Guided Exercises n°2 – Sequential file structures

Objectives: Writing algorithms and programming in C, using the abstract machine model discussed during course session, to perform operations on sequential file structures.

1. Course questions

- What are the advantages and disadvantages of TOF files?
- What is the loading factor of a file?
- How do you calculate the cost of an algorithm on a file structure?
- What are the advantages and disadvantages of overflow zones for a TOF file?
- What are the advantages and disadvantages of TOV{S/$\overline{S}$} files?
- How can records of varying sizes be separated when the length is not fixed?
- What metrics (or parameters) are generally used to evaluate the performance of a given file structure?

2. Basic access operations.

Consider the twelve cases in the diagram presented during course session, showing the different access methods. For each case, give the algorithms and develop the C programs for :

- Search for a record with a given key
- Insert a new record
- Delete a record with a given key
- Find all the records whose key belongs to a given interval

If the file is an ordered table, also develop the initial loading operation, which consists of filling the file at a rate of μ% per block with records read in ascending order according to their key.

3. Compacting a file.

Let F be a TOF file (seen as a table, ordered, and the records are of fixed size). The maximum capacity of a block is fixed at b records. We want to reduce the size of the file by recovering the spaces occupied by logically deleted records and any gaps that may exist in each block. After this operation, the file will no longer contain any logically deleted records, and all the blocks except the last one (perhaps) will be 100% full (the file will remain ordered).

The characteristics of the file are :

- The number of used blocks (nblk)
- The number of inserted records (nbIns)
- The number of logically deleted records (nbDel)

a) Give an algorithm that performs this operation in a single pass on the same file (without creating a new file or new blocks) and uses no more than two buffers in the main memory.

b) Give the cost of this operation in the worst case and on average.

4. Large arrays in secondary memory.

In the following, integer arrays are represented by sequential files (T$\bar{o}$F) because they cannot be fully loaded into main memory.

The elements of an array of size *n* are stored sequentially in the file. The capacity of the blocks is *b* integers. Therefore, all blocks are completely filled except for the last one.

a) What are the characteristics of such a file and provide its declaration.

b) Provide an efficient algorithm to compute the average value of the elements in the array.

c) Provide an efficient algorithm to perform the multiplication of two arrays as follows:

*(a1, a2, ... an) X (b1, b2, ... bn) = (a1*b1, a2*b2, ... an*bn)*

5. <u>Large matrices in secondary memory</u>.

In the following, integer matrices (two-dimensional arrays) are represented by sequential files (TōF) because they cannot be fully loaded into main memory. The elements of a matrix with *n* rows and *m* columns are stored sequentially, row by row in the file (i.e., the elements of row *i* are stored before those of row *i+1*, and within the same row, the element of column *j* precedes that of column *j+1*). The block capacity is *b* integers. Therefore, all blocks are completely filled except for the last one.

a) What are the characteristics of such a file and provide its declaration.

b) Provide an efficient algorithm to find the address (block number and record number) of the element m_{ij} (located at row *i* and column *j*) in a matrix *M*.

c) Provide an efficient algorithm to transform a matrix *M1(n, m)* stored row by row into a new matrix *M2(n, m)* stored column by column (i.e., the elements of column *j* are stored before those of column *j+1*, and within the same column, the element of row *i* precedes that of row *i+1*).

6. Let F be a TōF file whose only characteristic is the number of the last block and Buf the only buffer available in main memory for manipulating this file.

– Give the declaration of this file structure (using the model seen in course session) and the algorithm for physically deleting the record located at position *j* of block number *i* by replacing it with the last record in the file: 'Delete(F, i, j)'

– What is the cost of this deletion operation in the best-case and worst-case scenarios?

7. Let F be a TōF file of integers consisting of N blocks, and VP be a given integer value (referred to as the 'pivot'). We aim to reorganize the contents of F in place (i.e., without adding new blocks) so that all values ≤ VP are at the beginning of the file and all values > VP are at the end of the file. Thus, the goal is to partition the values in F into the left and right parts of the file according to the pivot VP.

