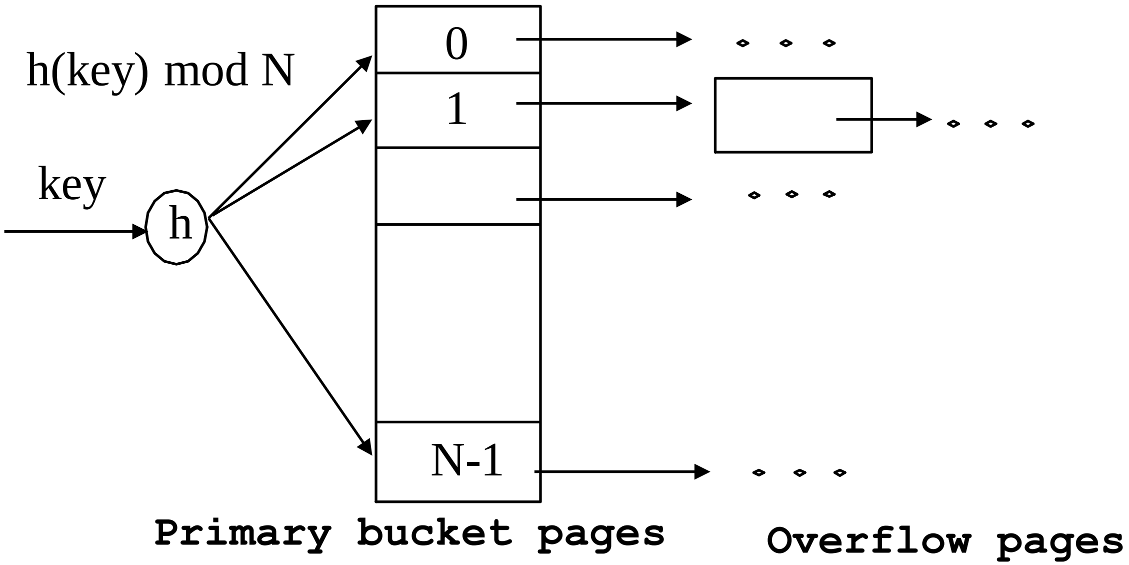
- The pages in the leaves are occupied at the 67%. This means that the physical pages are 50% more than the numero of "required leaves", i.e., the pages strictly required to store the data entries.
- When we use alternative 1, then the number of required leaves is the number of pages in the data file (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 and is secondary non unique, 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.

Hashed index organization



Static Hashing (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** should distribute the values uniformly into the range 0..N-1



Static Hashing (Cont.)

- The buckets contain the *data entries*
- The Hash function should distribute the values uniformly into the range 0..N-1
  - Usually, **h**(*key*) = (a × *key* + b), with a,b constants
  - There are techniques for choosing good hash functions **h**
- In static hashing, the number of buckets never changes. With insertions and deletions, this means long overflow chains, which are a problem for efficiency
  - *Extendible* and *Linear Hashing*: techniques for dealing with this problem

Static Hashing - detailed analysis (1)

- We assume alternative (2) and that the data file is a heap file.
- We assume that each data entry is a 1/10 the size of a data record. Also, as usual in static hashed files, we assume that pages are kept at about 80% occupancy, to minimize overflows as the file expands. Therefore, the number of pages required to store data entries is 1.25(0.10 B) = 0.125 B. The number of data entries that fit on a page is 10 (0.80 R)= 8R

➢ Scan: 0.125 B(D+8RC) + BR(D+C)

– Scan of all index data entries: 0.125 B(D+8RC)

– Every data entry can point to a different data file page: BR(D+C)

→High cost: no one ever scans a hash index!

Static Hashing - detailed analysis (1)

- Selection based on equality: H + D + D + 0.5(8R)C = H + 2D + 4RC
  - H is the cost of identifying the page through the hash function
  - We assume no overflow pages
  - We assume to find the (single) data entry after scanning half of a page
  - We also have to fetch the data record. How many pages we have to access depends on whether the index is clustering or not. If the index is clustering (remember that in this case this means that the data records with a given value of the search key are stored in as few pages as possible – probably one if their number is not high), then we will have one or a few accesses (in any case, as few as possible). If the index is non clustering, then we might even have one page access for each distinct data records with that value for the search key.

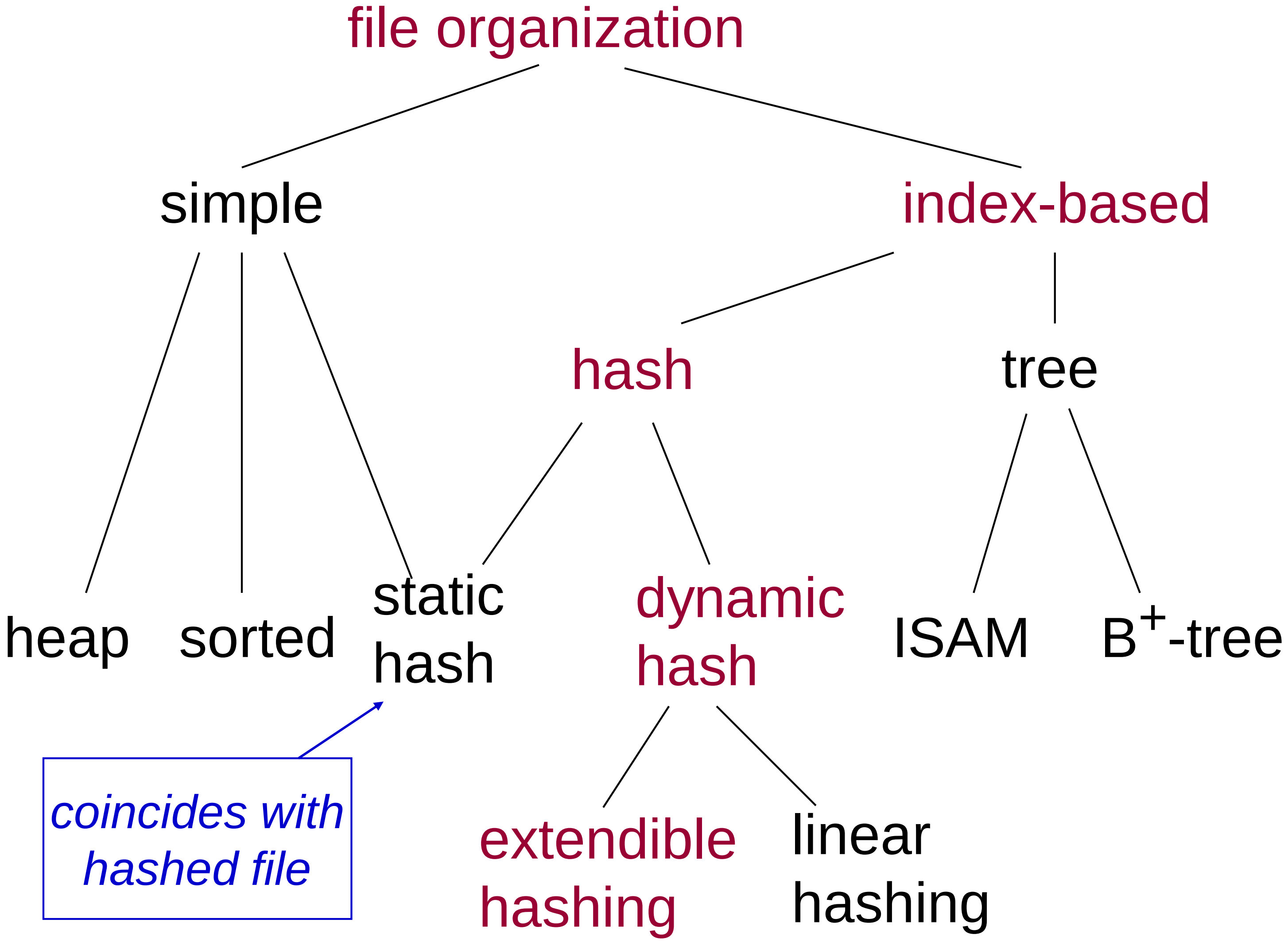
Static Hashing - detailed analysis (2)

- Selection based on range: B(D+RC)
  - No use of index! In this case, even if the index is clustered (the data records with a given value of the search key are stored in as few pages as possible) the index does not help!

