FILE ORGANIZATION

Introduction

- File organization is a critical aspect of data management that determines how data is stored, accessed, and managed across computing systems

- Explores fundamental concepts of file organization, from high-level application perspectives to low-level system implementations

- Investigates key allocation methods, indexing strategies, and the complex mechanisms that transform raw data storage into an efficient, structured ecosystem

Review

- A file is the concept through which a program or an application stores data in memory storage. Files are used at different levels of abstraction with different semantics:
  - Application Level
  - System Level

Review

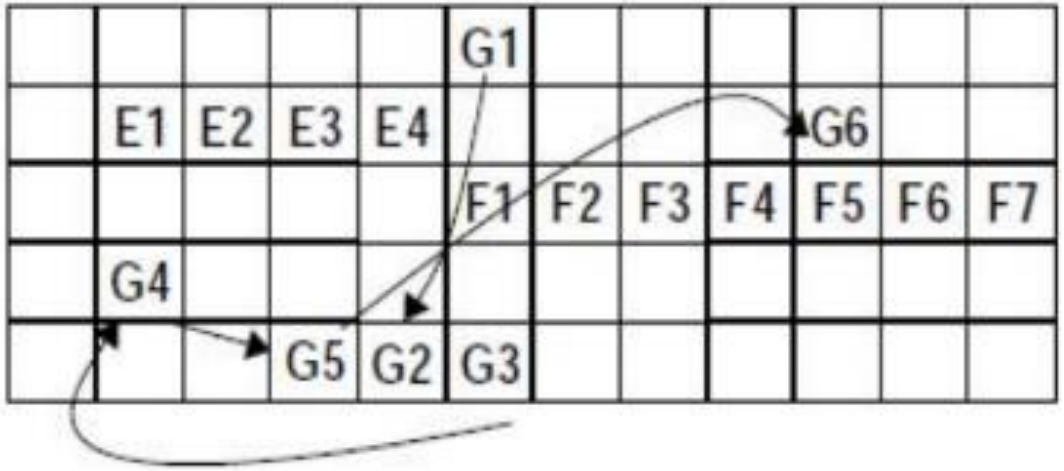
- **Note**
- All operations at the 'application' level (logical level) go through the system, which translates them into low-level operations to physically access the I/O blocks. Since I/O operations are time-consuming, the system maintains in main memory a special, limited-size area (the buffer cache) that allows it to keep copies of selected physical blocks according to certain strategies (for example, the most frequently used ones). This buffer area is completely transparent to the application programs that use the files.

Review

- **Example**
- When an application requests to read a specific record, the system first checks if the concerned block is already in main memory (in the buffer cache). If it is, the sought record will be directly transmitted to the application without any physical read operation.

File Modeling

- Memory storage is modeled as a contiguous area of sequentially numbered blocks (these numbers represent the block addresses). Blocks are contiguous areas of bytes of the same size, containing, among other things, the data (records) of files.



Characteristics and Header Block

- In order for the system to manage a file, it needs to know information about its characteristics: the blocks used by the file, the organization of the file, the associated access rights, etc.

Characteristics and Header Block

- **Example 1**
- For a certain type of file system, block 0 could be reserved to contain a table where each row provides information on the characteristics of a file (name, size, blocks used, etc.). When an application wishes to open a file with a given name, the system retrieves its information from this table.

Characteristics and Header Block

- **Example 2**

- In sequential memory storage such as magnetic tapes, this type of information (the characteristics) was found at the beginning of each file, which is why it was called the "header block".

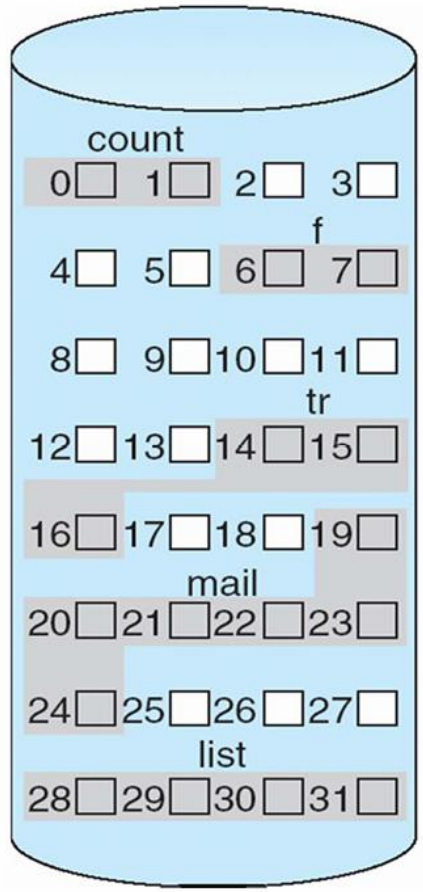
Allocation methods

- An allocation method refers to how disk blocks are allocated for files:
  - Contiguous allocation
  - Linked allocation
  - Indexed allocation

Contiguous Allocation

- Each file occupies a set of contiguous blocks on the disk
- Simple – only starting location (block x) and length (number of blocks) are required
- Random access
- Wasteful of space (dynamic storage-allocation problem)
- File cannot grow

