**Projet de programmation C**

# CDataframe

Idée : Halim Djerroud

Rédaction : Halim Djerroud, Asma Gabis

Mars 2024

**Consignes & Informations générales :**

➔ Ce projet est à réaliser exclusivement en langage C

➔ Pièce jointe au projet :

- Une implémentation d'une liste doublement chaînée
- Un schéma expliquant les différentes parties du projet et les différentes possibilités de sa réalisation.

➔ Organisation des équipes :

- Ce projet est à réaliser en binôme (un seul trinôme est autorisé si un nombre impair d'élèves)
- La liste des équipes est à remettre aux enseignants **au plus tard** à la fin de la première séance de suivi de projet

➔ Dépôt du projet : Deux dépôts sont exigés :

- Un dépôt intermédiaire contenant le squelette global du projet (principalement les fichiers .h)
- Un dépôt final de l'ensemble du projet avant la soutenance

➔ Dates clés :

- Date publication du projet : Semaine du **25/03/2024**
- Date de séance de suivie 1 : semaine du **25/03/2024**
- Date du dépôt intermédiaire : **21/04/2024 à 23h59**
- Date de séance de suivie 2 : semaine du **22/04/2024**
- Date de séance de suivie 3 : semaine du **13/05/2024**
- Date du dépôt final : **19/05/2024 à 23h59**
- Date de soutenance : Semaine du **20/05/2024**

➔ Rendu final : S'effectue sur Moodle sous forme d'une archive **.zip** contenant :

- Le code du projet avec tous les fichiers **.h** et **.c**
- Un fichier **README.txt** donnant la liste des programmes et comment les utiliser en pratique. Ce fichier doit contenir toutes les instructions nécessaires à l'exécution pour expliquer à l'utilisateur comment compiler et se servir de votre logiciel.

➔ Évaluation

- Barème détaillé : sera fourni plus tard
- Note finale du projet = Note code + Note soutenance (speeck + fonctionnalités réalisées)
- **Rappel :** note projet = 20% de la note du cours « Algorithmique et Structures de données 1 »
- Les membres d'une même équipe peuvent avoir des notes différentes en fonction des efforts fournis dans la réalisation de ce projet.

# Table des matières


1. Préambule 5

2. Introduction 5

3. Description du projet 5

I. Un CDataframe d'entiers 8

4. CDataframe 8
   4.1. Les colonnes . . . 8
      4.1.1. Créer une colonne . . . 9
      4.1.2. Insérer une valeur dans une colonne . . . 9
      4.1.3. Libérer l'espace allouée par une colonne . . . 10
      4.1.4. Afficher le contenu d'une colonne . . . 10
      4.1.5. Autres fonctions . . . 11
   4.2. Le CDataframe . . . 11
      4.2.1. Fonctionnalités . . . 12

II. Un CDataframe presque parfait 14

5. De nouvelles structures 14
   5.1. La colonne . . . 14
      5.1.1. Créer une colonne . . . 16
      5.1.2. Insérer une valeur dans une colonne . . . 17
      5.1.3. Libérer l'espace alloué par une colonne . . . 18
      5.1.4. Afficher une valeur . . . 18
      5.1.5. Afficher le contenu d'une colonne . . . 20
      5.1.6. Autres fonctions . . . 21
   5.2. Le CDataframe . . . 21

6. Fonctionnalités avancées 21
   6.1. Trier une colonne . . . 22
   6.2. Trier une colonne . . . 24
   6.3. Afficher le contenu d'une colonne triée . . . 25
   6.4. Effacer l'index d'une colonne . . . 27
   6.5. Vérifier si une colonne dispose d'un index . . . 27
   6.6. Mettre à jour un index . . . 28
   6.7. Recherche dichotomique . . . 28