- Insertion: 2D + C + H + 2D + C = H + 4D + 2C
  - Cost of inserting the data record in the heap (2D + C)
  - Cost of inserting the data entry in the index (H + 2D + C)

- Deletion: H + 2D + 4RC + 2D = H + 4D + 4RC
  - Cost of locating the data record and the data entry (H + 2D + 4RC)
  - Cost of writing the modified data page and index page (2D)

Hashed index organization



file organization

simple

index-based

hash

tree

heap

sorted

static hash

dynamic hash

ISAM

B^+-tree

coincides with hashed file

extendible hashing

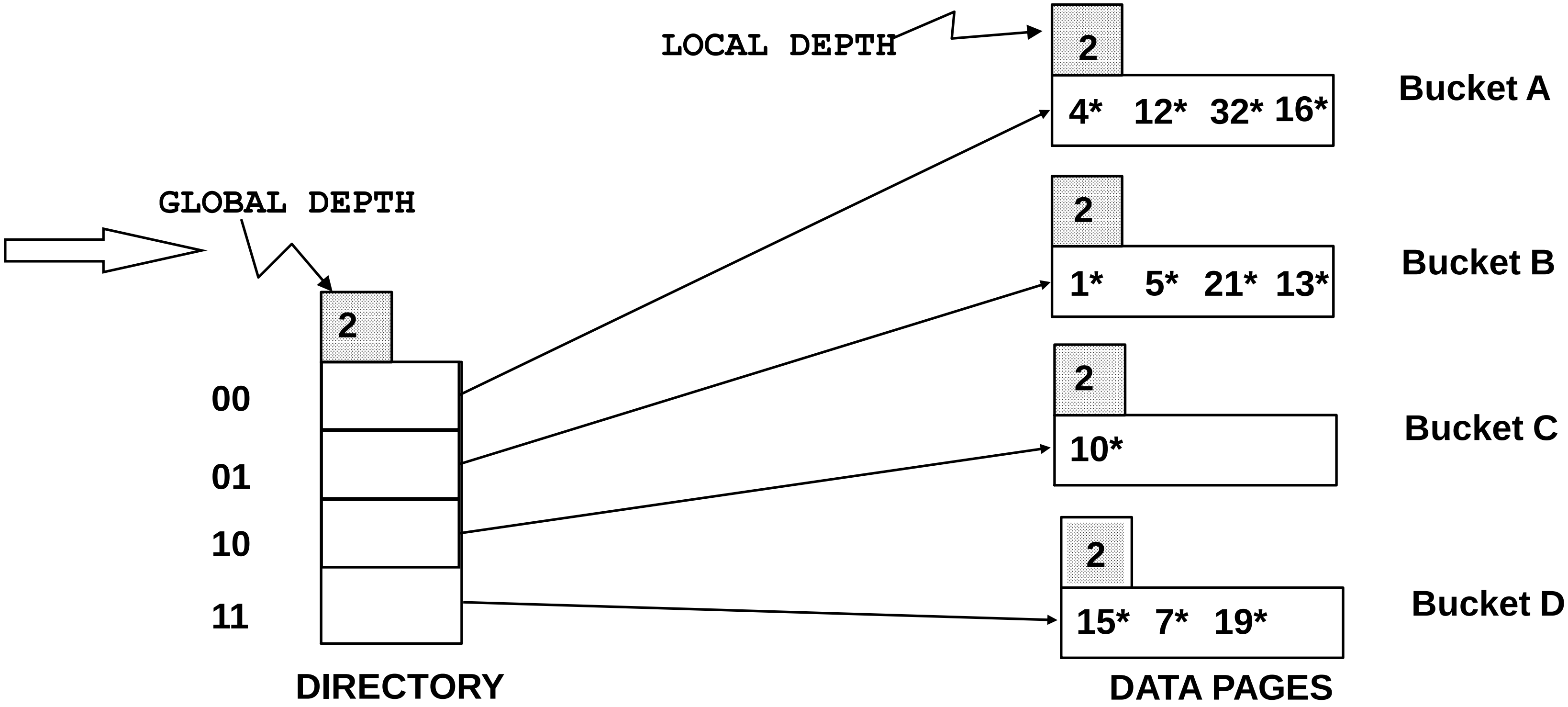
linear hashing

Extendible Hashing

- When a bucket (primary page) becomes full, why not re-organizing the file by doubling the number of buckets?
  - Reading and writing all the pages is too costly
  - *Idea*: use a *directory of pointers to buckets*, double only such directory, and split only the bucket that is becoming full
  - The directory is much smaller than the file, and doubling the directory is much less costly than doubling all buckets
  - Obviously, when the directory is doubled, the hash function has to be adapted

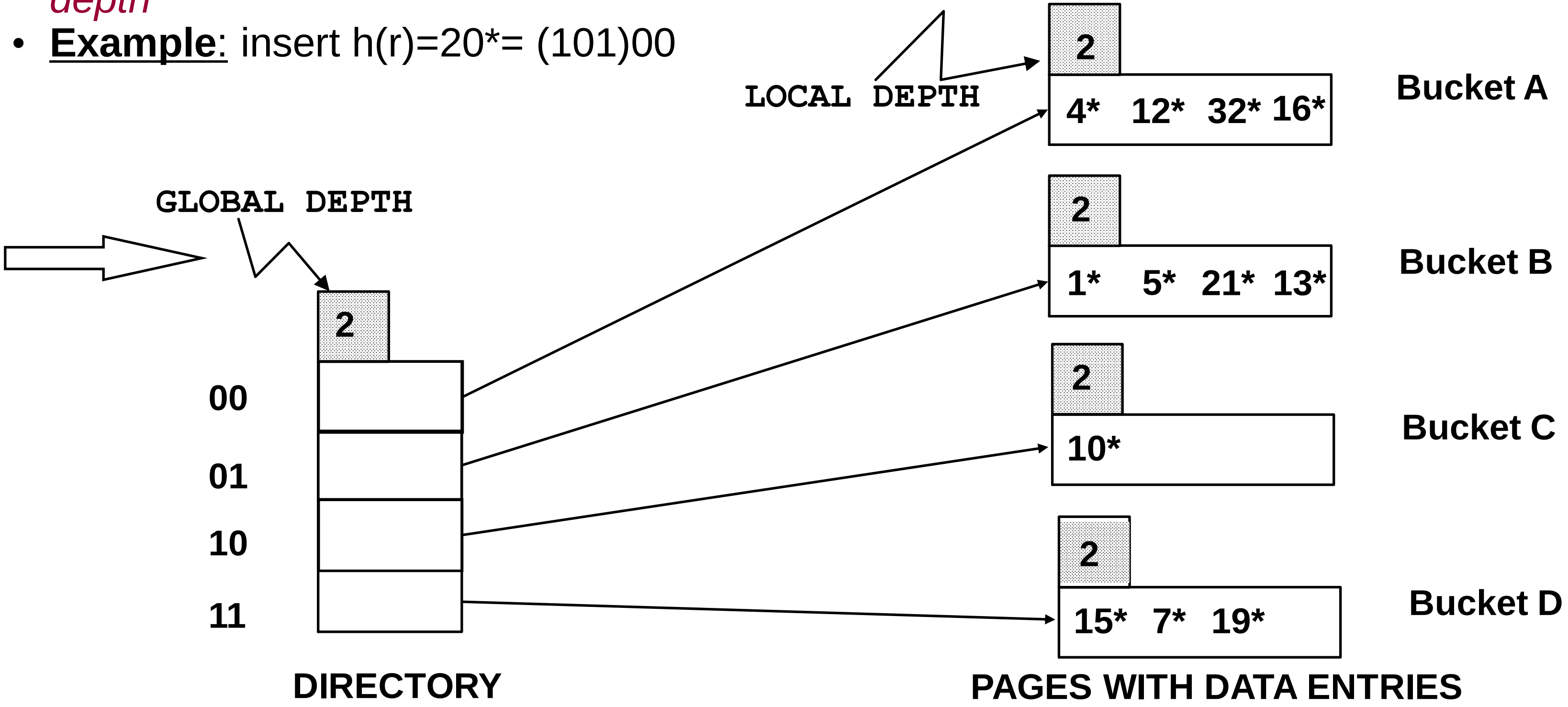
Example

- The directory is an array of 4 items -- in the picture, k is denoted by h(k)*
- To find the bucket for *k*, consider the last g bits of **h**(*k*), where g is the *global depth* (note that, with this method, the buckets for k_1 and k_2 can be the same, even if k_1 and k_2 do not collide -- i.e., even if h(k_1) ≠ h(k2))
- If **h**(*k*) = 5 = binary 101, then 5* is in the bucket pointed by 01