Contiguous Allocation

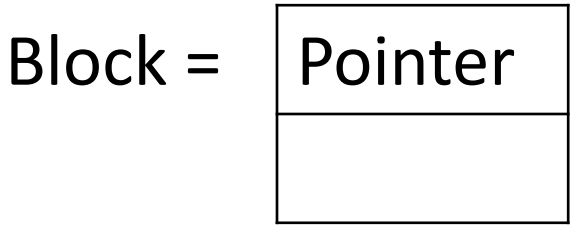


directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

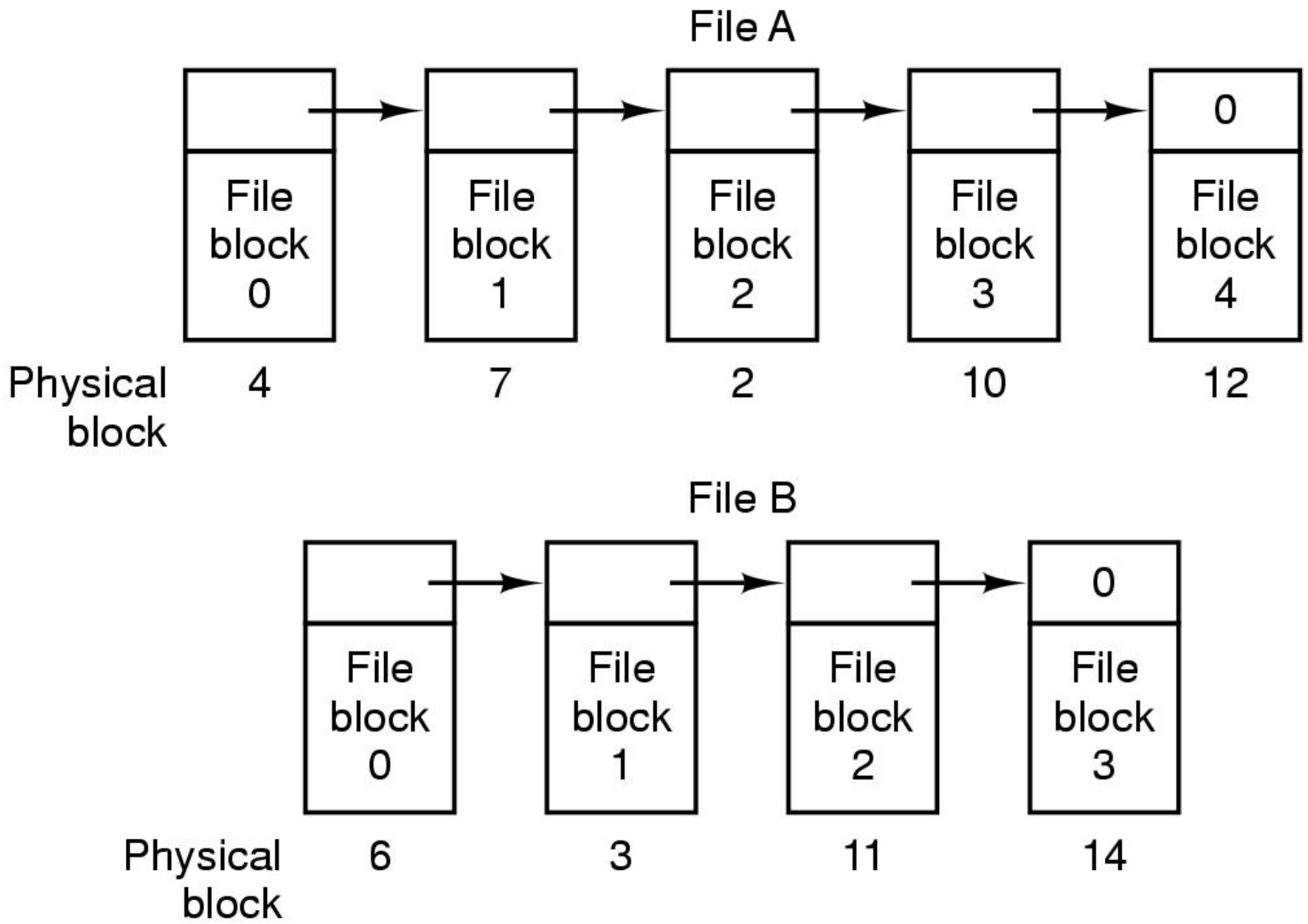
Linked Allocation

- Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk.

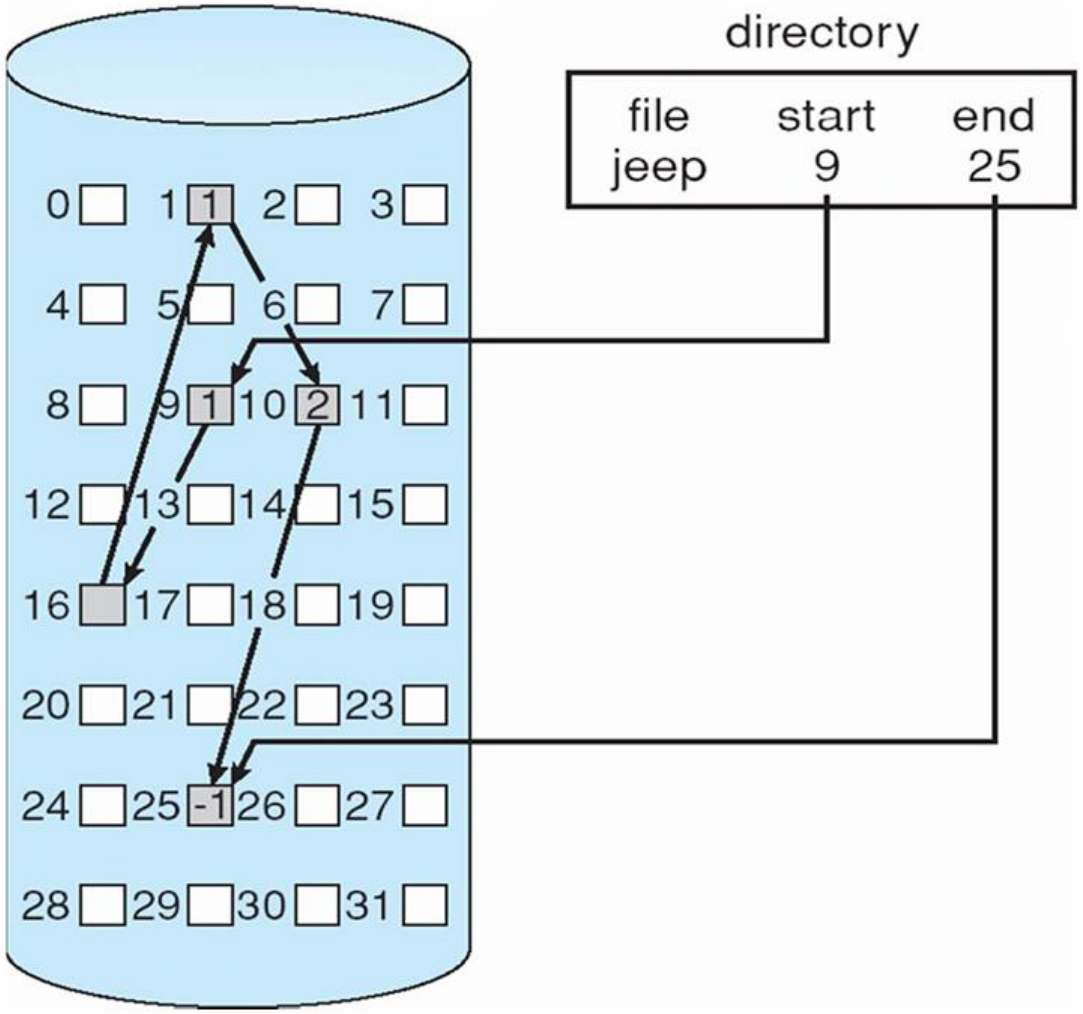


- Simple – need only starting address
- Free-space management system – no waste of space
- No random access
- a space is required for the pointers.

Linked Allocation



Linked Allocation



Indexed Allocation

- Access operations such as search, and insertion can become increasingly inefficient as data grows.
- Indexing methods help enhance performance to some extent by utilizing an auxiliary structure, known as an index table, to speed up searches."