Example:

The file before reorganization:

block0	block1	block2	block3	block4	block5
[23 37 19 38 10]	[49 9 9 20 39]	[38 21 2 23 28]	[11 49 48 36 33]	[13 41 20 31 9]	[6]

The file after reorganization with a pivot = 15 :

block0	block1	block2	block3	block4	block5
[10 6 9 13 11]	[9 9 2 20 39]	[49 38 21 23 28]	[23 49 48 36 33]	[38 19 41 20 31]	[37]

|---------- Left part ------||---------- Right part ----------|
(≤ 15) (> 15)

In the result file, the order of the values in each of the two parts (left and right) is not important.

a) Give the declaration of the file and the two associated buffers.

b) Give an algorithm for performing such a reorganization operation using only 2 buffers in main memory and with an input/output cost not exceeding N reads and N writes.

8. Provide the binary search algorithm for a TOV$\overline{S}$ file where the only characteristic is the number of the last block. The structure of the blocks is as follows:

Type Tblock = struct

tab : array[b] of characters // containing the records

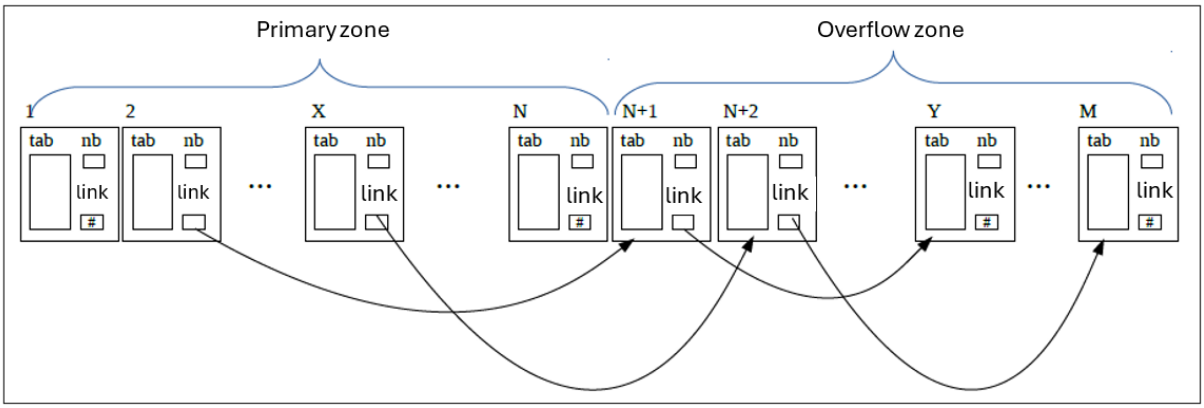
free_pos : integer // indice of the last free position in tab

key1, key2 : array[10] of characters // the smallest and biggest keys in the tab

End_struct

We assume that each record is stored as a string starting with its size (in 3 characters), followed by a logical deletion character, then its fixed-size key (in 10 characters). The remaining characters represent the other fields of the record.

9. We define an ordered file structure with overflow area management. The records are of fixed size and have a key of a totally ordered type.



The file, called F, is composed of two parts:

- An ordered primary zone consisting of N contiguous blocks. This zone is managed as a specific TOF structure. Its size (N) can only change during reorganization operations.
- An overflow zone, partially ordered, starting from block N+1 and extending as the number of overflow records increases. This zone is managed as a set of LOF structures. For each block X that overflows in the primary zone, a list in the overflow zone is associated, with the head link being the block X. This list of blocks contains the records that could not be inserted into block X because they have become full.

The blocks can contain a maximum of b records.

During the initial loading, the primary zone is filled with a predetermined fill rate, using records given in ascending order. The overflow zone will initially be empty. The last record in the primary zone represents a fictitious data item with an infinite key value (the highest value of the key type).

Locating a record with key c begins with a binary search in the primary zone and continues, if necessary, in the overflow zone, sequentially on one of the lists in this zone.

The insertion of a record into block i (in the primary zone) at position j naturally causes intra-block shifts. However, to avoid inter-block shifts, if the primary block (block i) is already full, one of the records from block i is moved and inserted into the overflow zone in the list pointed to by the link field of block i (if it already exists). If the list does not exist, it will be created, and the link field will be updated accordingly.