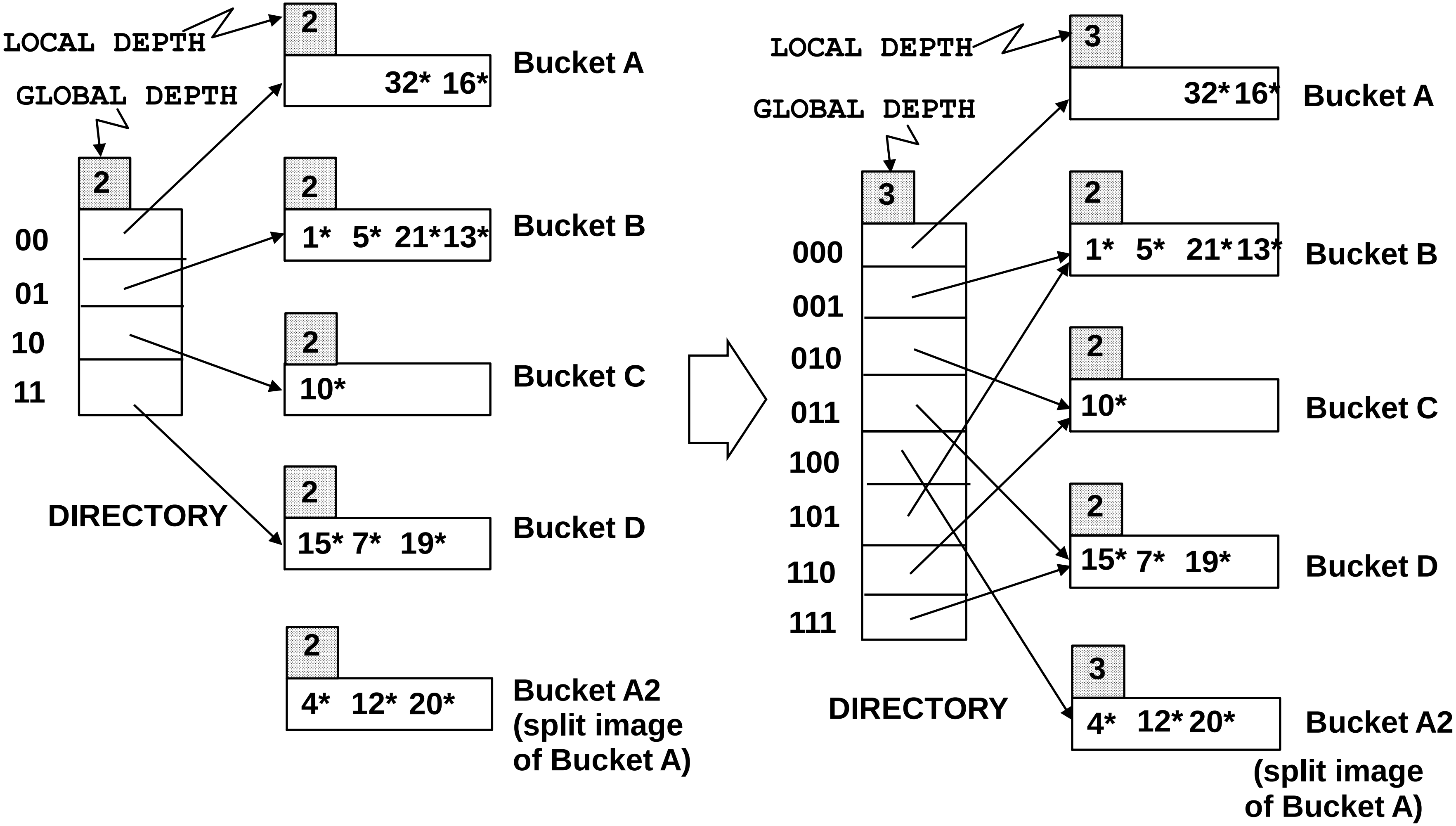


Insertion of a data entry

- **Insertion**: If the bucket B is full, but the search key value k is such that h(k) is different from at least one entry in B in the last c+1 bits (where c is the local depth), then split it, and re-distribute the records in B and its split image according to the same hash function h, but using c+1 bits (in the example, c+1=3 bits!)
- If needed, double the directory. In particular, the decision on whether to double or not the directory is based on comparing the *global depth* and the *local depth of the bucket to be split: we should double the directory if we split and the global depth is equal to the local depth*
- **Example**: insert h(r)=20*= (101)00



Insert h(r)=20* (double the directory)



Observations

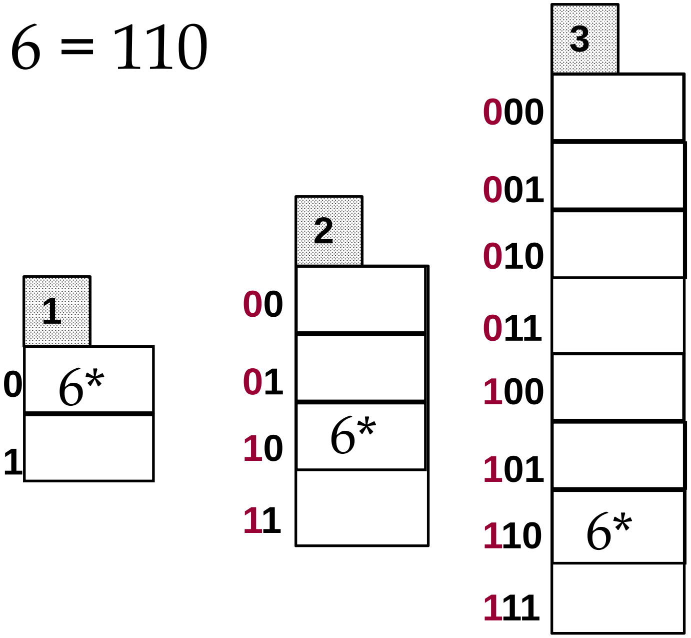
- 20 = binary 10100. The last 2 bits (00) tell us that r belongs to either A or A2. We need another bit to decide which is the right one
  - *Global depth of the directory*: Maximum number of bits needed to decide which is the bucket of a given entry
  - *Local depth of a bucket*: Number of bits used to decide whether an entry belongs to a bucket
- When is a bucket split?
  - When inserting, if the bucket is full and we can distribute the data entries in two buckets using c+1 bits (where c is the local depth)
- When is the directory doubled?
  - When inserting, if the bucket is split, and if *local depth of the bucket = global depth,* then insertion would make *local depth > global depth*; it follows that the directory must be doubled in this case. Doubling the directory means copying it, and then setting up the pointers to the split image

Observations

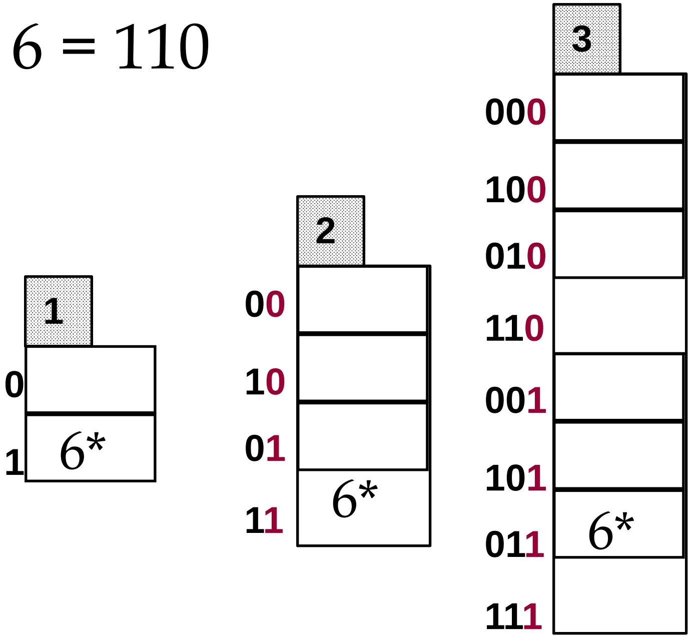
- Initially, all local depths are equal to the global depth, which is the number of bits needed to express the total number of initial buckets
- We increment the global depth g by 1 each time the directory doubles
- Whenever a bucket is split, we increment by 1 the local depth of the two resulting buckets
- If a bucket ha local depth c, the hash values of the data entries in it agree on the last c bits, and no data entries in any other bucket has a has value with the same last c bits
- At most 2^{g-c} directory elements point to a bucket with local depth c (if g=c, then exactly one directory element points to the bucket, and splitting such a bucket requires doubling the directory)

Doubling the directory

We use the least significant bits, because in this way we double the directory simply by copying (using the most significant bits, we could not simply do it by copying)