Indexed Allocation

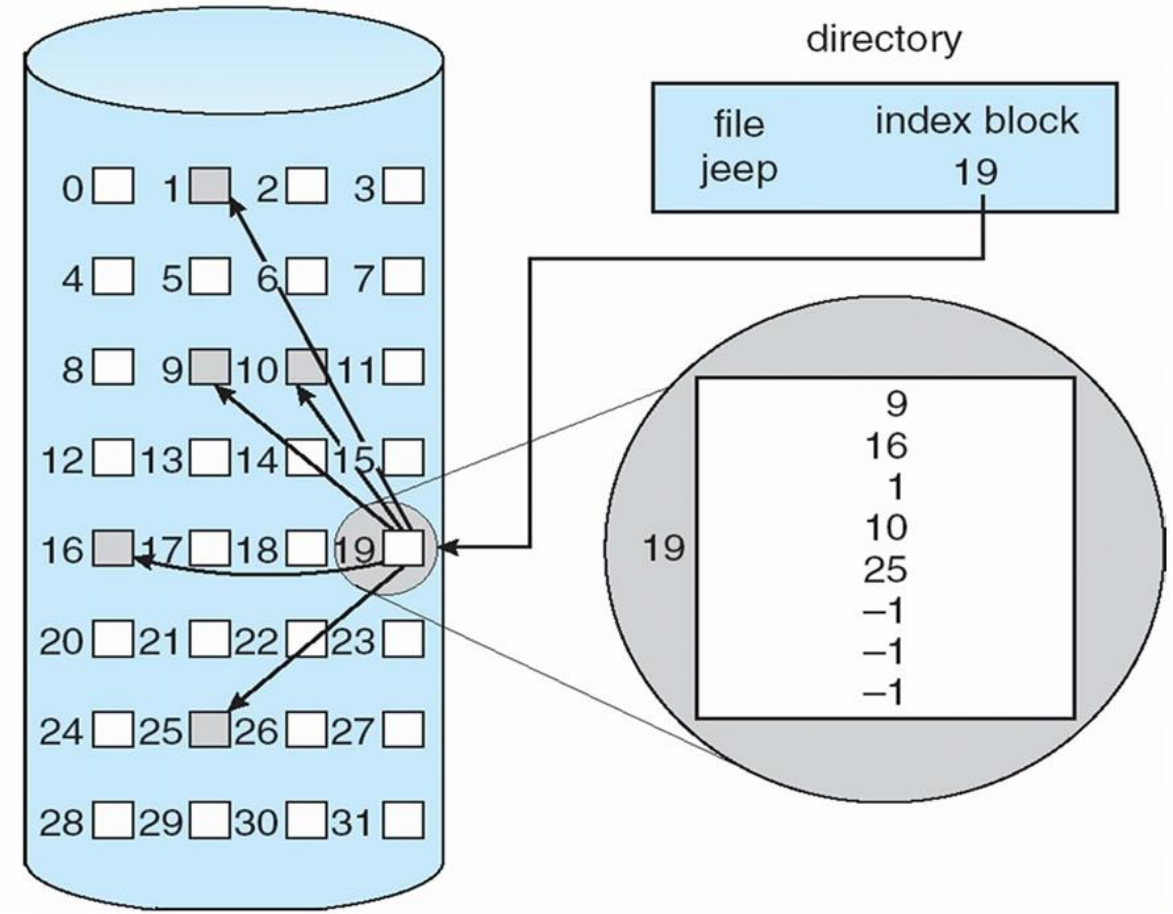
- **Without an Index:**
  - **Sequential Traversal:** Scanning records sequentially, resulting in linear complexity.
  - **Binary Search:** Applicable only if the file is sorted, achieving logarithmic complexity.
- **With an Index:**
  - Traverse the index to locate the desired record, enabling direct access.
  - **Caution:** Updates to the file become more expensive due to the need to maintain the index.

Indexed Allocation

- **Definition:**

- Indexes are created based on one or more fields used to build the index. An index is typically an organized table of <**key, address>** pairs designed to speed up the search process for records in a file.

Indexed Allocation



Indexed Allocation

- **Note:**
- For these methods, two files are used: the *index file* and the *data file*. Typically, the data file is not sorted. It can be an array or a linear linked list of blocks on the disk. The index file is assumed to be in memory and sorted according to the keys of the records. The index can be at one or multiple levels, as we will describe later. The internal organization of the file is arbitrary.

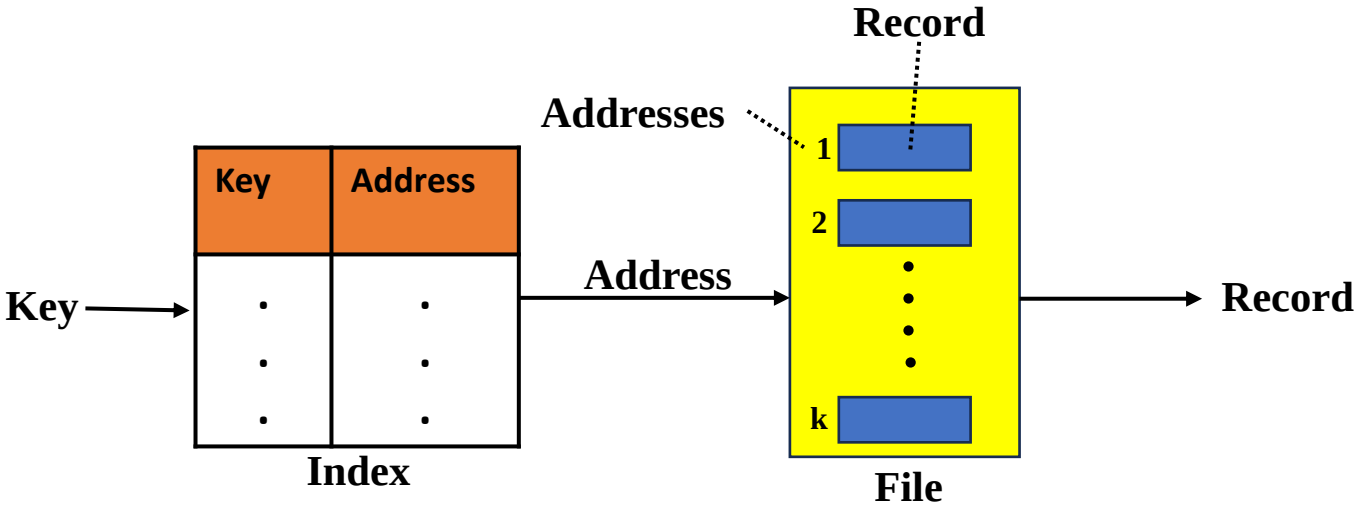
Indexed Allocation

Example 1

- The use of indexes is based on the following observation: to find a book in a library, instead of examining each book one by one (corresponding to sequential search), it is faster to consult the catalog where they are organized by subject, author, and title. Each entry in an index includes a value extracted from the data and a pointer to its original location. An entry can be easily retrieved by searching for its location in the index.