III. Un CDataframe parfait 29

7. Implémentation du CDataframe 29
   7.1. Fonctions à implémenter . . . 30
      7.1.1. Création du CDataframe . . . 30
      7.1.2. Suppression d'un CDataframe . . . 31
      7.1.3. Supprimer une colonne . . . 31
      7.1.4. Compter le nombre de colonnes . . . 31

7.2. Autres fonctions . . . . . 32

**8. Fichiers** **32**
8.1. Charger un CDataframe depuis un fichier CSV . . . . . 33
8.2. Exporter un CDataframe dans un fichier CSV . . . . . 34

**A. Implémentation d'une liste chaînée** **35**
A.1. Fichier entête (`list.h`) . . . . . 35
A.2. Fichier source (`list.c`) . . . . . 36

# 1. Préambule

Les logiciels tableurs tels que "LibreOffice Calc" ou "MS Excel", sont des outils puissants pour manipuler des données, les trier, visualiser des graphiques, calculer des sommes et des moyennes, et bien plus encore. Cependant, lorsque vous recevez quotidiennement des milliers de données sous forme de fichiers structurés (CSV par exemple), et que vous devez effectuer tous ces traitements de façon répétitive, l'utilisation de logiciels tableurs devient fastidieuse. En effet, leur fonctionnement naturel impose l'ouverture de chaque fichier manuellement et la répétition des mêmes actions plusieurs fois, ce qui peut être chronophage et source d'erreurs.

Une des solutions existantes pour l'automtisation des tâches dans les tableurs est la programmation de Macros. Certains logiciels tableurs comme Calc permettent d'enregistrer ces macros pour automatiser des séquences d'actions effectuées sur des tâches simples et répétitives. Néanmoins, leur utilisation peut vite devenir complexe et difficile à gérer pour des traitements plus élaborés.

L'autre solution consiste à utiliser directement des scripts puissants et flexibles en passant par un langage de programmation tels que Python. En effet, grâce à sa librairie **Pandas**, il est possible de réaliser un large éventail de fonctions pour importer, nettoyer, analyser et visualiser des données. Cette souplesse passe par l'utilisation d'une structure de données, propre à Pandas, appelée **"DateFrame"** qui se gère de façon similaire à une feuille de calcul se trouvant dans un tableur.

Toutefois, une telle librairie n'est pas disponible en langage C, c'est pourquoi nous souhaitons dans ce projet proposer une alternative en développant une librairie écrite en langage C et qui permet de réaliser quelques unes des fonctionnalités existantes sur **Pandas**.

# 2. Introduction

L'objectif de ce projet est de créer un ensemble de fonctions en langage C (appelées communément une librairie) qui permettent de faciliter la manipulation de données.

Pour ce faire, il est important de comprendre le fonctionnement d'une feuille de calcul dans un tableur ou encore comprendre le fonctionnement d'un **"DateFrame"** dans **Pandas**.

En effet, cette structure est composée de cellules organisées sous forme d'un tableau 2D (matrice). Toutefois, une utilisation désordonnée de ces cellules peut vite devenir un casse tête. Il est donc important que l'utilisateur organise lui même les cellules afin de correspondre à une abstraction du problème qu'il souhaite résoudre. Il existe probablement plusieurs façons d'organiser des données et que les tableurs peuvent être utilisés pour d'autres fins telles que la création des emplois du temps ou l'élaboration de menus hebdomadaires, etc. mais dans ce projet nous souhaitons adopter l'idée de les organiser comme étant un ensemble de colonnes qui offrent des fonctionnalités similaires à celles qu'offre le **"DateFrame"** dans **Pandas**.

# 3. Description du projet

Dans ce projet, nous allons implémenter une structure composée d'un ensemble de colonnes, appelée : **CDataframe**. Chaque colonne a un titre et permet de stocker un nombre indéfini de données de même type. Toutes les colonnes doivent avoir le même nombre de données afin de former un matrice. Si des données sont manquantes alors elle sont remplacées par des valeurs par défaut que l'on définira plus tard.