Least significant bits



Most significant bits

Extendible Hashing: observations

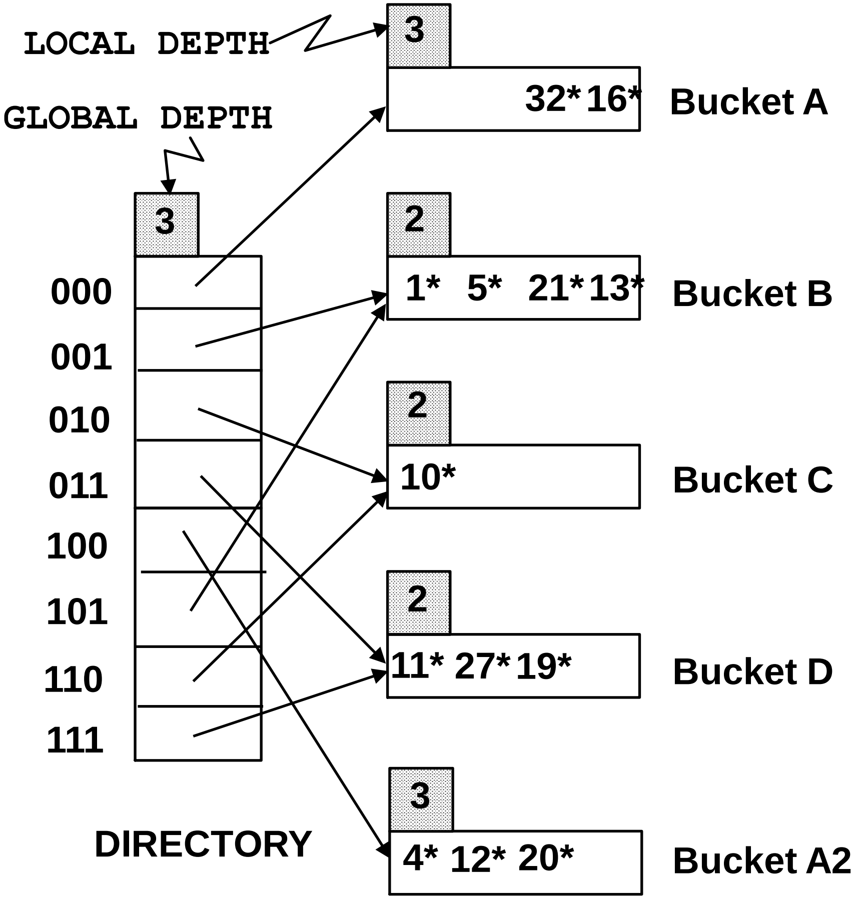
- If the directory fits in main memory (buffer), we need one access for the search with equality condition, otherwise we need two
  - Example. File of size 100MB, 100 bytes per record, page of size 4KB
    - 1.000.000 records (data entries)
    - 40 records per page
    - 25.000 entries in the directory
  - If the distribution of the hash function values is not uniform, then the size of the directory can be very large
  - Collisions (k_1 and k_2 such that $h(k_1) = h(k_2)$) may still require overflow pages (a collision in a full bucket cannot be handled with the split image!)

- **Deletion**: If the deletion of a data entry makes a bucket empty, then it can be merged with its split image. Additionally, if every entry in the directory points to the same record as its split image, then we can halve the directory

Exercise 6

Insert k1, k2, k3, k4, k5

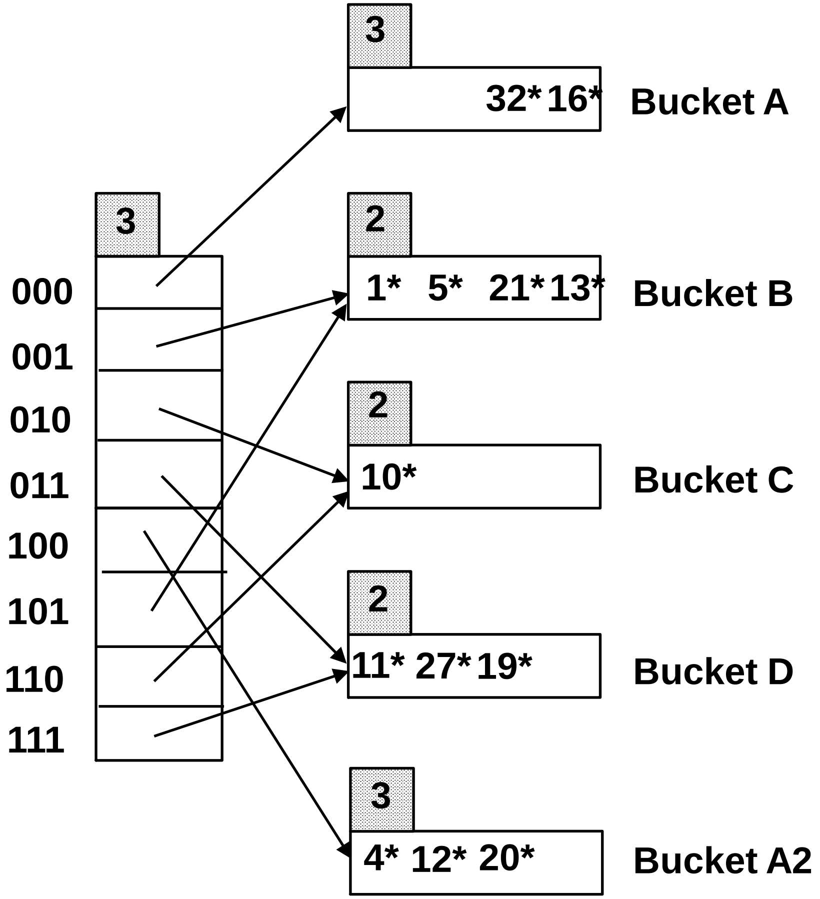
h(k1) = 011011 = 27
h(k2) = 011100 = 28
h(k3) = 011011 = 27
h(k4) = 111100 = 60
h(k5) = 000111 = 7



Exercise 6 - solution

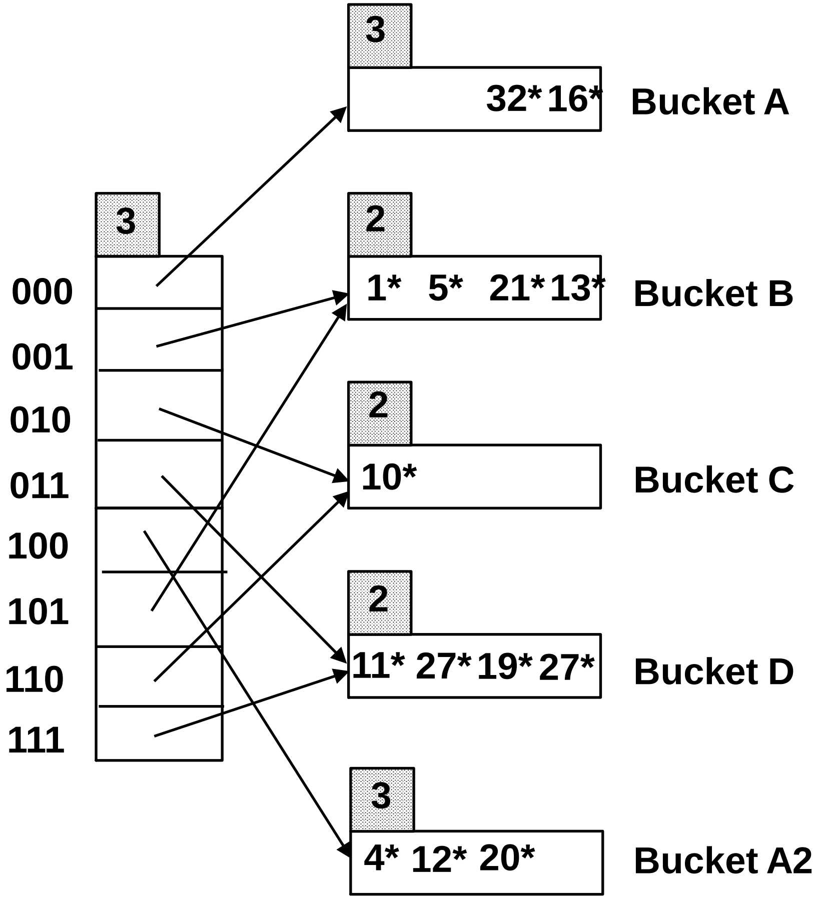
Insert k1

h(k1) = 011011 = 27
h(k2) = 011100 = 28
h(k3) = 011011 = 27
h(k4) = 111100 = 60
h(k5) = 000111 = 7