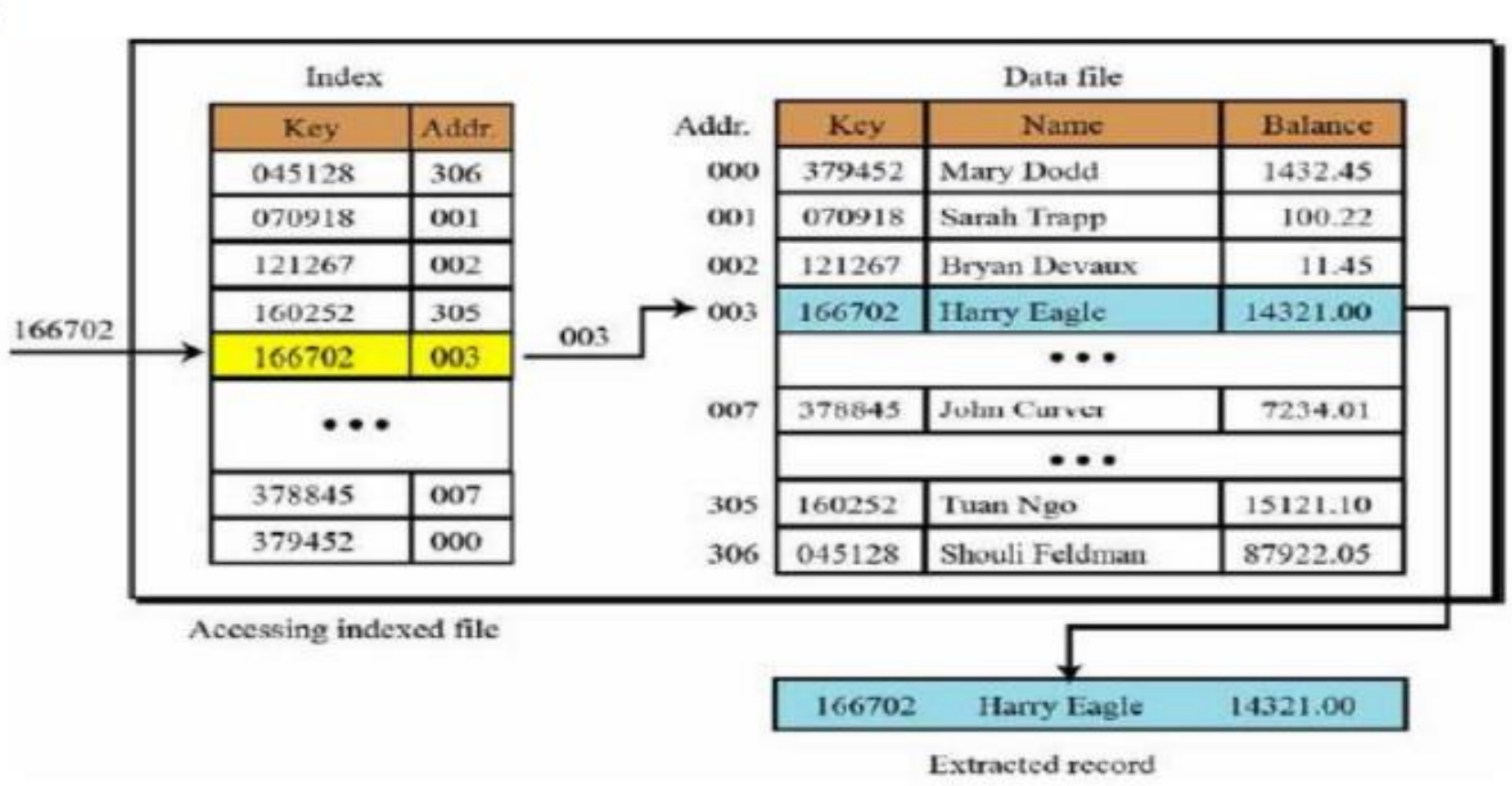
Indexed Allocation

- **Example 2**
- INDEXED FILES
- To access a record in a file randomly, we need to know the address of the record.



Indexed Allocation

- **Example 3**



Advantages of index methods

- Even if the index is not in memory, index methods have the following advantages:
  - Enable binary search even for variable-length records.
  - Performing binary search on the index file is much faster than on the data file itself.
  - Deletion of records using "flags" is done without disk access.

Disadvantages of index methods

- If the index is too large to fit in memory:
  - Binary search becomes expensive.
  - Rearrangement is costly due to shifts.
- In such cases, other techniques are employed, such as trees or hashing.

Index Types based on Key Density

- **Dense:** If it contains all the keys from the data file. In this case, there is no need to keep the file sorted.
- **Sparse:** If it does not contain all the keys from the data file (for example, only one key per block). In this case, the file must be sorted.

Dense Index

- Contains a record for every value of the sorting key in the indexed file.
- Over time, this type of index can occupy a significant amount of memory space.
- However, they have a very short search time.

Sparse Index

- Contains records for some values of the sorting key in the indexed file.

  Sparse indexes have advantages:

  - They occupy **less space**.
  - They impose **fewer constraints during insertions and deletions**.

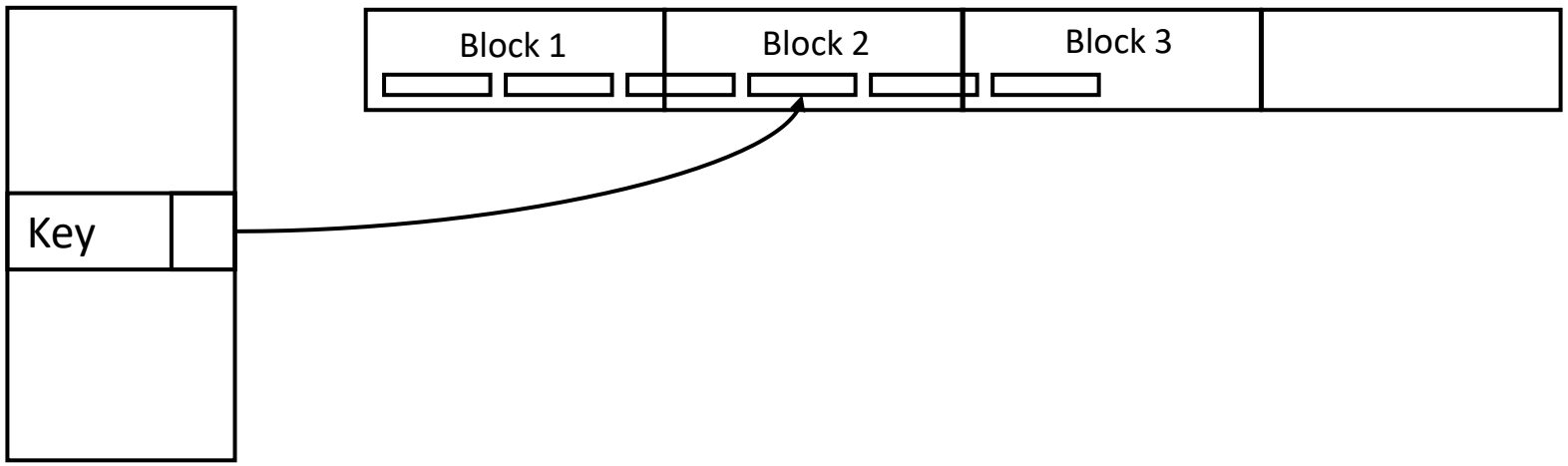
Types based on the nature of the key

Primary Index:

- If the key field does not contain duplicate values, the index is then considered "primary.“
- An index ordered in the same way as the data file, which is sequentially ordered according to a key. (The indexing field is equal to this Key.)

