Part II.
An almost perfect CDataframe

n this second part, the idea is to improve the previous CDataframe by acting on two aspects:

- Improving the structure of the column and the CDataframe to enable it to be used more widely.
- Adding advanced features

5. New structures

In order to allow the CDataframe to store data of different types, it is necessary to modify the column structure. It will always be a set of columns. However, we want the data in the same column to be of the same type, but the data in two different columns to be of two different types.

To achieve this, we need to use **generic types**.

5.1. The column

From a "structure" point of view, a column will essentially contain a title, an array of data with its two physical and logical sizes (as we cannot have its size beforehand).

The data stored in a column can be of any type. It is therefore necessary to create an additional attribute which will indicate the type of data it contains for each column created. In this project, we are going to restrict the number of types to the following set:

- Natural integers : $[0, 2^{32} - 1]$
- Relative integers: $[-2^{31}, 2^{31} - 1]$.
- Single-precision floats: encoded on 32 bits
- Double precision floating point : encoded on 64 bits
- Character : encoded on 8 bits
- Strings : array of characters
- Any type : Structured type

Then, to define a column type, we'll create an enumeration of these types as follows:

```
enum enum_type
{
    NULLVAL = 1 , UINT, INT,  CHAR, FLOAT, DOUBLE, STRING, STRUCTURE
};
typedef enum enum_type ENUM_TYPE;
```

Finally, to simplify the implementation of certain advanced functions (for the students who will be implementing the advanced functions in this second part), we have provided an attribute **"index"** which is defined as an array of integers, and which we will set to $NULL$ in this part.

All these attributes mean that the new column structure will be represented as shown in Figure 5.

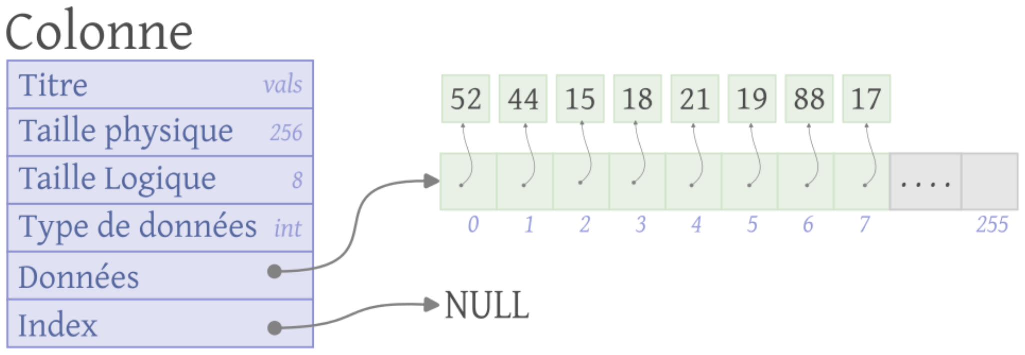

Figure 5: Structure of a column that can store any type of data.

We can see from this diagram that the data array contains pointers to the stored data (depending on the type of column). This is done deliberately so that the lack of information can be represented by the value $NULL$. In fact, this could have been represented by the value 0. However, 0 can only represent integer values. What's more, the 0 value could represent significant information that we want to store.

What type of data array will it be?

To enable our data array to adapt to the type of data that will be stored in the column, we need to opt for a solution that will provide flexibility. One of the tools offered by the C language to achieve this is the **union**.

A union is conceptually identical to a structure in that they both allow the user to combine different types under a single name. They are, however, different in the way memory is allocated to their members. On this link, you'll find all the information you need to understand how to use a **union**.

For our data array, we're going to define the following new **union** type:

```c
union column_type{
    unsigned int            uint_value;
    signed   int            int_value;
    char                    char_value;
    float                   float_value;
    double                  double_value;
    char*                   string_value;
    void*                   struct_value;
};
typedef union column_type COL_TYPE ;
```

But what does (void*) mean in this union?

`(void *)` designates a generic pointer. It is also a tool offered by the C language to implement a generic type. A variable of type `(void *)` means that it represents a pointer to any variable of any type. Note that it cannot contain values, only addresses (because addresses are all the same size). Then, to obtain a pointer to the desired type, you must first *cast* (force its type) the pointer `(void *)`.

Thus, the COLUMN structure and type will be defined in the C language as follows:

```c
struct column {
  char *title;
  unsigned int size; //logical size
  unsigned int max_size; //physical size
  ENUM_TYPE column_type;
  COL_TYPE **data; // array of pointers to stored data
  unsigned long long int *index; // array of integers
};
typedef struct column COLUMN;
```

5.1.1. Create a column

This function is used to create a column with a type and title given in the parameters. This means allocating the memory required to store only an empty column (with no data), assigning it a title and also initialising all the other attributes. This function will simply assign the value $NULL$ to the pointer *data*, it will not allocate memory space to store the data. It must also return a pointer to the newly created column, or a `NULL` pointer if the creation allocation failed.