Exercise 6 - solution

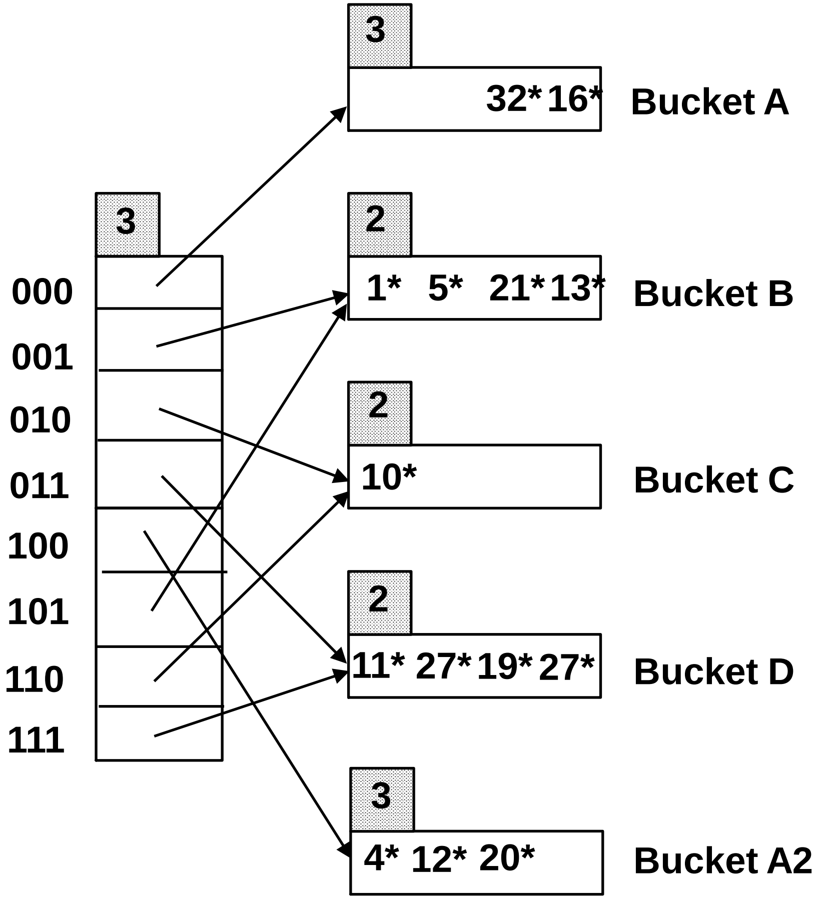
h(k1) = 011011 = 27
h(k2) = 011100 = 28
h(k3) = 011011 = 27
h(k4) = 111100 = 60
h(k5) = 000111 = 7



Exercise 6 - solution

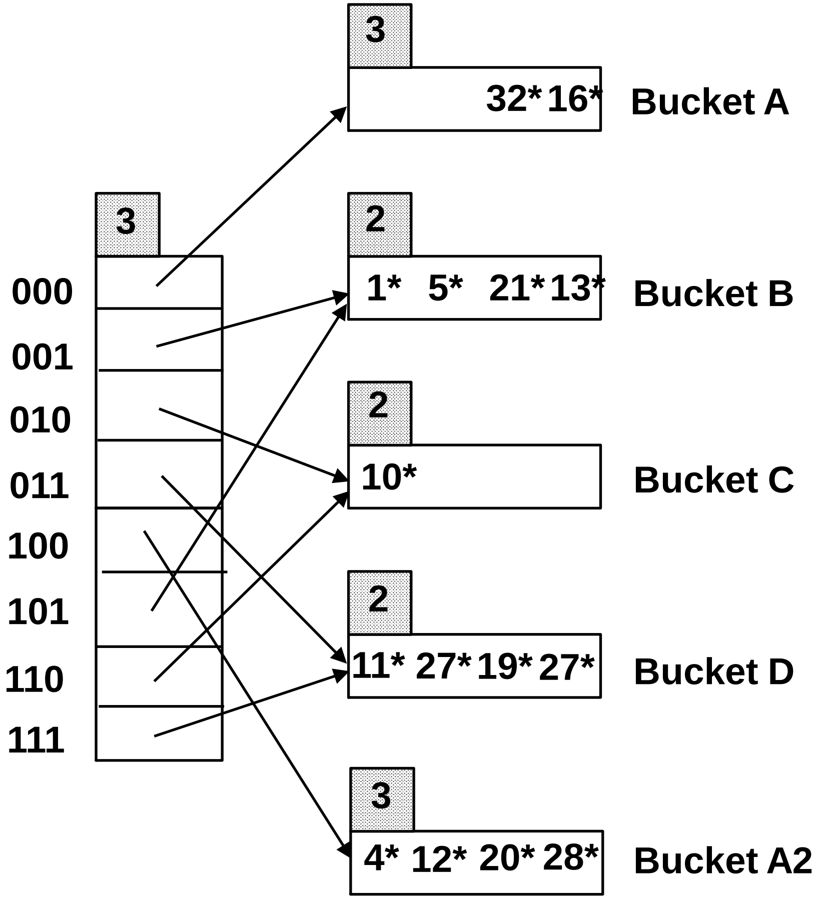
Insert k2

h(k1) = 011011 = 27
h(k2) = 011100 = 28
h(k3) = 011011 = 27
h(k4) = 111100 = 60
h(k5) = 000111 = 7



Exercise 6 - solution

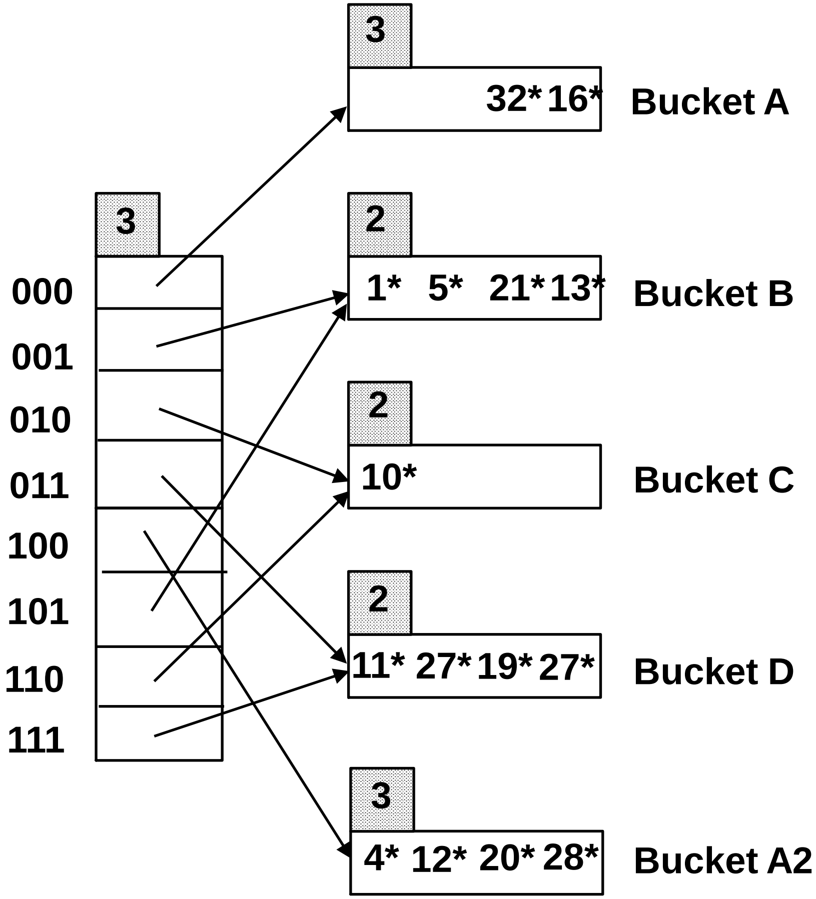
h(k1) = 011011 = 27
h(k2) = 011100 = 28
h(k3) = 011011 = 27
h(k4) = 111100 = 60
h(k5) = 000111 = 7



Exercise 6 - solution

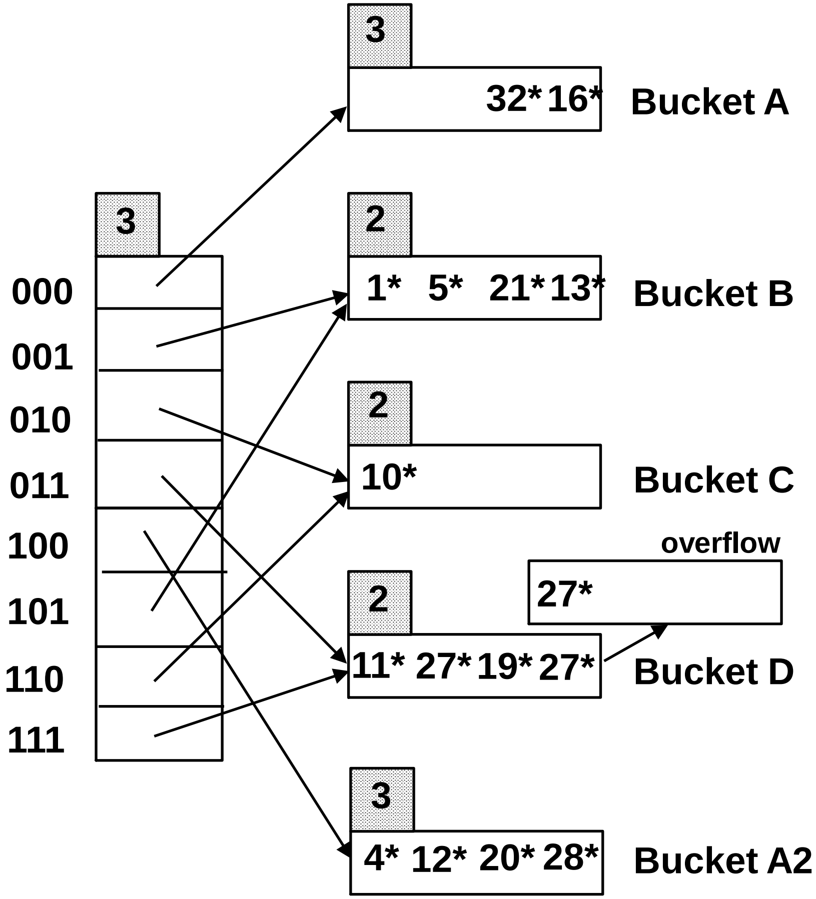
Insert k3

h(k1) = 011011 = 27
h(k2) = 011100 = 28
h(k3) = 011011 = 27
h(k4) = 111100 = 60
h(k5) = 000111 = 7