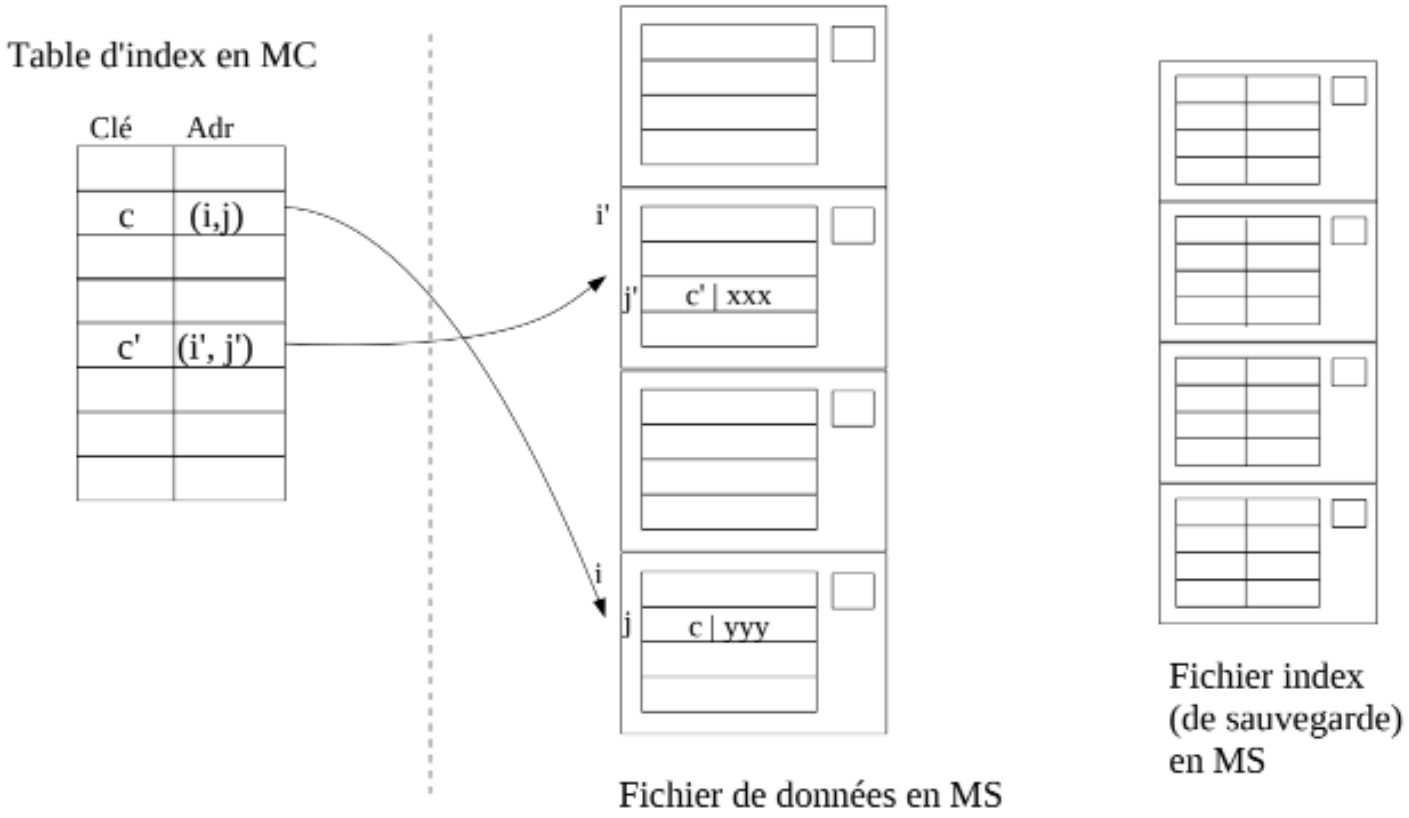
Primary Index at One Level

- The index is a set of pairs (key, address) entirely arranged in main memory. The file is a collection of blocks (in this example, the blocks are contiguous) on the disk. Each block contains a set of records. Records may span two logically consecutive blocks. This is illustrated in the following figure:



Primary Index at One Level

- The data and index files can be of any structure (contiguous blocks, chained blocks, etc.). Similarly, the records can be of fixed or variable format (with or without overlapping).



Basic Operations

- To develop an indexing method, the following basic operations are necessary:
    - Create an index and data file
    - Load the index file into memory before use
    - Search for a record with a given key
    - Insert a record
    - Delete a record
    - Modify a record

Search for a Record with a Given Key

- To search for a record with a given key in the file, start with a binary search on the index.
    - If the key is found, the block containing the record is brought into main memory to retrieve the record.
    - If the record spans two blocks, the second block is also brought in to retrieve the rest of the record.

Insert a Record

- Inserting a record is done as follows:
  - If the key is not found in the index, it is inserted, potentially causing shifts.
  - The record is then inserted at the end of the file.

Delete a Record

- Deletion is typically logical, hence extremely fast.
- Reorganization is then needed to physically eliminate the deleted records.
- Typically, a new file is constructed as part of this process.

Primary Index at Multiple Levels

- If the index is too large to fit in main memory, a second index is built on the ordered index file. In this case, a single key is chosen for each block of the index file (sparse index) to construct the second index.
- If the second index is still too large to fit in main memory, it is stored on disk (second index file), and a third index is built by choosing a key for each block of the second index file. This process can be repeated, as necessary.