Function prototype :

```c
/**
 * Create a new column
 * @param1 : Column type
 * @param2 : Column title
 * @return : Pointer to the created column
 */
COLUMN *create_column(ENUM_TYPE type, char *title);
```

Example of use :

```c
 COLUMN *mycol = create_column(CHAR, "My Column");
```

5.1.2. Inserting a value in a column

The following function is used to insert a value into a column. This function is generic, i.e. it inserts a value of any type (whose address is given as a parameter) into the data array `data`. The function returns 1 if the insertion was successful, 0 otherwise.

The pointer `(void *)` points to any value, which means that the function must first convert this value into the same type as that of the column into which it is to be inserted (mycol). If the pointer `value` is $NULL$ then the value is equal to `NULL` but it must still be inserted to indicate an absence of value.

The function `insert_value(COLUMN *col, void *value)`, must check if the logical size of the data array has not reached its maximum size (same value as the physical size). If not, the function must increase the physical size by reallocating an additional $256 bytes and update the value of the physical size.$

Reminder : The `realloc` function can sometimes change the location of the data array in the event that the system does not find a contiguous space large enough to extend the original array. Also, during the first allocation, i.e. when the physical size is equal to zero, it is the `malloc` function that should be used, as the `realloc` function does not allow space to be reserved but only extends an already existing space.

Function prototype :

```c
/**
  * @brief: Insert a new value into a column
  * @param1: Pointer to the column
  * @param2: Pointer to the value to insert
  * @return: 1 if the value is correctly inserted 0 otherwise
 */
int  insert_value(COLUMN *col, void *value);
```

Example of use :

```c
COLUMN *mycol = create_column(CHAR, "My column");
char a = 'A', c = 'C';
insert_value(mycol, &a);
insert_value(mycol, NULL);
insert_value(mycol, &c);
```

Note that the function takes a pointer to the value to be inserted as a parameter. It must not be inserted directly into the data array, as shown in the following code:

```c
COLUMN *mycol = create_column(INT, "New column");
for(int i = 0 ; i < 100 ; i++){
    insert_value(mycol, &i);
}
```

The correct method is to first allocate the space that will contain it, and then assign it the value to be stored. The following example will help you do this:

```
int insert_value(COLUMN *col, void *value){
    ...
    // check memory space
    ...
    switch(col->column_type){
        ...
        case INT:
            col->data[col->size] = (int*) malloc (sizeof(int));
            *((int*)col->data[col->size])= *((int*)value);
            break;
        ...
    }
    ...
    col->size++;
    ...
}
```

5.1.3. Free memory allocated by a column

A function which frees the memory allocated by a column. Note that this function must free all the memory allocated by the column's attributes before freeing the column's space.

Function prototype :

```
/**
 * @brief: Free the space allocated by a column
 * @param1: Pointer to the column
 */
 void delete_column(COLUMN **col);
```

5.1.4. Display a value

Displaying a value in C requires the use of the **"printf"** function, which needs to know the type of data in order to associate it with the correct format (

However, in our case, a column can contain any type. To display its values, you need to retrieve the type of data stored in the column and adapt the C code (using the *switch...case* instruction) in order to display them correctly.

To avoid this time-consuming implementation, we propose in this project to include a function that converts any data in the column and of any type into a string. In this way, when it is displayed, only the "%s" format will be sufficient.

The prototype of this function is as follows:

```c
/**
 * @brief: Convert a value into a string
 * @param1: Pointer to the column
 * @param2: Position of the value in the data array
 * @param3: The string in which the value will be written
 * @param4: Maximum size of the string
 */
void convert_value(COLUMN *col, unsigned long long int i, char *str, int size);
```

Figure 6 shows an example of the effect of this function on a value stored in a column of *INT*:

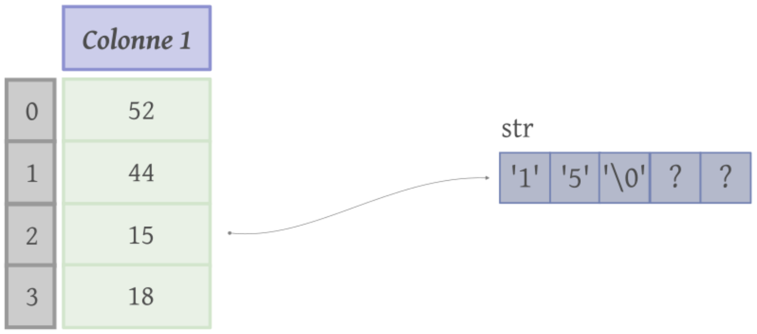

Figure 6: Convert the contents of line 3 (index 2) into a string.