Exercise 6 - solution

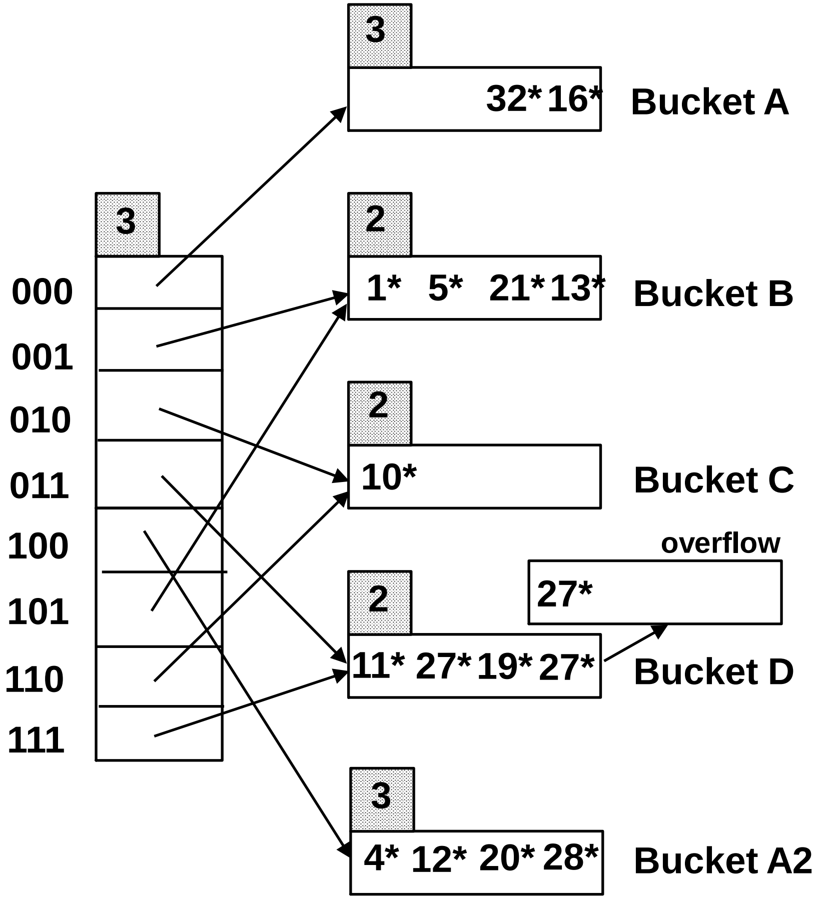
h(k1) = 011011 = 27
h(k2) = 011100 = 28
h(k3) = 011011 = 27
h(k4) = 111100 = 60
h(k5) = 000111 = 7



Exercise 6 - solution

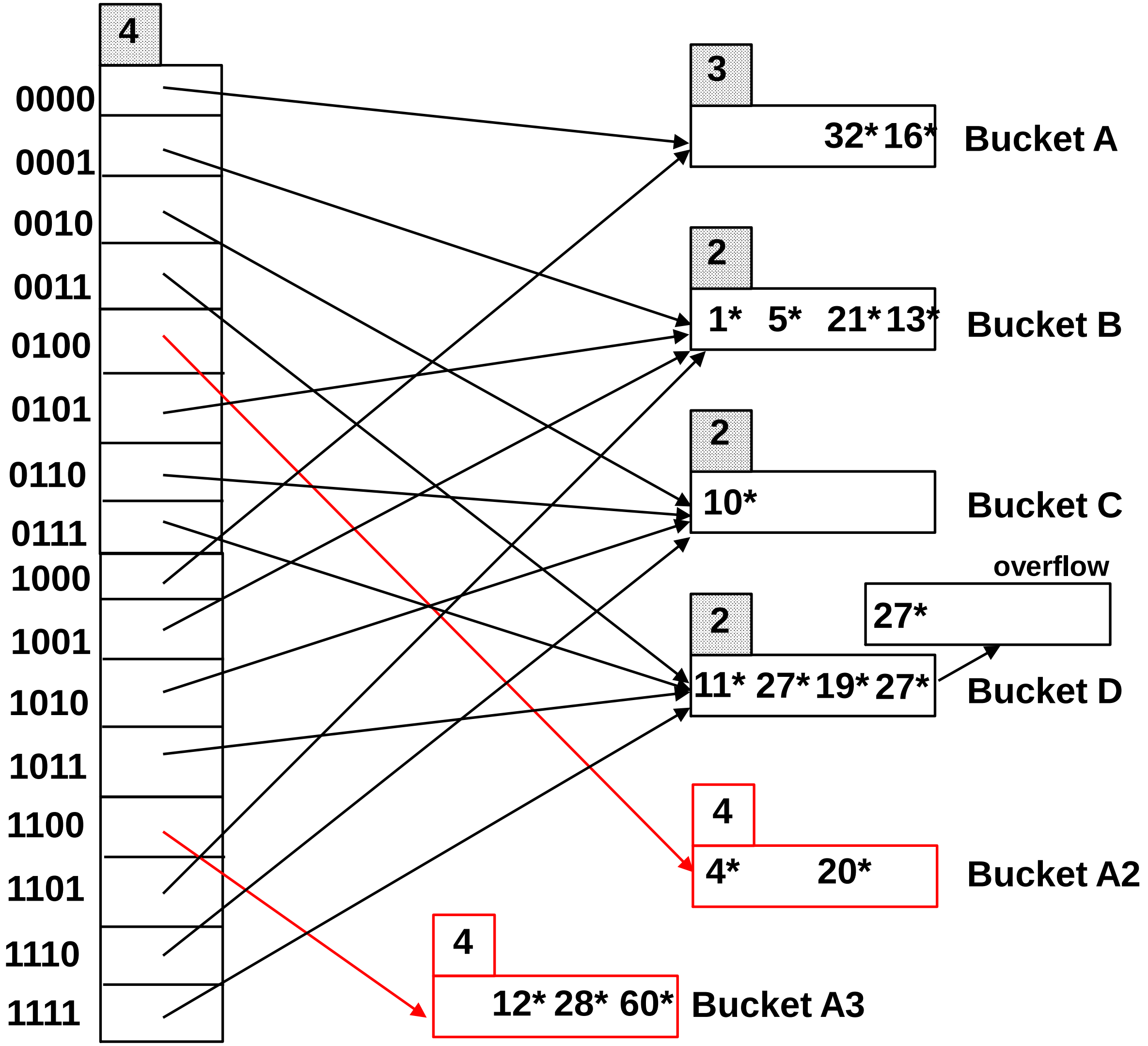
Insert k4

h(k1) = 011011 = 27
h(k2) = 011100 = 28
h(k3) = 011011 = 27
h(k4) = 111100 = 60
h(k5) = 000111 = 7