Une fois la structure créée, nous souhaitons pouvoir offrir à l'utilisateur la possibilité d'effectuer au minimum l'ensemble des fonctionnalités suivantes :

1. Alimentation
   - Création d'un CDataframe vide
   - Remplissage du CDataframe à partir de saisies utilisateurs
   - Remplissage en dur du CDataframe
2. Affichage
   - Afficher tout le CDataframe
   - Afficher une partie des lignes du CDataframe selon une limite fournie par l'utilisateur
   - Afficher une partie des colonnes du CDataframe selon une limite fournie par l'utilisateur
3. Opérations usuelles
   - Ajouter une ligne de valeurs au CDataframe
   - Supprimer une ligne de valeurs du CDataframe
   - Ajouter une colonne au CDataframe
   - Supprimer une colonne du CDataframe
   - Renommer le titre d'une colonne du CDataframe
   - Vérifier l'existence d'une valeur (recherche) dans le CDataframe
   - Accéder/remplacer la valeur se trouvant dans une cellule du CDataframe en utilisant son numéro de ligne et de colonne
   - Afficher les noms des colonnes
4. Analyse et statistiques
   - Afficher le nombre de lignes
   - Afficher le nombre de colonnes
   - Nombre de cellules égales à $x$ ($x$ donné en paramètre)
   - Nombre de cellules contenant une valeur supérieure à $x$ ($x$ donné en paramètre)
   - Nombre de cellules contenant une valeur inférieure à $x$($x$ donné en paramètre)

Afin de simplifier la réalisation de ce projet, nous allons procéder par étapes en décomposant le travail en 3 parties :

- **Partie 1 :** La structure ne contiendra que des données de type "entier", organisées en colonnes où chacune des colonnes va avoir un titre.
  Sur cette structure, nous devons pouvoir appliquer au mimimum toutes les fonctionnalités décrites ci-dessus.

- **Partie 2 :** Dans cette partie, il est demandé d'étendre le travail précédent sur deux volets :
  - Développer de nouvelles fonctionnalités en plus des opérations de base réalisées dans la partie 1
  - Permettre d'organiser des données de types différents dans un même CDataframe où chaque colonne doit avoir des données de même type, mais que deux colonnes différentes peuvent avoir des données de types différents tel qu'illustré dans la figure 1 ci-dessous :

| | Colonne 1 | Colonne 2 | | Colonne n |
|---|---|---|---|---|
| Titre | Place | Code | .... | Indice-p |
| | 52 | Lima | | 1.158 |
| | 44 | Bravo | | 6.135 |
| | 15 | Zulu | | NULL |
| | 18 | Tango | | NULL |

FIGURE 1 – Exemple d'un tableau de données.

**Ainsi, il sera possible pour les élèves qui ne saurons pas implémenter la nouvelle structure de la colonne de continuer à travailler avec la première structure tout en l'enrichissant avec de nouvelles fonctionnalités.**

— **Partie 3 :** Implémenter le CDataframe comme étant une liste doublement chaînée et permettre le chargement de données à partir/dans un fichier **.csv**.

# Première partie
# Un CDataframe d'entiers

## 4. CDataframe

Nous souhaitons stocker des données en colonnes, chaque colonne dispose d'un entête et des données. Toutes les données stockées dans la colonne sont du même type.

Pour stocker nos données il nous faut une structure qui va être à la fois bien solide pour permettre de contenir facilement un grand nombre de données , et flexible afin de permettre d'ajouter, supprimer, déplacer les colonnes ou les lignes.

La structure globale telle qu'elle a été décrite ressemble beaucoup à un tableau 2D. Néanmoins, l'ajout de titres aux colonnes rend l'utilisation d'un tableau impossible car en langage C, un tableau ne peut contenir des données de types différents. Il est donc nécessaire de penser à un type structuré pour une colonne qui lui permettra d'être représentée par un titre et un tableau d'entiers. Pour qu'enfin, le CDataframe ne sera autre qu'un tableau de colonnes comme illustré sur la Figure 2.