Example of use :

```c
#define N 5
...
char str[5];
COLUMN *mycol = create_column(INT, "My column");
int a = 52, b = 44, c = 15, d = 18;
insert_value(mycol, &a);
insert_value(mycol, &b);
insert_value(mycol, &c);
insert_value(mycol, &d);
convert_value(mycol, 2, str, N);
printf("%s \n", str);
delete_column(&mycol);
...
```

Program output:

```
15
```

Tip :

To implement this function, you can use the function snprintf defined in `<stdio.h>` from the standard C.

Example

```
void convert_value(COLUMN* col, unsigned long long int i, char* str, int size){
    ...
    switch(col->column_type){
        ...
        case INT:
            snprintf(str, size, "%d", *((int*)col->data[i]));
            break;
        ...
    }
    ...
}
```

You are asked to think of a function that can convert structured data into a string.

5.1.5. Display the contents of a column

The following function displays the contents of a column. For each line it must also display the line number (index of the cell in which the value to be displayed is located) followed by the value contained in this cell. If the value is zero ($NULL$) then it must be able to display the character string $NULL$.

Function prototype :

```
/**
 * @brief: Display the content of a column
 * @param: Pointer to the column to display
 */
void print_col(COLUMN* col);
```

Example of use :

```
COLUMN *mycol = create_column(CHAR, "Column 1");
char a = 'A', c = 'C';
insert_value(mycol, &a);
insert_value(mycol, NULL);
insert_value(mycol, &c);
print_col(mycol);
```

Output :

```
[0] A
[1] NULL
[2] C
```

5.1.6. Other functions

In addition to the above functions, we need to implement all the functions that enable the following operations to be carried out:

- Return the number of occurrences of a value x (x given as a parameter).
- Return the value present at position x (x given as a parameter).
- Return the number of values that are greater than x (x given as a parameter).
- Return the number of values that are less than x (x given as a parameter).
- Return the number of values which are equal to x (x given as a parameter).

Note that other useful functions could potentially be added as the need arises in the rest of this project.

5.2. The CDataframe

You have two choices:

- Treat a CDataframe as an array of columns. To do this, follow the instructions described in section 4.2.
- Consider a CDataframe to be a double-chained list. To do this, follow the instructions in the 7 section.

From there, develop at least the basic functions listed in the project description.

Faire section 7 avant de passer au 6

6. Advanced functionalities

We can imagine a multitude of advanced functions to be added to the CDataframe, inspired by the functions offered in the "Dataframe" of "Pandas" in Python.

In what follows, we will detail the functionality that allows you to sort the columns of the CDataframe, but it is possible to plan to add one or more other functions such as concatenation or join.

6.1. Sort a column

Sorting a column allows you to display its values in a certain order (ascending or descending) and also to check the existence (search) of a value in a column very quickly using the dichotomous search technique.

Sorting a column involves changing the position of the elements in the column. If that column belongs to a *CDataframe*, then you also have to move values into other columns, which becomes inefficient and tedious. Even more annoyingly, if a new sort is performed on the same *CDataframe* according to a different column, the laborious sorting work done previously is lost.

To solve this problem efficiently, we're going to use the **"index"** attribute (array of integers) already provided for in the **COLUMN** structure. This is used to indicate the position of each element in the column. Thus, in a CDataframe, there will be an index associated with each column as illustrated in Figures 7 and 8.

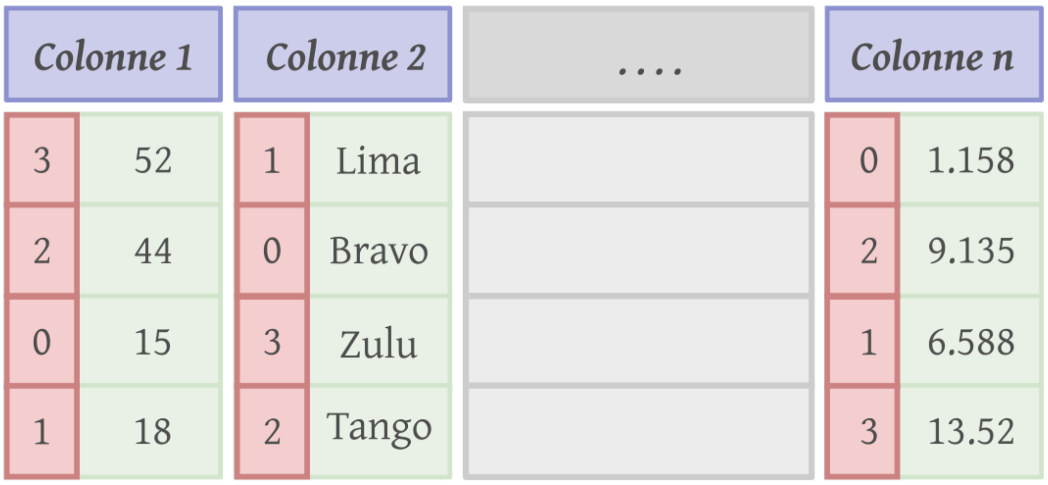

Figure 7: Status of the CDataframe with the indexes associated with the columns.