Exercise 6 - solution

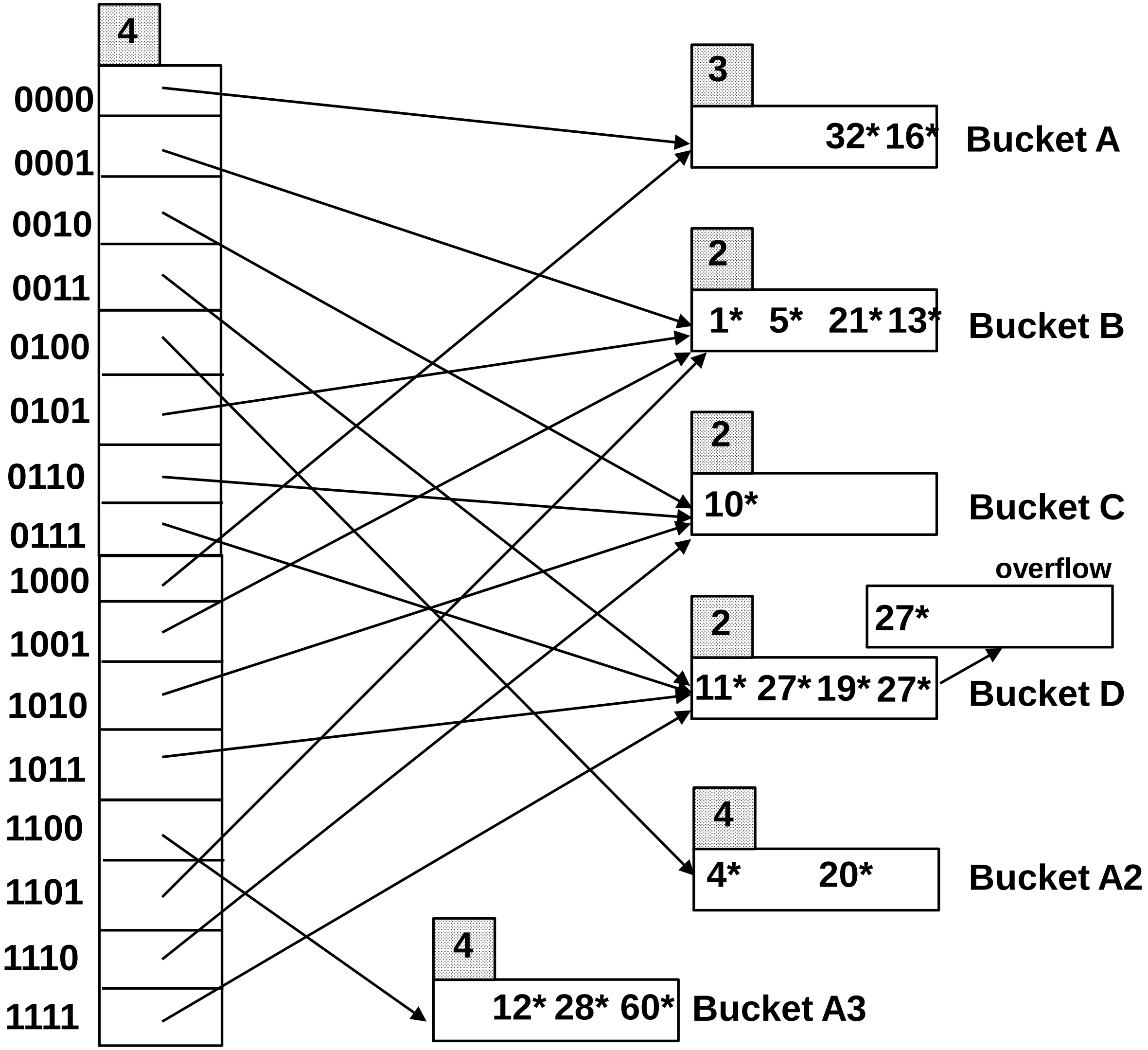
h(k1) = 011011 = 27
h(k2) = 011100 = 28
h(k3) = 011011 = 27
h(k4) = 111100 = 60
h(k5) = 000111 = 7



Exercise 6 - solution

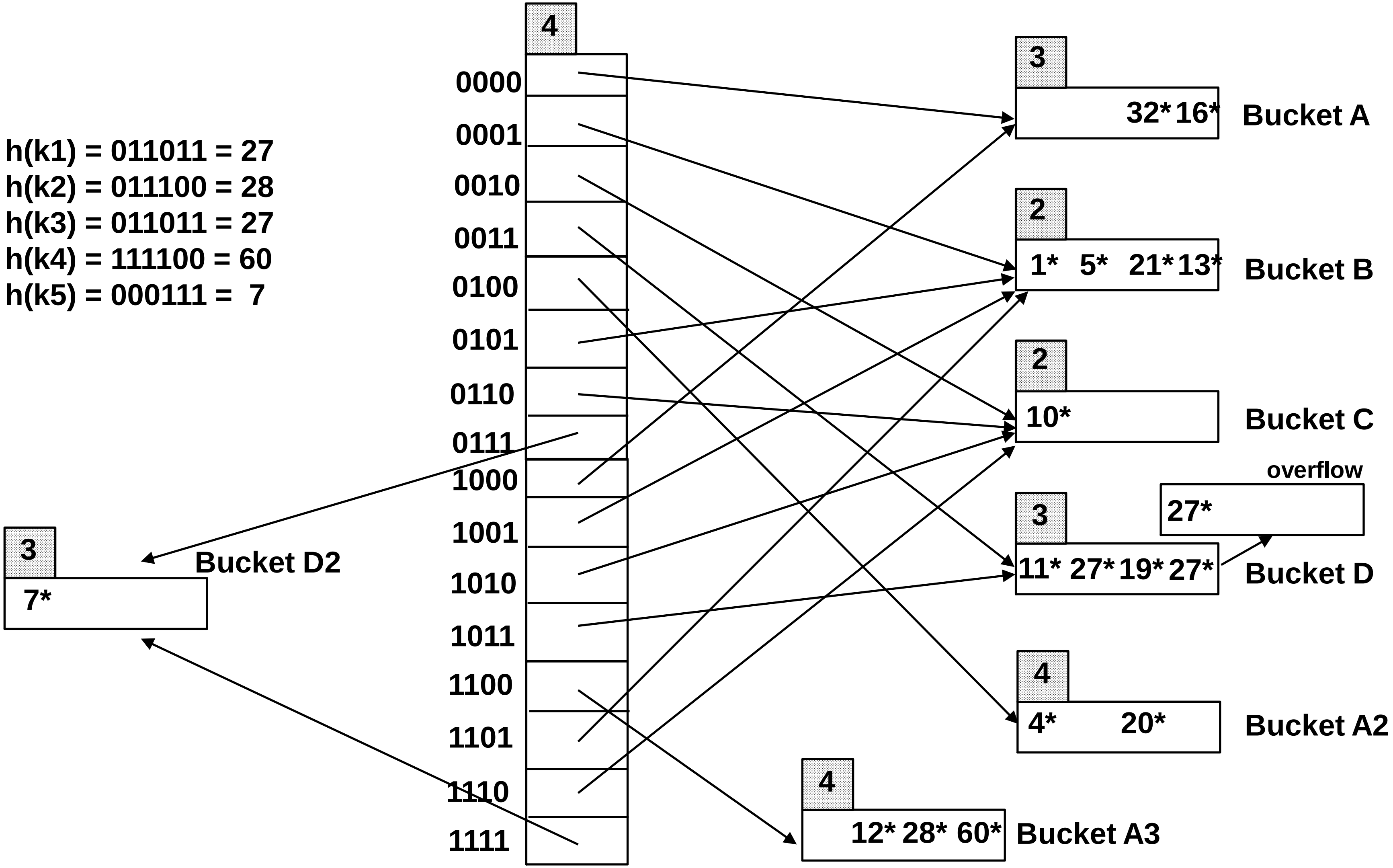
Insert k5

h(k1) = 011011 = 27
h(k2) = 011100 = 28
h(k3) = 011011 = 27
h(k4) = 111100 = 60
h(k5) = 000111 = 7



Exercise 6 - solution

h(k1) = 011011 = 27
h(k2) = 011100 = 28
h(k3) = 011011 = 27
h(k4) = 111100 = 60
h(k5) = 000111 = 7