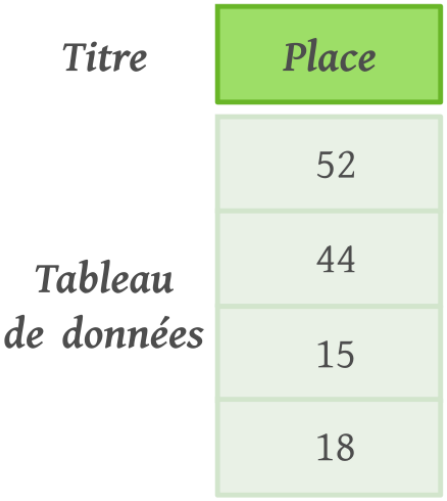


FIGURE 2 – Description d'une colonne.

### 4.1. Les colonnes

Une structure "column" contient le titre de la colonne qui est une chaîne de caractères et des données de type entier dans cette partie. Pour un accès rapide aux données (accès direct) nous allons utiliser des tableaux. Étant donné que nous ne connaissons pas la taille préalable de nos données, nous aurons donc besoin de sa taille physique (le nombre de cases à allouer dynamiquement) et de sa taille logique (le nombre de valeurs insérées dans la colonne).

Lorsque le nombre de valeurs insérées (taille logique) a atteint le nombre maximum de cases allouées (taille physique), une réallocation d'espace est requise. Cependant, pour éviter d'effectuer des réallocations répétitives (case par case) qui sont coûteuses en temps, nous allons opter pour la réallocation d'un autre bloc de 256 cases.

```
#define REALOC_SIZE 256
```

Cette structure "column" est donc le premier élément à définir dans votre projet.

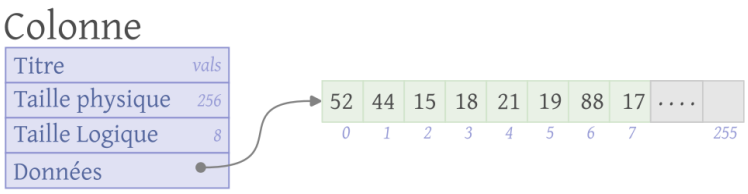


FIGURE 3 – Colonne d'un CDataframe.

### 4.1.1. Créer une colonne

Cette fonction permet de créer dynamiquement une colonne vide à partir d'un titre. La fonction ne doit pas allouer d'espace pour les données du tableau mais initialiser uniquement le pointeur sur le tableau d'entiers à $NULL$. Elle doit également initialiser l'ensemble des attributs de la colonne et retourner un pointeur sur la colonne créée.

**Prototype de la fonction :**

```
/**
 * Create a column
 * @param1 : Column title
 * @return : Pointer to created column
 */
COLUMN *create_column(char* title);
```

**Exemple d'utilisation :**

```
COLUMN *mycol = create_column("My column");
```

### 4.1.2. Insérer une valeur dans une colonne

Cette fonction permet d'insérer une valeur dans une colonne, toutes les deux sont données en paramètres. La fonction doit pouvoir vérifier la disponibilité de l'espace physique avant insertion. Dans le cas où le nombre de cases allouées est $NULL$ ou épuisé, la fonction doit pouvoir déclencher une allocation ou une ré-allocation de 256 cases supplémentaires avant d'insérer la valeur donnée en paramètres. Ainsi, si l'insertion s'est bien effectuée, la fonction retourne 1, 0 sinon.

À l'issue de cette action, les attributs (taille physique, taille logique, etc.) de la colonne doivent être mis à jour.