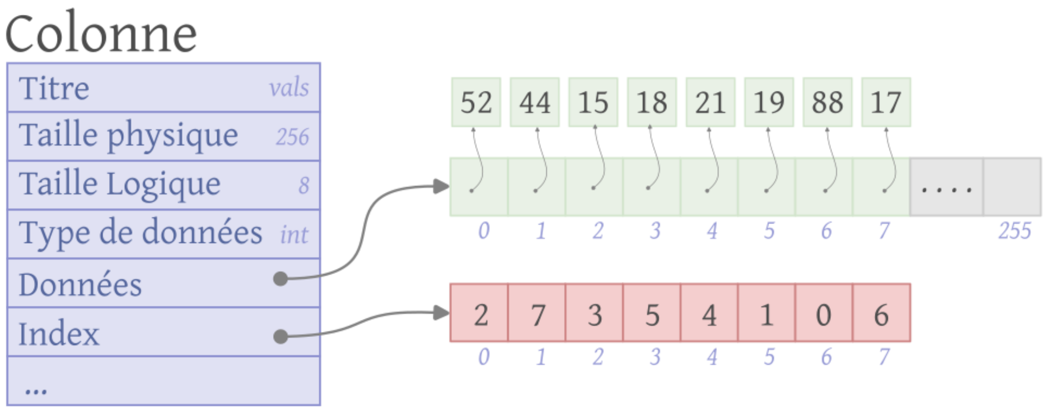

Figure 8: Structure of a column using an index.

How will it work?

Initially, the index value assigned to each item of data in the column corresponds to the line number (starting with the value 0).

The idea is to use this index to help sort the CDataframe efficiently. In fact, the trivial method you might naturally think of would be to apply a sorting algorithm to one of the CDataframe columns. However, a sorting algorithm imposes permutations of values on the same column, and for the CDataframe to be consistent, it would be necessary to apply the same permutations to the other columns of the CDataframe. If, in addition, a sort had to be performed on another column, the first sort would be cancelled.

Associating indexes with CDataframe columns has two advantages:

- To sort a column, it will no longer be necessary to swap the data. Simply swap the values in the index array;
- To sort several CDataframe columns, simply sort the indexes associated with each of them without having to move the data from their initial boxes.

What should I do if new data is inserted after a sort action?

When a column is sorted, the insertion of new data requires the assignment of a sequential number which does not necessarily correspond to the sort order already established. To avoid re-sorting the column, we propose to add other attributes to the **COLUMN** structure to determine the state of the index. The attributes to be added are as follows:

- **valid_index** : which is an attribute which can take 3 values. It is 0 if the column is not sorted at all, -1 if the column is partially sorted (this happens when a new value is inserted. In this case, the whole column is sorted except for the last value), 1 if the column is correctly sorted.
- **sort_dir**: This attribute indicates the sorting direction (ascending or descending). It takes 0 for ascending order, 1 otherwise.

The column structure will therefore be extended as follows:

```
struct column {
  ...
  unsigned long long  *index;
  // index valid
  //  0 : no index
  // -1 : invalid index
  //  1 : valid index
  int valid_index;
  // direction de tri Ascendant ou Déscendant
  // 0 : ASC
  // 1 : DESC
  int sort_dir;
};
```

Note that whatever the "COLUMN" structure used previously, the addition of these new attributes linked to the index will only have an impact on the column creation function where their initialization will have to be completed.

6.2. Sort a column

To sort a column, an efficient sort algorithm is required. Some of the most efficient sorting algorithms are :

- The quick sort (Quicksort) which works well if the array is not sorted;
- The insertion sort which works better than the *Quicksort* algorithm in the case where the array is almost sorted.

To improve the performance of our sorting, we want to take advantage of these two techniques. Thus, if the attribute **valid_index** is 0 (unsorted column), the *Quicksort* algorithm should be applied. If the **valid_index** attribute is -1 (column partially sorted), then the insertion sort will be applied.