4

0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111

3
32* 16*
Bucket A

2
1* 5* 21* 13*
Bucket B

2
10*
Bucket C

overflow
27*

3
11* 27* 19* 27*
Bucket D

4
4* 20*
Bucket A2

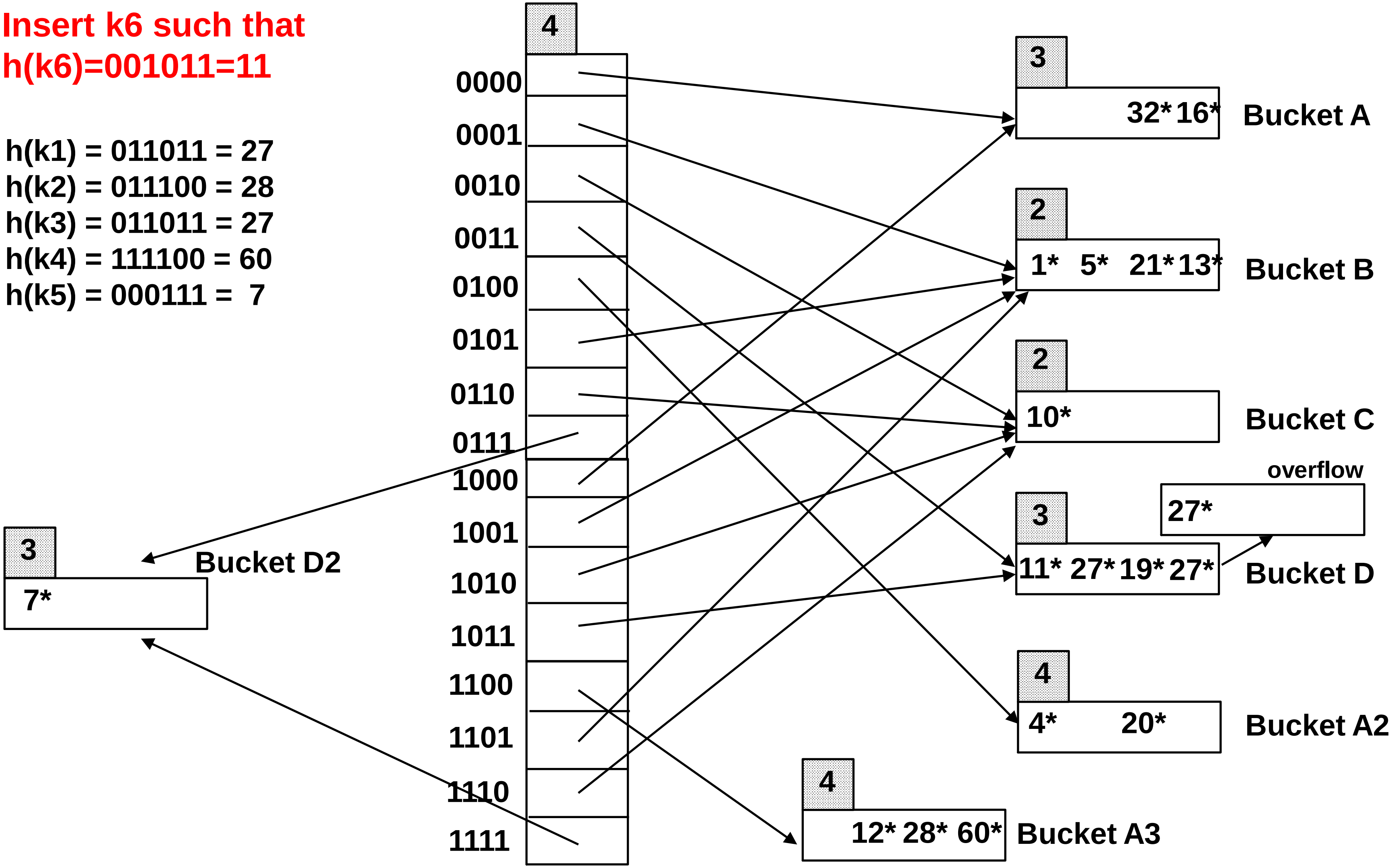
4
12* 28* 60*
Bucket A3

3
7*
Bucket D2

Exercise 7

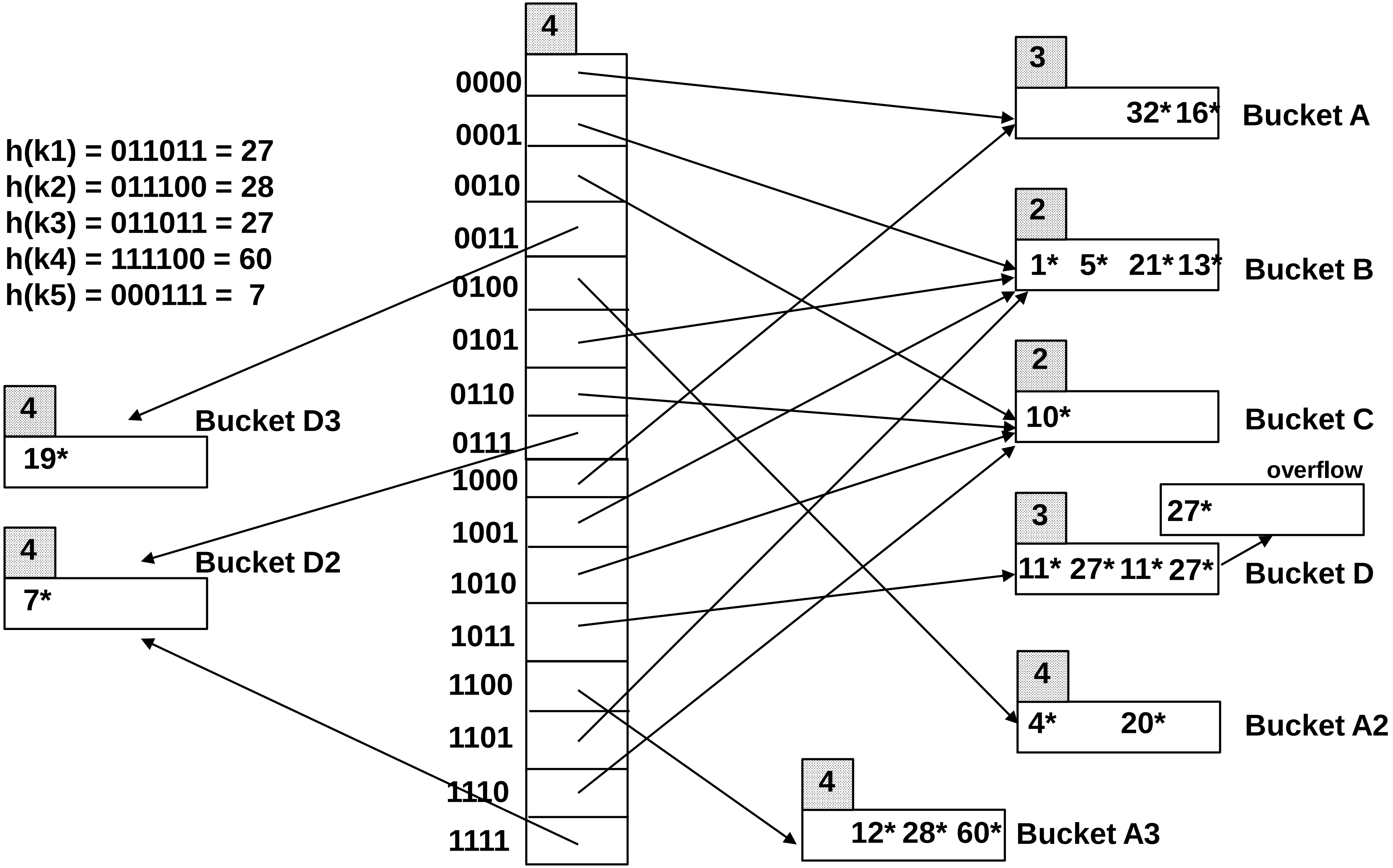
Insert k6 such that h(k6)=001011=11

h(k1) = 011011 = 27
h(k2) = 011100 = 28
h(k3) = 011011 = 27
h(k4) = 111100 = 60
h(k5) = 000111 = 7



Exercise 7 - solution

h(k1) = 011011 = 27
h(k2) = 011100 = 28
h(k3) = 011011 = 27
h(k4) = 111100 = 60
h(k5) = 000111 = 7



Comparing different file organizations

- In general
  - We base our comparison on the cost of simple operations
  - We do not consider the cost of operations on main memory data
  - For search based on equality, we assume that the number of records satisfying the equality is such that all such records fit in one page
- Heap files
  - We ignore the cost of space management (it is infrequent that space management tasks are needed)
- Sorted file
  - In "search based on equality", we assume that the equality condition matches the sort order (i.e., it is on at least the first field of the composite key)

The cost for different file organizations

- Clustered tree-based index
  - We assume that alternative (1) is used
  - We assume that the search operations refers to at least the first component of the search key
  - We ignore the cost of keeping the tree balanced (it is infrequent that this needs occur)
- Unclustered tree-based index
  - We assume that the search operations refers to at least the first component of the search key
  - We ignore the cost of keeping the tree balanced (it is infrequent that this needs occur)
- Unclustered static hash-based index
  - We assume that the search operations refers to at least the first component of the search key
  - We assume that there are no overflow chains
  - The analysis for dynamic hash-based is similar (the problem of overflow chains is irrelevant, although we have higher average cost for search)

Summary

File Organization	*Scan*	*Search based on equality*	*Search based on range*	*Insertion*	*Deletion*
Heap file	**BD**	**BD**	**BD**	**2D** **(we ignore time for space management)**	**Cost of search + D (we ignore time for space management)**
Sorted File	**BD**	**D $\log_2$B**	**D $\log_2$B + # of further pages**	**Cost of search + 2BD**	**Cost of search + 2BD**
Clustered tree-based index (alternative 1)	**1.5BD**	**D $\log_F$(1.5B)**	**D $\log_F$(1.5B) + #of further pages**	**Cost of search + D**	**Cost of search + D**
Unclustered tree-based index (alternative 2)	**BD (R +0.15)**	**D($\log_F$(0.15B) + #of further records)**	**D($\log_F$(0.15B) + #of further records)**	**D(3+ $\log_F$(0.15B))**	**Cost of search + 2D**
Unclustered static hash-based index	**BD(R+ 0.125)**	**D(1 + #of further records)**	**BD**	**4D**	**Cost of search + 2D**

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

Discussion

- 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
  - Inefficient management of insertion and deletion (overflow)

- Dynamic hash index
  - Efficient search based on equality, insertion and deletion
  - Optimal support for search based on equality
  - Inefficient scan and search based on range
  - Good management of insertion and deletion (no overflow)

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