**Prototype de la fonction :**

```c
/**
 * @brief : Add a new value to a column
 * @param1 : Pointer to a column
 * @param2 : The value to be added
 * @return : 1 if the value is added 0 otherwise
 */
int insert_value(COLUMN* col, int value);
```

**Exemple d'utilisation :**

```c
COLUMN *mycol = create_column("My column");
int val = 5;
if (insert_value(mycol, val))
    printf("Value added successfully to my column\n");
else
    printf("Error adding value to my column\n");
```

**Rappel :** La fonction `realloc` peut parfois changer l'emplacement du tableau de données dans le cas où la système ne trouve pas un espace contiguë assez grand pour étendre le tableau original. Aussi, lors de la première allocation c'est à dire quand la taille physique est égale à zéro, c'est la fonction `malloc` qu'il faut utiliser, car la fonction `realloc` ne permet pas de réserver de l'espace mais uniquement étendre un espace déjà existant.

### 4.1.3. Libérer l'espace allouée par une colonne

Cette fonction prend en paramètre une colonne et permet de libérer la mémoire qui a été allouée à son tableau de données et à elle même.

**Prototype de la fonction :**

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

### 4.1.4. Afficher le contenu d'une colonne

La fonction suivante, doit permettre d'afficher le contenu d'une colonne. Pour chaque ligne elle doit aussi afficher le numéro de la ligne (indice de la case dans laquelle se trouve la valeur à afficher) suivi de la valeur contenue dans cette case.

**Prototype de la fonction :**

```c
/**
 * @brief: Print a column content
 * @param: Pointer to a column
 */
void print_col(COLUMN* col);
```

**Exemple d'utilisation :**

```c
COLUMN *mycol = create_column("My column");
insert_value(mycol, 52);
insert_value(mycol, 44);
insert_value(mycol, 15);
print_col(mycol);
```

**Sortie :**

```
[0] 52
[1] 44
[2] 15
```

#### 4.1.5. Autres fonctions

En plus des fonctions précédentes, il faudra implémenter l'ensemble des fonctions qui permettent la réalisation des opérations suivantes :

- Retourner le nombre de d'occurrences d'une valeur $x$ ($x$ donné en paramètre).
- Retourner la valeur présente à la position $x$ ($x$ donné en paramètre).
- Retourner le nombre de valeurs qui sont supérieures à $x$ ($x$ donné en paramètre).
- Retourner le nombre de valeurs qui sont inférieures à $x$ ($x$ donné en paramètre).
- Retourner le nombre de valeurs qui sont égales à $x$ ($x$ donné en paramètre).

Il est à noter que d'autres fonctions utiles seront potentiellement à ajouter lorsque le besoin se présente dans la suite de ce projet.

### 4.2. Le CDataframe

Jusqu'à présent nous n'avons pas implémenté de fonctions qui permettent de créer un CDataframe, car ce dernier est composé de colonnes. Maintenant que nous avons nos briques de base, il est possible de concevoir un CDataframe qui n'est autre qu'un tableau dynamique de pointeurs vers des colonnes comme illustré sur la Figure 4.