To perform this record relocation during an insertion into a full block, intra-block shifts are carried out only between positions j and b-1. This is followed by the expulsion of the second-to-last record in the block (position b-1) and its insertion into the overflow zone. This ensures that the last record of each primary block is never moved to the overflow zone. This approach guarantees that binary searches can be performed on the primary zone to quickly locate a record with a given key.

The insertion of a record into the overflow zone follows the general pattern of insertions in LOF-type structures. Once the block in the list has been located (sequentially), the record is inserted. If the block is already full, it splits by allocating a new block at the end of the overflow zone (M+1), and the records from the original block that caused the split are evenly distributed (half and half) between the two blocks. The chaining is, of course, updated to maintain the list structure.

Deletions are performed logically.

The characteristics of the file F are as follows:

- N: the number of the last block in the primary zone.
- M: the number of the last block in the overflow zone.
- nbI: the number of inserted records.

a) Give an efficient algorithm '*Locate(c, i, j)*' to locate a record with a given key c in the primary zone of this file. The algorithm should return (in i) the block number in the primary zone and the position (in j) where the record should be located (only in the primary zone). Specify the complexity of your algorithm.

b) Give an efficient algorithm '*List(a, b)*' to perform an interval query. The algorithm should display all records whose keys belongs to the interval [a, b]. What is the impact of the size of the overflow lists on the performance of this operation?

c) Give the '*Reorganize(rate, newName)*' algorithm for reorganizing the file. This involves constructing a new file named '*newName*' using the logically non-deleted records from the old file. The '*rate*' parameter specifies the desired initial load for the new file.

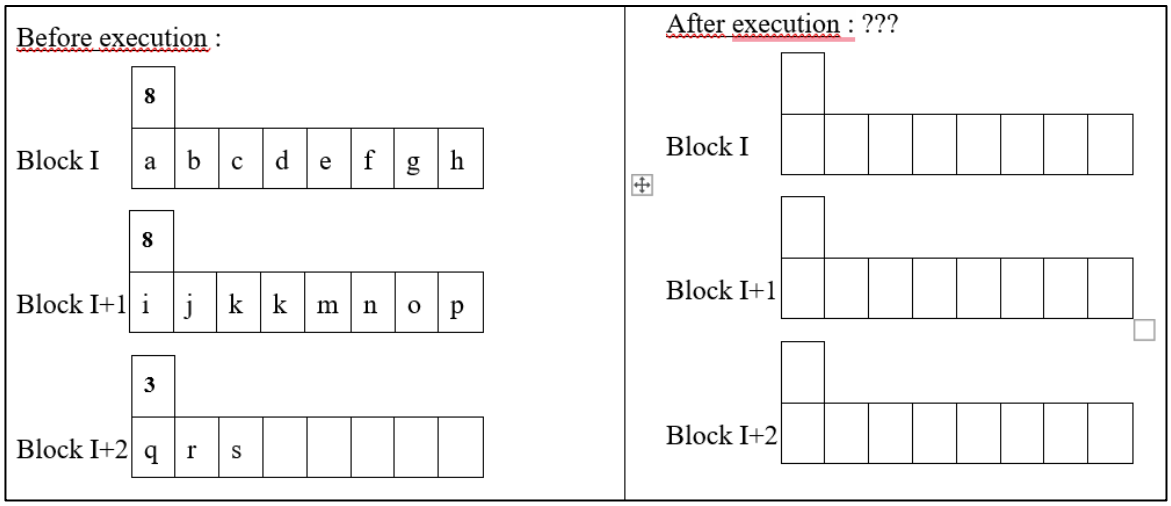
10. (a) What does the algorithm below do, and what is the structure of the file used?
Important Note: Blocks i and i+1 are full, while block i+2 is not full.

```
// A file is defined as follows:
F : FILE of TBlock BUFFER Buf, Buf2 HEADER (nblk:int) ;
// L'entête(F,1) désigne le nombre de blocs dans le fichier
Type TBlock = Struct
   nb : int ;              // Number of items in the block
   Tab : array[1..b] of characters ; // Array containing a maximum of b items of character strings
End_struct
…
```

```
// In Buf we have already read the block i+2
// Quotient (a,b): integer ; This function calculates the quotient of a over b and returns an integer.
// Rest(a,b) : integer ; This function calculates the rest of a over b and returns an integer.
Q := QUOTIENT(2*B+Buf.nb , 3);
R := REST(2*B+Buf.nb , 3);
// Updating the block I+2
// Block I+2 will contain Q+R items
// Shift in block I+ 2
j := 0;
    For k := Buf.nb   downto   1
        Buf.Tab[Q+R-j] := Buf.Tab[k];
        j ++;
    End_For
// Take ((B-Q)*2) items from block I+1
ReadBlock (F , Buf2 , I+1);
j := 1 ;
    For   k :=(2*Q - B + 1)   To   B
        Buf.Tab[j] := Buf2.Tab[k] ;
        j ++ ;
    End_For
Buf.nb := Q+R;
WriteBlock (F , Buf , I+2);
ReadBlock (F , Buf , I);

// Updating block I+1
// Shifting in the block I+1
j := 0 ;
    For   k := 2*Q – B   downto   1
        Buf2.Tab[Q-j] := Buf2.Tab[k];
        j ++ ;
    End_For
// Take the (B-Q) items from Block I
j := 1 ;
    For   k := Q+1 To   B
        Buf2.Tab[j] := Buf.Tab[k];
        j ++ ;
    End_For
Buf2.nb := Q;
WriteBlock (F , Buf2 , I+1);
// Updating the bloc I
Buf.nb := Q;
WriteBlock (F , Buf , I);
…
```