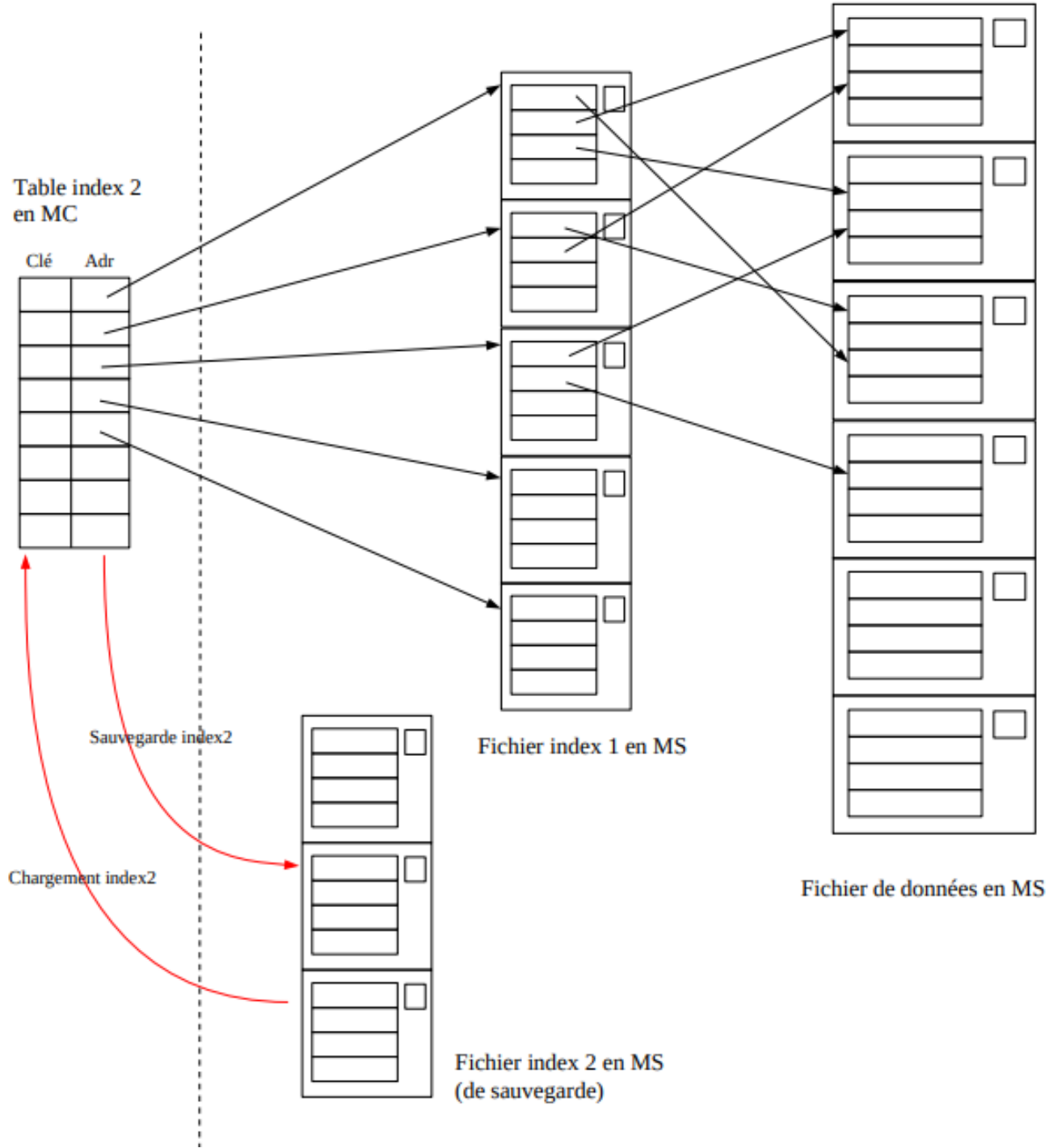
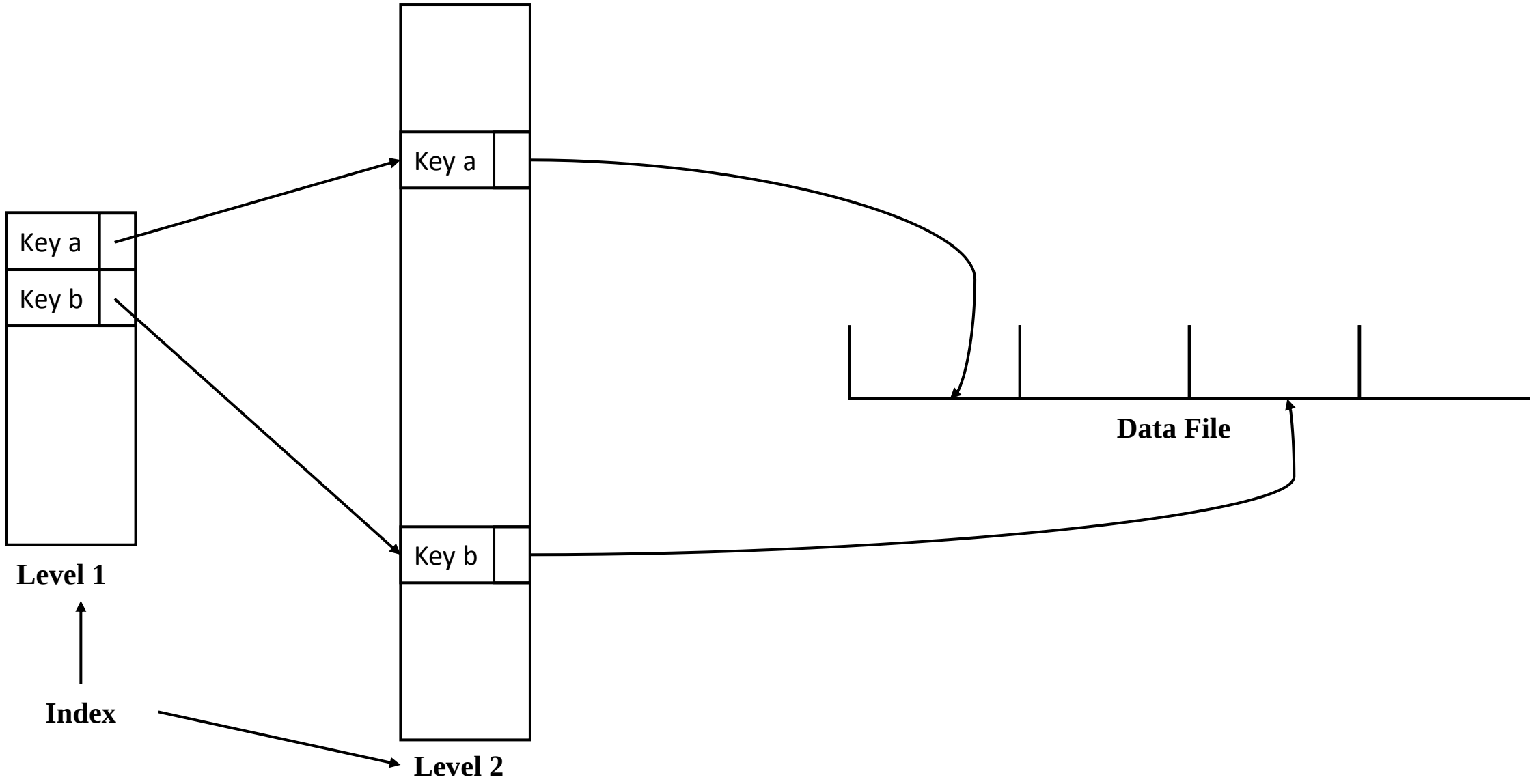


Table index 2
en MC
Clé
Adr
Fichier index 1 en MS
Fichier de données en MS
Sauvegarde index2
Chargement index2
Fichier index 2 en MS
(de sauvegarde)


Key a
Key b
Level 1
Index
Key a
Key b
Level 2
Data File

Search for a Record with a Given Key

- To search for a record with a given key, begin by searching for the key in the level 1 index. An interval is then selected. The search continues in the level 2 index only within this interval. If the key is found, proceed in the same way as in the case with a single index. It is clear that the two binary searches on the two small vectors (level 1 index and a portion of the level 2 index) are much faster than a single binary search on a single large vector (level 2 index).

FILE ORGANIZATION

Hashing

Outlines

- Introduction
- Hashing Principle
- Terminology
- Hash Functions
- Collision Resolution Methods
- Conclusion

Introduction

- **Issue:** Suppose we want to organize data (records) arriving in any order into an array.
- **Possible Solutions:** Generally, there are two ways to organize them:

1. If the array is unordered:
   - **Sequential search → O(n) → Slow**
   - **Insertion is done at the end of the array → O(1) → Fast**

Introduction

2. If the array is ordered:

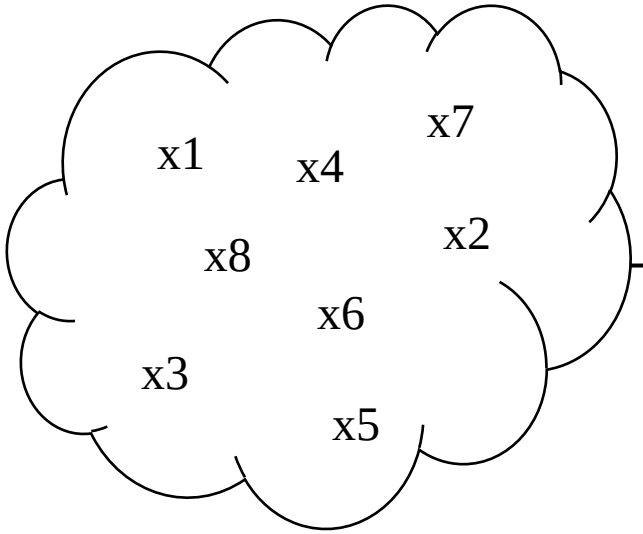
- **Binary search for retrieval → O(log2(n)) → Fast**
- **Insertion causes shifting of elements in the array → O(n) → Slow**

Introduction

- A third possibility for organizing data in an array is to:
  - Place the data "x" at a location "y" calculated by a function "h" such that y = h(x).
  - This is referred to as a hash table or **Hashing**.
  - In this type of organization, whether inserting or searching for data, the operations can be performed quickly (O(c)).