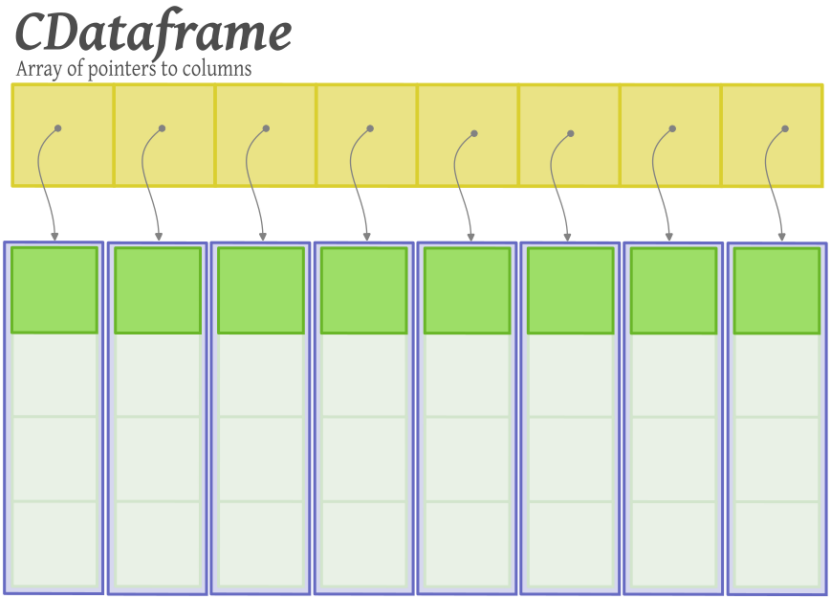


FIGURE 4 – Structure d'un CDataframe.

### 4.2.1. Fonctionnalités

Pour pouvoir utiliser le CDataframe, il faut implémenter un ensemble de fonctions qui se traduiront comme fonctionnalités que l'utilisateur pourra choisir au travers d'un menu dans son programme principal.

Les fonctionnalités de base que doit assurer votre CDataframe sont celles listées dans la description du projet, à savoir :

1. Alimentation
   - Création d'un CDataframe vide
   - Remplissage du CDataframe à partir de saisies utilisateurs
   - Remplissage en dur du CDataframe
2. Affichage
   - Afficher tout le CDataframe
   - Afficher une partie des lignes du CDataframe selon une limite fournie par l'utilisateur
   - Afficher une partie des colonnes du CDataframe selon une limite fournie par l'utilisateur
3. Opérations usuelles
   - Ajouter une ligne de valeurs au CDataframe
   - Supprimer une ligne de valeurs du CDataframe
   - Ajouter une colonne au CDataframe
   - Supprimer une colonne du CDataframe
   - Renommer le titre d'une colonne du CDataframe
   - Vérifier l'existence d'une valeur (recherche) dans le CDataframe
   - Accéder/remplacer la valeur se trouvant dans une cellule du CDataframe en utilisant son numéro de ligne et de colonne
   - Afficher les noms des colonnes
4. Analyse et statistiques

— Afficher le nombre de lignes
— Afficher le nombre de colonnes
— Nombre de cellules contenant une valeur égale à $x$ ($x$ donné en paramètre)
— Nombre de cellules contenant une valeur supérieure à $x$ ($x$ donné en paramètre)
— Nombre de cellules contenant une valeur inférieure à $x$($x$ donné en paramètre)

# Deuxième partie
# Un CDataframe presque parfait

Dans cette deuxième partie, l'idée est d'améliorer le CDataframe précédent, et ce, en agissant sur deux volets :

— Améliorer la structure de la colonne et du CDataframe pour offrir une utilisation plus large de celui-ci.
— Ajouter des fonctionnalités avancées

## 5. De nouvelles structures

Afin de permettre au CDataframe de stocker des données de types différents, il est nécessaire de modifier la structure de la colonne. En effet, il s'agira toujours d'un ensemble de colonnes. Seulement, nous souhaitons que les données d'une même colonne soient de même type mais que celles de deux colonnes différentes soient de deux types différents.

Pour obtenir cela, nous allons devoir utiliser **des types génériques**.

### 5.1. La colonne

Du point de vue "structure", une colonne contiendra essentiellement un titre, un tableau de données avec ses deux tailles physique et logique (car nous ne pouvons pas avoir sa taille préalablement).