Insertion sort

```
Algorithm 1: Insertion sort
    Data: 1D array $tab$ of size $N$
    Result: The array $tab$ sorted
1   for $i \leftarrow 2$ to $N$ do
2       $k \leftarrow tab[i]$;
3       $j \leftarrow i - 1$;
4       while $j > 0$ and $tab[j] > k$ do
5           $tab[j+1] \leftarrow tab[j]$;
6           $j \leftarrow j - 1$;
7       end
8       $tab[j+1] \leftarrow k$;
9   end
```

Quick sort

```
Algorithm 2: Quick sort
    Data: 1D array $tab[]$ of size $N$
    Result: The array $tab[]$ sorted
1   Function Quicksort($tab[]$, $left$, $right$):
2       if $left < right$ then
3           $pi \leftarrow$ PARTITION($tab[]$, $left$, $right$);
4           Quicksort($tab[]$, $left$, $pi - 1$);
5           Quicksort($tab[]$, $pi + 1$, $right$);
6       end
7   Function Partition($tab[]$, $left$, $right$):
8       $pivot \leftarrow arr[right]$;
9       $i \leftarrow left - 1$;
10      for $j \leftarrow left$ to $right - 1$ do
11          if $tab[j] \leq pivot$ then
12              $i \leftarrow i + 1$;
13              Swap $tab[i]$ and $tab[j]$;
14          end
15      end
16      Swap $tab[i+1]$ and $tab[right]$;
17      return $i + 1$;
```

This is the prototype sorting function:

```c
#define ASC  0
#define DESC 1
/**
 * @brief: Sort a column according to a given order
 * @param1: Pointer to the column to sort
 * @param2: Sort type (ASC or DESC)
 */
void sort(COLUMN* col, int sort_dir);
```

6.3. Display the contents of a sorted column

The function for displaying a column previously implemented allowed values to be displayed sequentially. With the introduction of the index, this function will not fulfil its role properly, particularly when the column is sorted.

We have therefore asked you to implement another display function that will display the values in the sequential order indicated in the index array.

Function prototype :

```
/**
 * @brief: Display the content of a sorted column
 * @param1: Pointer to a column
 */
void print_col_by_index(COLUMN *col);
```

Example of use :

```
COLUMN *mycol = create_column(INT, "sorted column");
int a = 52;
int b = 44;
int c = 15;
int d = 18;
insert_value(mycol, &a);
insert_value(mycol, &b);
insert_value(mycol, &c);
insert_value(mycol, &d);
printf("Column content before sorting : \n");
print_col(mycol);
sort(mycol,ASC);
printf("Column content after sorting : \n");
print_col_by_index(mycol);
```

Example of output :

```
Column content before sorting :
[0] 52
[1] 44
[2] 15
[3] 18

Column content after sorting :
[0] 15
[1] 18
[2] 44
[3] 52
```

Example of use with strings:

```
    COLUMN *mycol = create_column(STRING, "String column");
    insert_value(mycol, "Lima");
    insert_value(mycol, "Bravo");
    insert_value(mycol, "Zulu");
    insert_value(mycol, "Tango");
    printf("Column content before sorting : \n");
    print_col(mycol);
    sort(mycol,ASC);
    printf("Column content after sorting : \n");
    print_col_by_index(mycol);
```

Example of output :

```
    Column content before sorting :
    [0] Lima
    [1] Bravo
    [2] Zulu
    [3] Tango

    Column content after sorting :
    [0] Bravo
    [1] Lima
    [2] Tango
    [3] Zulu
```

6.4. Delete column index

It is true that the presence of indexes improves performance when sorting and searching for information in the CDataframe. However, they require additional storage space and management. So sometimes you want to lighten the load by deleting the index array associated with certain columns.

The following function deletes the association of an index with a given column. This involves freeing the memory allocated to the index array, assigning the value $NULL$ to the pointer *index* and updating the attribute *valid_ index* to the value 0.

Function prototype :

```
    /**
     * @brief: Remove the index of a column
     * @param1: Pointer to the column
     */
    void erase_index(COLUMN *col);
```

6.5. Check if a column has an index

This function is used to check the presence of an index by consulting the state of the "**valid_index**" attribute. It returns 0 if the index does not exist, -1 if the index is invalid, 1 otherwise.

Function prototype :

```c
/**
 * @brief: Check if an index is correct
 * @param1:  Pointer to the column
 * @return:  0: index not existing,
            -1: the index exists but invalid,
             1: the index is correct
 */
int check_index(COLUMN *col);
```

6.6. Update an index

This function is used to update an index. In fact, all you have to do is call the `sort` function. This function can be called when a new value is inserted after a sort operation has been applied.

Function prototype :

```c
/**
 * @brief: Update the index
 * @param1: Pointer to the column
 */
void update_index(COLUMN *col);
```

6.7. Dichotomous search

This function allows you to search for a value given as a parameter in a sorted column.