(b) Consider the following example representing three consecutive blocks I, I+1, and I+2 of a given file. By executing the precedent algorithm, represent the contents of each block I, I+1, and I+2 where B=8, with blocks I and I+1 being full, and block I+2 not being full.



11. We want to efficiently implement a FIFO (First-In, First-Out) queue model in secondary storage using files structured as lists of blocks. The elements of the queue are records of a given type *T_rec* with a fixed format. We have two buffers available in main memory.

- Give an efficient solution for implementing this model that allows the reuse of freed blocks.

  Reminder: The FIFO queue model consists of the following operations:

  { '*CreateQueue(F)*': Initializes F ; '*IsQueueEmpty(F)*': Tests if F is empty ; '*Enqueue(F, e)*': Adds element e to the end of the queue ; '*Dequeue(F, e)*': Removes the element at the front of the queue and stores/retrieves it in e }.

12. We aim to efficiently implement a FIFO queue model in secondary memory using files structured as contiguous blocks managed in a circular manner. The elements of the queue are records of a given type 'T_rec' with a fixed format. We have two buffers available in main memory.

- Give an efficient solution for implementing this model. The FIFO queue model to be implemented includes the following operations:
  - CreateQueue(F, N): Initializes F with a file of N blocks.
  - NbElement(F): Returns the number of elements present in the queue.
  - EnqueueGroup(F, n, T): Adds the n elements from array T to the end of the queue.
  - DequeueGroup(F, n, T): Removes and retrieves the first n elements from the front of the queue into array T.

13. Let F be a TōVC file (a file viewed as an unordered table, with variable-length records and inter-block overlap). Within a block, the records are prefixed by a logical deletion indicator (one character) and their length (five characters).

We aim to compact file F by physically removing the logically deleted records.

Give an algorithm to perform this operation while minimizing the number of physical reads, given that we have two buffers (buf1 and buf2) in the main memory.

Considerations:

- The type of a block is: Array[b] of characters.

- A record can straddle several blocks.
- The header block contains the following characteristics:
  - The number of the last block.
  - The first free position in the last block.
  - The total number of inserted records.
  - The total number of logically deleted records.

14. Fragmentation

Let F be a file viewed as an unordered table, with a fixed format; C1 and C2 are two given keys. We want to fragment F into three files (F1, F2, and F3) as follows:

- F1 should contain all records from F where the key is < C1.
- F2 should contain all records from F where the key is ≥ C1 and < C2.
- F3 should contain all records from F where the key is ≥ C2.

Give a module that performs this fragmentation using four buffers in main memory (buf, buf1, buf2, and buf3). The operation must be conducted in a single pass (one traversal of F).