De leur côté, les données stockées sur une colonne peuvent être de n'importe quel type. Il est donc nécessaire de créer un attribut supplémentaire qui indiquera pour chaque colonne créée, le type de données qu'elle contient. Dans ce projet, nous allons restreindre le nombre de types à l'ensemble suivant :

— Entiers naturels : $[0, 2^{32} - 1]$
— Entiers relatifs : $[-2^{31}, 2^{31} - 1]$
— Flottants simple précision : codé sur 32 bits
— Flottants double précision : codé sur 64 bits
— Caractère : codé sur 8 bits
— Chaînes de caractères : tableau de caractère
— Type quelconque : Type structuré

Puis, pour définir un type de colonne, nous allons créer une énumération de ces types comme suit :

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

Enfin, pour simplifier la réalisation de certaines fonctionnalités avancées (pour les élèves qui implémenteront les fonctionnalités avancées de cette deuxième partie), nous prévoyons un attribut **"index"** qui est défini comme étant un tableau d'entiers, et que nous allons mettre à $NULL$ dans cette partie.

Tous ces attributs font que la nouvelle structure d'une colonne sera représentée comme indiqué syr la Figure 5.

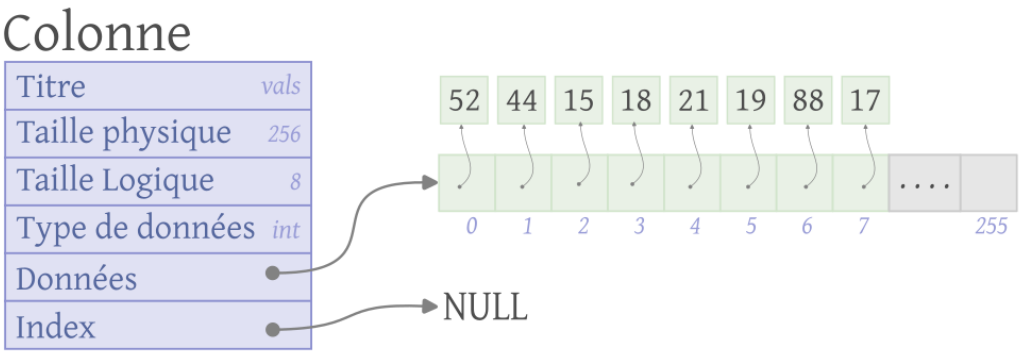


FIGURE 5 – Structure d'une colonne pouvant stocker n'importe quel type de données.

Nous remarquons à partir de ce schéma, que le tableau de données contient des pointeurs vers les données stockées (selon le type de la colonne). Ceci est volontaire afin de pouvoir représenter le manque d'information par la valeur *NULL*. En effet, on aurait pu représenter cela par la valeur 0. Seulement, le 0 ne peut être représentatif que pour des valeurs de type entier. De plus, la valeur 0 pourrait représenter une information significative que l'on souhaite stocker.

### Quel sera le type du tableau de données ?

Pour permettre à notre tableau de données de s'adapter au type de données qui sera stocké dans la colonne, il est nécessaire d'opter pour une solution qui apportera de la souplesse. L'un des outils qu'offre le langage C pour réaliser cela est l'**union**.

L'union est conceptuellement identique à une structure étant donné qu'elles permettent toutes les deux à l'utilisateur de combiner différents types sous un seul nom. Elles sont toutefois, différentes dans la façon dont la mémoire est allouée à leurs membres. Sur ce lien, vous allez trouver toutes les informations utiles pour mieux comprendre l'utilisation d'une **union**.

Pour notre tableau données, nous allons donc définir le nouveau type **union** suivant :

```
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 ;
```

### Mais que signifie (void*) dans cette union ?

`(void *)` désigne un pointeur générique. C'est un outil offert également par le langage C pour implémenter un type générique. Une variable de type `(void *)` signifie qu'elle représente un pointeur sur n'importe quelle variable de n'importe quel type. Attention, il ne peut pas contenir de valeurs, mais seulement des adresses (car les adresses font toutes la même taille). Ensuite, pour obtenir un pointeur sur le type souhaité, il faut aupréalable *caster* (forcer son type) le pointeur `(void *)`.