Function prototype :

```c
/**
 * @brief: Check if a value exists in a column
 * @param1: Pointer to the column
 * @param2: Pointer to the value to search for
 * @return: -1: column not sorted,
             0: value not found
             1: value found
 */
int search_value_in_column(COLUMN *col, void *val);
```

Part III.
A perfect CDataframe

This section looks at advanced features that can be applied to CDataframe.

- Add efficiency by implementing the CDataframe with a chained linear list instead of an array of columns.
- Add the ability to load data from a **.csv** file to avoid having to enter several lines of data.
- Enable the contents of a CDataframe to be stored in a **.csv** file.

7. Implementation of the CDataframe

To provide more flexibility for storing the CDataframe columns, we propose to implement it as a linked list.

The idea is that each column is stored in a link. To ensure better access, we think that a bi-directional list structure would be ideal (Figure 9) where the list is accessed from both the header and the tail, and from each link it is possible to access both the next link and the previous link.

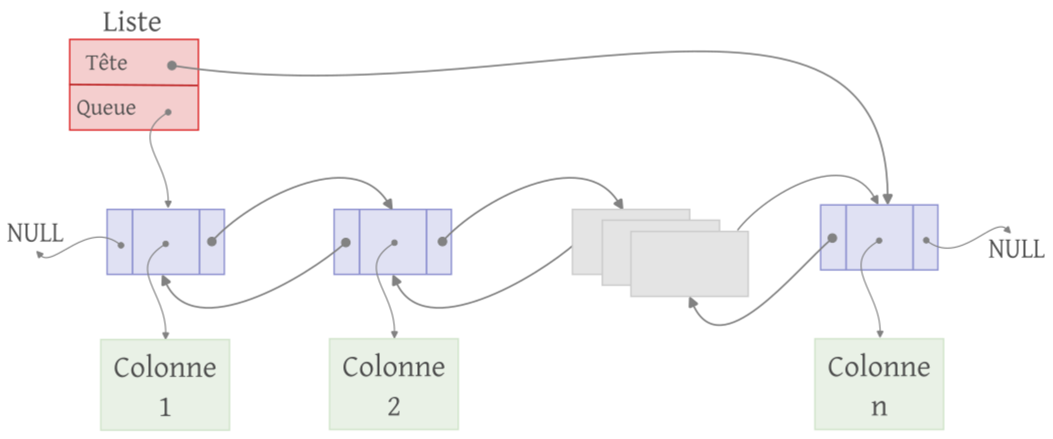

Figure 9: A CDataframe as a linked list.

The structures required to implement the list are as follows:

```
/**
 * Élément lnode
 */
typedef struct lnode_ {
    void *data;  // Pointer to a column
    struct lnode_ *prev;
    struct lnode_ *next;
} LNODE;

/**
```

```
 * Une liste
*/
typedef struct list_ {
    lnode *head;
    lnode *tail;
} LIST;
```

```
typedef LIST CDATAFRAME;
```

An implementation of a generic linked list is given in appendix A. You are free to reuse the given list or to create your own implementation.

Recommendation

We suggest that you separate your code into four files.

- **cdataframe (.h/.c) :** cdataframe manipulation functions. Only these functions can be used by the end user of the library. The corresponding header file must therefore be well written and well documented.
- **column (.h/.c):** column management functions
- **list (.h/.c):** linked list given in appendix
- **sort (.h/.c):** sorting algorithms.

7.1. Functions to be implemented

7.1.1. Creation of a CDataframe

Function prototype :

```
/**
 * Création d'un dataframe
*/
CDATAFRAME *create_cdataframe(ENUM_TYPE *cdftype, int size);
```

Example of use :

```
ENUM_TYPE cdftype [] = {INT,CHAR,INT};
CDATAFRAME *cdf = create_cdataframe(cdftype, 3);
```

7.1.2. Deleting a CDataframe

Propose a function to delete a CDataframe. This function must delete all the columns of the CDataframe and free all the memory previously allocated before deleting the CDataframe given as a parameter.

Function prototype :

```
/**
 * @brief: Column deletion
 * param1: Pointer to the CDataframe to delete
 */
 void delete_cdataframe(CDATAFRAME **cdf);
```

7.1.3. Delete a column

Propose a function which allows you to delete a column from a CDataframe by specifying the title of the column.

Function prototype :

```
/**
 * @breif: Delete column by name
 * @param1: Pointer to the CDataframe
 * @param2: Column name
 */
void delete_column(CDATAFRAME *cdf, char *col_name);
```

7.1.4. Number of columns

This function counts the number of columns in the CDataframe.