Principle

Data to store



The function "h" should return values between 0 and N-1

Organization through address calculation

h(x)

Hash Table "TH"

0	
1	x2
2	x4
3	x8
4	
5	x6
6	
…	x5
…	x1
…	
…	x3
N-1	x7

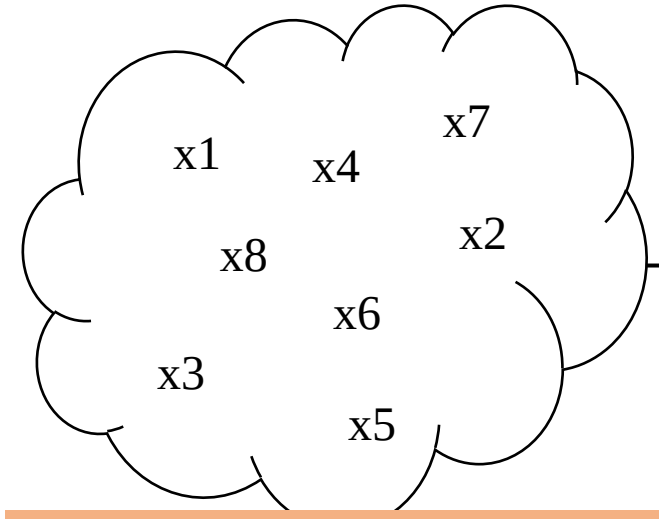
Address calculation allows storing data (x) in a table (HT) of size N, using a function (h)

Terminology

- The function h is called the hash function.
- The primary address (h(x)) of data x is the result returned by the hash function h.
- Synonyms are data that share the same primary address, i.e., x1 and x2 are synonyms if h(x1) = h(x2). It is also said that x1 and x2 collide.
- Overflow occurs when there is data not in its primary address. It is also said to be stored in a secondary address.
- The secondary address is determined by a given method, known as a collision resolution method.

Principle

Data to store



Organization through address calculation

h(x)

Hash Table "TH"

0	
1	x2
2	x4
3	x8
4	
5	x6
6	
…	x5
…	x1
…	
…	x3
N-1	x7

To use a hashing technique, the following must be defined:

1. A hash function "h."

2. A collision resolution method.

HASH FUNCTIONS

- The goal is to find a function h such that $0 \leq h(x) < N$, minimizing the number of collisions.
  - Ideally, having a bijective hash function, meaning a function that assigns a **new** location in the table for each data to be inserted.
  - The worst-case scenario is when all data is hashed to the same address.
  - An acceptable solution is one where some data shares the same address (h is surjective).

HASH FUNCTIONS

- There are several hash functions, the most commonly used ones being:

    1. The division function.
    2. The middle square function.
    3. The radix transformation function.

HASH FUNCTIONS

1. The division function: **h(x) = x MOD N**

   - It returns the remainder of the division by N, where N is the size of the table.
   - It is an easy and fast function to compute, but its quality depends on the value of N.
   - It is demonstrated that:
     - Choosing N as a power of 2 is generally not a good idea.
     - Choosing N as a prime number is usually a good choice.

HASH FUNCTIONS

1. The division function: **h(x) = x MOD N**
   - **Example:** Calculate the hash function for the following data:

X	N = 10	N = 11
5	**5**	**5**
55	**5 → Collision**	**0**
23	**3**	**1**
453	**3 → Collision**	**2**

No collisions in the case where N = 11 because N is a prime number.

HASH FUNCTIONS

2. The middle square function
   - Square the data x (x^2) and take the middle digits.
   - This method gives good results if the squared number does not have zeros.
   - **Example:** Calculate the hash function for the following data:

X	x^2	N = 10	N = 100
500	250000	0	0
12	144	4	14 or 44
453	205209	5 or 2	52

HASH FUNCTIONS

- The radix transformation function: **$h(x) = (x)_b$ MOD N**
  - Convert the data "x" into a base "b" numeral system and take the remainder of the division by "N."
  - **Example:** Calculate the hash function for the following data when b = 11 and N = 10 or 100:

X_{10}	X_{11}	N = 10	N = 100
12	11	1	11
453	382	2	82

HASH FUNCTIONS

- In conclusion, there is no universal hash function.
- However, a good function should be:
  - Fast to compute
  - Uniformly distribute elements
- It depends on:
  - The machine
  - The elements
- But no function can completely avoid collisions, and they will need to be handled.

HASH FUNCTIONS

- **Exercise 1:** Let E = {a, b, c, d, e, f, g, h} be a set of records. We want to insert these records into a hash table of 10 slots based on their key:

Record	**a**	**b**	**c**	**d**	**e**	**f**	**g**	**h**
Key	5	51	23	453	500	12	38	42

1. Calculate the primary address of each record in cases where the hash function is:
   a) Division, i.e., h(x) = x MOD N
   b) Middle square, i.e., square the data x (x^2) and take the middle digit
   c) Radix transformation, i.e., $(x)_b$ MOD N, where b = 11

HASH FUNCTIONS

- **Solution:** Let E = {a, b, c, d, e, f, g, h} be a set of records. We want to insert these records into a hash table of 10 slots based on their key:

1. Calculate the primary address of each record.

Record	**a**	**b**	**c**	**d**	**e**	**f**	**g**	**h**
Key or x	5	51	23	453	500	12	38	42
X MOD N	5	1	3	3	0	2	8	2
Middle Square	2	6	2	5	0	4	4	7
	5	0		2				6
x_{11} MOD N	5	7	1	2	5	1	5	2

COLLISION RESOLUTION METHODS

- During the insertion of x, if the primary address h(x) is already in use by another data, the collision resolution method helps find another (free) location for x.

COLLISION RESOLUTION METHODS

- To resolve collisions, two strategies are available:

a. Direct methods or addressing by computation:

1. Linear probing
2. Double hashing

b. Indirect methods or chaining:

3. Separate chaining
4. Internal chaining

COLLISION RESOLUTION METHODS

1. Linear Probing:

- If a collision occurs at the position h(x), we try the preceding positions: h(x)-1, h(x)-2, h(x)-3, ..., 0, N-1, N-2, ..., until finding an empty slot.
- Encountering an empty slot indicates that the data does not exist.
- A vacant slot in the hash table must be sacrificed to complete the testing sequence.

COLLISION RESOLUTION METHODS

1. **Linear Probing:** Exercise 1 (Question 2. a):

Record	a	b	c	d	e	f	g	j
h(x)	5	1	3	3	0	2	8	2

Index	**Empty**	**Record**
0	T	
1	T	
2	T	
3	T	
4	T	
5	T	
6	T	
7	T	
8	T	
9	T	

Initial state of the Hash Table

After the insertion of a, b, and c:

Index	**Empty**	**Record**
0	T	
1	F	b
2	T	
3	F	c
4	T	
5	F	a
6	T	
7	T	
8	T	
9	T	

COLLISION RESOLUTION METHODS

1. **Linear Probing:** Exercise 1 (Question 2. a):

Record	a	b	c	d	e	f	g	j
h(x)	5	1	3	3	0	2	8	2

Collision

Index	**Empty**	**Record**
0	T	
1	T	b
2	T	**d**
3	T	c
4	T	
5	T	a
6	T	
7	T	
8	T	
9	T	

Calculation of h(d) - 1 = 2 ⇒ empty slot

COLLISION RESOLUTION METHODS

1. **Linear Probing:** Exercise 1 (Question 2. a):

Record	a	b	c	d	e	f	g	j
h(x)	5	1	3	3	0	2	8	2

Collision

After the insertion of e

Index	**Empty**	**Record**
0	F	e
1	F	b
2	F	**d**
3	F	c
4	T	
5	F	a
6	T	
7	T	
8	T	
9	T	

Calculation of h(f) - 1 = 1 ⇒ occupied slot

Calculation of h(f) - 2 = 0 ⇒ occupied slot

Calculation of h(f) - 3 + 10 = 9 ⇒ empty slot

COLLISION RESOLUTION METHODS

1. **Linear Probing: Exercise 1** (Question 2. a):