Function prototype :

```
/**
 * @brief: Number of columns
 * @param1: Point to the CDataframe
 * @return: Number of columns in the CDataframe
 */
int get_cdataframe_cols_size(CDATAFRAME *cdf);
```

7.2. Other functions

If you have chosen to use this CDataframe structure, you will need to allow it to offer at least the following additional functions:

1. Filling
    - Creation of an empty CDataframe
    - Filling in the CDataframe with user input
    - Hard filling of the CDataframe
2. Displaying
    - Display the entire CDataframe
    - Dispalay a part of the CDataframe rows according to a user-provided limit
    - Display a part of the columns of the CDataframe according to a limit supplied by the user
3. Usual operations
    - Add a row of values to the CDataframe
    - Delete a row of values from the CDataframe
    - Add a column to the CDataframe
    - Delete a column from the CDataframe
    - Rename the title of a column in the CDataframe
    - Check the existence of a value (search) in the CDataframe
    - Access/replace the value in a CDataframe cell using its row and column number
    - Display column names
4. Analysis and statistics
    - Display the number of rows
    - Display the number of columns
    - Display the number of cells equal to x (x given as parameter)
    - Display the number of cells containing a value greater than x (x given as a parameter)
    - Display the number of cells containing a value less than x(x given as parameter)

Faire la section 6 avant de faire la 8

8. Files

Faire la section 6 avant de faire la 8

When developing a data manipulation tool, it is important to allow it to access files (generally the format used is (**CSV**) in read and write mode in order to load/store a large amount of data.

A CSV (Comma-Separated Values) file is a file format commonly used to store tabular data, such as spreadsheet or database data. It is widely used because it is simple to understand and manipulate. In a CSV file, each line of the file represents a line of data, and the values in each line are separated by a delimiter, usually a comma (","), although other delimiters such as semicolons (";") or tabs ("\t") can also be used. Null values are noted by `"NULL"` or simply left empty.

Example content of a data.csv file:

```
52,Lima,1.158
44,Bravo,9.135
15,Zulu,6.588
18,Tango,13.52
22,Kilo,8.1400
75,Alpha,6.1400
13,Echo,7.18000
19,Romeo,3.4440
85,Mike,2.11
27,Hotel,5.951
36,Delta,1.22
12,Golf,0.64
```

In order to avoid manual data entry or hard initialisations of the CDataframe, in this project we want to be able to load data from a **.csv** file. Also, when data is entered by the user, we want to be able to store it in a **.csv** file.

Example of use :

Display from line 2 to line 9 of the previous file:

```
ENUM_TYPE cdftype [] = {INT,STRING,FLOAT};
CDATAFRAME *cdf = load_from_csv("data.csv", cdftype, 3);
// Example of a function that allows partial display of the CDataframe
print_dataframe_by_line(df,2,9);
```

Expected display:

```
18          Tango         13.520000
22          Kilo        8.1400001
75          Alpha         6.1400001
13          Echo        7.1800001
19          Romeo         3.4440001
85          Mike        2.1100001
27          Hotel         5.9510001
```

8.1. Load a CDataframe from a CSV file

This function is used to load data from a CSV file. It takes as parameters the name of the file and an array of types describing the types of each column in the CSV file.

Function prototype :

```
/**
 * @brief: Create a CDataframe from csvfile
 * @param1: CSV filename
 * @param2: Array of types
 * @param3: Size of array in param2
 */
CDATAFRAME* load_from_csv(char *file_name, ENUM_TYPE *dftype, int size);
```

You will find on the page the instructions needed to open and read a CSV file in C language.

8.2. Exporting a CDataframe to a CSV file

This function is used to export a CDataframe as a CSV file. The export format is the same as the import file, i.e. columns are separated by commas `','` and rows are separated by line breaks `'\n'`. The first line represents the column name. The path to the file is given as a parameter string.

```
/**
 * @brief: Export into a csvfile
 * @param1: Pointer to the CDataframe
 * @param2: csv filename where export file, if the file exists,
 *          it will be overwritten
 */
void save_into_csv(CDATAFRAME *cdf, char *file_name);
```

You will find on this page instructions for opening and writing to a CSV file in C.

A. Implemntation of a double-linked list

A.1. Header file (`list.h`)

```c
#ifndef _LIST_H_
#define _LIST_H_


/**
 * Élément lnode
 */
typedef struct lnode_ {
    void *data;
    struct lnode_ *prev;
    struct lnode_ *next;
} lnode;


/**
 * Une liste
 */
typedef struct list_ {
    lnode *head;
    lnode *tail;
} list;


/**
 * création d'un noeud
 */
lnode *lst_create_lnode(void *dat);

/**
 * crée la liste et retourne un pointeur sur cette dernière
 */
list *lst_create_list();

/**
 * supprimer la liste
 */
void lst_delete_list(list * lst);

/**
 * Insère pnew au début de la liste lst
 */
void lst_insert_head(list * lst, lnode * pnew);

/**
 * Insère pnew à la fin de la liste lst
 */
void lst_insert_tail(list * lst, lnode * pnew);

/**
 * Insère l'élément pnew juste après ptr dans la liste lst
 */
void lst_insert_after(list * lst, lnode * pnew, lnode * ptr);

/**
 * Supprime le premier élément de la liste
 */
void lst_delete_head(list * lst);


/**
 * Supprime le dernier élément de la liste
 */
void lst_delete_tail(list * lst);


/**
 * Supprime le lnode pointé par ptr
 */
void lst_delete_lnode(list * lst, lnode * ptr);


/**
 * Supprime tous les éléments de la liste lst
 */
void lst_erase(list * lst);


/**
 * retourne le premier node s'il existe sinon NULL
 */
lnode *get_first_node(list * lst);

/**
 * retourne le denier node s'il existe sinon NULL
 */
lnode *get_last_node(list * lst);

/**
 * retourne le node  suivant
 */
lnode *get_next_node(list * lst, lnode * lnode);
/**
 * retourne le node precedent
 */
void *get_previous_elem(list * lst, lnode * lnode);

#endif
```

A.2. Source file (list.c)

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "list.h"

lnode *lst_create_lnode(void *dat) {
    lnode *ptmp = (lnode *) malloc(sizeof(lnode));
    ptmp->data = dat;
    ptmp->next = NULL;
    ptmp->prev = NULL;
    return ptmp;
}



list *lst_create_list() {
    list *lst = (list *) malloc(sizeof(list));
    lst->head = NULL;
    lst->tail = NULL;
    return lst;
}


void lst_delete_list(list * lst) {
    lst_erase(lst);
    free(lst);
}


void lst_insert_head(list * lst, lnode * pnew) {
    if (lst->head == NULL) {
        lst->head = pnew;
        lst->tail = pnew;
        return;
    }
    pnew->next = lst->head;
    pnew->prev = NULL;
    lst->head = pnew;
    pnew->next->prev = pnew;
}

void lst_insert_tail(list * lst, lnode * pnew) {
    if (lst->head == NULL) {
        lst->head = pnew;
        lst->tail = pnew;
        return;
    }
    pnew->next = NULL;
    pnew->prev = lst->tail;
    lst->tail = pnew;
    pnew->prev->next = pnew;
}

void lst_insert_after(list * lst, lnode * pnew, lnode * ptr) {
    if (lst->head == NULL) {
        lst->head = pnew;
        lst->tail = pnew;
    } else if (ptr == NULL) {
        return;
    } else if (lst->tail == ptr) {
        lst_insert_tail(lst, pnew);
    } else {
        pnew->next = ptr->next;
        pnew->prev = ptr;
        pnew->next->prev = pnew;
        pnew->prev->next = pnew;
    }
}

void lst_delete_head(list * lst) {
    if (lst->head->next == NULL) {
        free(lst->head);
        lst->head = NULL;
        lst->tail = NULL;
        return;
    }
    lst->head = lst->head->next;
    free(lst->head->prev);
    lst->head->prev = NULL;
}

void lst_delete_tail(list * lst) {
    if (lst->tail->prev == NULL) {
        free(lst->tail);
        lst->head = NULL;
        lst->tail = NULL;
        return;
    }
    lst->tail = lst->tail->prev;
    free(lst->tail->next);
    lst->tail->next = NULL;
}

void lst_delete_lnode(list * lst, lnode * ptr) {
    if (ptr == NULL)
        return;
    if (ptr == lst->head) {
        lst_delete_head(lst);
        return;
    }
    if (ptr == lst->tail) {
        lst_delete_tail(lst);
        return;
    }
    ptr->next->prev = ptr->prev;
    ptr->prev->next = ptr->next;
    free(ptr);
}

void lst_erase(list * lst) {
    if (lst->head == NULL)
        return;
    while (lst->head != lst->tail) {
        lst->head = lst->head->next;
        free(lst->head->prev);
    }
    free(lst->head);
    lst->head = NULL;
    lst->tail = NULL;
}

lnode *get_first_node(list * lst) {
    if (lst->head == NULL)
        return NULL;
    return lst->head;
}

lnode *get_last_node(list * lst) {
    if (lst->tail == NULL)
        return NULL;
    return lst->tail;
}

lnode *get_next_node(list * lst, lnode * lnode) {
    if (lnode == NULL)
        return NULL;
    return lnode->next;
}

void *get_previous_elem(list * lst, lnode * lnode) {
    if (lnode == NULL)
        return NULL;
    return lnode->prev;
}
```