Record	a	b	c	d	e	f	g	j
h(x)	5	1	3	3	0	2	8	2

Collision

After the insertion of f, g

Index	**Empty**	**Record**
0	F	e
1	F	b
2	F	**d**
3	F	c
4	T	
5	F	a
6	T	
7	T	
8	F	g
9	F	**f**

Calculation of h(f) - 1 = 1 ⇒ occupied slot

Calculation of h(f) - 2 = 0 ⇒ occupied slot

Calculation of h(f) - 3 + 10 = 9 ⇒ empty slot

Calculation of h(j) - 4 + 10 = 8 ⇒ occupied slot

Calculation of h(j) - 5 + 10 = 7 ⇒ empty slot

COLLISION RESOLUTION METHODS

1. **Linear Probing:** Exercise 1 (Question 2. a):

Record	a	b	c	d	e	f	g	j
h(x)	5	1	3	3	0	2	8	2

After the insertion of j

Index	**Empty**	**Record**
0	F	e
1	F	b
2	F	**d**
3	F	c
4	T	
5	F	a
6	T	
7	F	**j**
8	F	g
9	F	**f**

COLLISION RESOLUTION METHODS

1. Linear Probing:

Record	a	b	c	d	e	f	g	j
h(x)	5	1	3	3	0	2	8	2
Address	Primary	P	P	Secondary	P	S	P	S

- The search for "k" (such that h(k) = 2) ends with failure in the empty slot at index 6. The test sequence is: 2, 1, 0, 9, 8, 7.
- If we were to insert "k," the data would be assigned to slot 6 (if it's not the last empty slot).
- The table is considered full when the number of inserted elements equals N-1, requiring the sacrifice of an empty slot.

Index	**Empty**	**Record**
0	F	e
1	F	b
2	F	**d**
3	F	c
4	T	
5	F	a
6	T	
7	F	**j**
8	F	g
9	F	**f**

COLLISION RESOLUTION METHODS

1. Linear Probing:

- The search for a data point x proceeds as follows:

  a) Calculate the primary address of x (i.e., i = h(x)).

  b) If hash table slot "i" contains the data x, the search is complete.

  c) Otherwise, search for data x in the preceding slots: i-1, i-2, i-3, ..., 0, N-1, N-2, ..., until finding data x or an empty slot.

  d) If the search stops with an empty slot, it indicates that the data does not exist.

COLLISION RESOLUTION METHODS

1. Linear Probing:

- The insertion of data x unfolds as follows:

a) Calculate the primary address of x (i.e., i = h(x)).

b) If slot "i" in the hash table is empty, then insert x into this slot and mark it as non-empty.

c) Otherwise, traverse the preceding slots: i-1, i-2, i-3,..., 0, N-1, N-2, ..., until finding an empty slot (i.e., "j").

d) Insert x into slot "j" and mark it as non-empty.

COLLISION RESOLUTION METHODS

1. Linear Probing:

- The physical deletion of data x results in an empty slot.
- This new empty slot may make other data inaccessible.
- For example, if we delete b by emptying slot 1, we simultaneously lose data f (where h(f) = 2) because it is no longer accessible.
- Therefore, tests must be performed before emptying a slot.

Index	Empty	Record
0	F	e
1	F	b
2	F	**d**
3	F	c
4	T	
5	F	a
6	T	
7	F	**j**
8	F	g
9	F	**f**

COLLISION RESOLUTION METHODS

1. Linear Probing:

- The principle of deleting data x is as follows:
  a) Search for the address i of x.
  b) Traverse all preceding slots: j = i-1, i-2, i-3,..., 0, N-1, N-2, …, until finding an empty slot. For each slot j, check that its data remains accessible if slot i is emptied.
  c) If all slots j remain accessible when emptying i, then empty slot i and stop.
  d) Otherwise, move slot j to slot i and attempt to empty its original location by testing the slots above that have not yet been tested. This follows the same principle as applied to slot i.

COLLISION RESOLUTION METHODS

1. Linear Probing:

- For example, if we delete b by emptying slot 1,
- We will move the data f (h(f) = 2) to slot 1 and the data j (h(j) = 2) to slot 9.

Index	Empty	Record
0	F	e
1	F	b
2	F	**d**
3	F	c
4	T	
5	F	a
6	T	
7	F	**j**
8	F	g
9	F	**f**

Index	Empty	Record
0	F	e
1	F	f
2	F	d
3	F	c
4	T	
5	F	a
6	T	
7	F	
8	F	g
9	F	j

COLLISION RESOLUTION METHODS

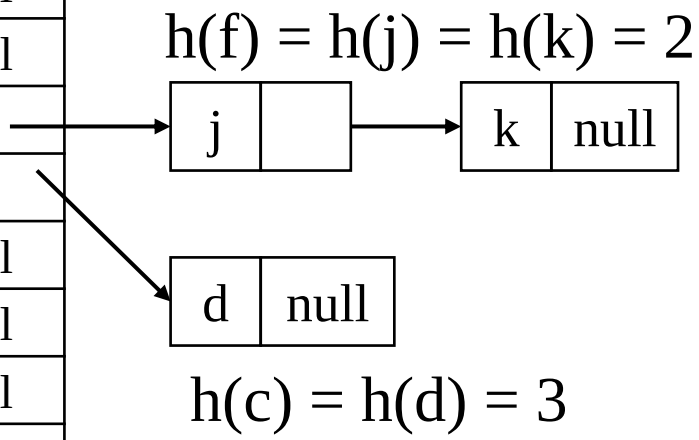
3. Separate chaining:

- Overflow data (in case of collisions) is stored in an area not "addressable" by the hash function, for example, outside the table in the form of a Linked List.

- The "Link" field connects the data in primary slots with their synonyms in overflow.

- The number of inserted data may exceed the size of the table (N).

Index	Empty	Record	Link
0	F	e	Null
1	F	b	Null
2	F	f	
3	F	c	
4	T		Null
5	F	a	Null
6	T		Null
7	F		Null
8	F	g	Null
9	F		Null



COLLISION RESOLUTION METHODS

3. Separate chaining:

- The search for data x proceeds as follows:

  a) Calculate the primary address of x (i.e., i = h(x)).

  b) If the "i" slot in the hash table contains the data x, then the search is complete.

  c) Otherwise, continue the sequential search in the "link" list.

COLLISION RESOLUTION METHODS

3. Separate chaining:

- The insertion of data x proceeds as follows:
    a. Calculate the primary address of x (i.e., i = h(x)).
    b. If the "i" slot is empty, then insert the data into this slot.
    c. Otherwise, insert (at the beginning) into the list associated with this slot.

COLLISION RESOLUTION METHODS

3. Separate chaining:

- The deletion of data x proceeds as follows:
    a. Search for data x in the hash table.
    b. If x is in its primary address (the data in slot h(x)), then
        i. If the "link" list is not empty, move the first element of the list into slot h(x) (overwriting x).
        ii. Otherwise, empty slot h(x).
    c. Otherwise (x is in overflow in a list), delete it from that list.

Conclusion

- File organization is a sophisticated discipline bridging logical data representation and physical storage mechanisms
- No universal solution exists - file management requires careful consideration of data volume, access patterns, and system constraints
- Future developments will focus on creating more intelligent and dynamic methods for managing increasingly complex data landscapes
- The ultimate goal: developing storage systems that are fast, reliable, and adaptable to evolving computational needs