Ainsi, la structure et le type COLUMN seront définis en langage C comme suit :

```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. Créer une colonne

Cette fonction permet de créer une colonne d'un type et d'un titre donnés en paramètres. C'est-à-dire allouer la mémoire nécessaire pour stocker uniquement une colonne vide (sans données), lui affecter un titre et aussi initialiser l'ensemble de autres attributs. Cette fonction va se contenter d'affecter la valeur *NULL* au pointeur *data*, elle ne va pas allouer d'espace mémoire pour stocker les données. Également, elle doit retourner un pointeur sur la colonne nouvellement créée ou bien un pointeur `NULL` si l'allocation de création a échoué.

**Prototype de la fonction :**

```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);
```

**Exemple d'utilisation :**

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

### 5.1.2. Insérer une valeur dans une colonne

La fonction suivante permet d'insérer une valeur dans une colonne. Cette fonction est générique, c'est-à-dire qu'elle va permettre d'insérer une valeur de n'importe quel type (dont l'adresse est donnée en paramètre) dans le tableau de données `data`. La fonction retourne 1 si l'insertion s'est bien effectuée, 0 sinon.

Le pointeur `(void *)` pointe sur une valeur quelconque, ce qui signifie que la fonction doit d'abord convertir cette valeur dans le même type que celui de la colonne dans laquelle on cherche à l'insérer (mycol). Si le pointeur `value` est *NULL* alors la valeur est égale à `NULL` mais elle doit quand même être insérée pour indiquer une absence de valeur.

La fonction `insert_value(COLUMN *col, void *value)`, doit vérifier si la taille logique du tableau de données n'a pas atteint sa taille maximale (même valeur que la taille physique). Dans le cas contraire, la fonction doit augmenter la taille physique en réallouant 256 octets supplémentaire et mettre à jour la valeur de la taille physique.

**Rappel :** La fonction `realloc` peut parfois changer l'emplacement du tableau de données dans le cas où la système ne trouve pas un espace contiguë assez grand pour étendre le tableau original. Aussi, lors de la première allocation c'est à dire quand la taille physique est égale à zéro, c'est la fonction `malloc` qu'il faut utiliser, car la fonction `realloc` ne permet pas de réserver de l'espace mais uniquement étendre un espace déjà existant.

**Prototype de la fonction :**

```
/**
  * @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);
```

**Exemple d'utilisation :**

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

Notez que la fonction prend en paramètre un pointeur sur la valeur à insérer. Il ne faut pas l'insérer directement dans le tableau de données comme le montre le code suivant :

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

La méthode correcte est d'abord d'allouer de l'espace qui va la contenir, puis de lui affecter la valeur à stocker. Pour vous aider à faire cela on vous propose l'exemple suivant :

```
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. Libérer l'espace alloué par une colonne

Une fonction qui permet de libérer la mémoire allouée par une colonne. Attention cette fonction doit libérer tous les espaces mémoire alloués par les attributs de la colonne avant de libérer l'espace de la colonne.

**Prototype de la fonction :**

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

### 5.1.4. Afficher une valeur

L'affichage d'une valeur en langage C passe par l'utilisation de la fonction **"printf"** qui exige de connaître le type de la donnée afin de lui associer le bon format ($\%d, \%c, \%s, ...$).

Néanmoins, dans notre cas, une colonne peut contenir n'importe quel type. Un affichage de ses valeurs, exige alors de récupérer le type de données stockées dans la colonne et d'adapter le code C (en utilisation l'instruction *switch...case*) afin de les afficher correctement.

Pour éviter cette implémentation qui s'avère longue, nous proposons dans ce projet de prévoir une fonction qui convertit n'importe quelle donnée de la colonne et de n'importe quel type en une chaîne de caractères. Ainsi, lors de l'affichage, il seul le format "$\%s$" sera suffisant.

Le prototype de cette fonction est le suivant :

```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);
```

Voici dans la Figure 6 un exemple illustrant l'effet de cette fonction sur une valeur stockée dans une colonne de $INT$ :

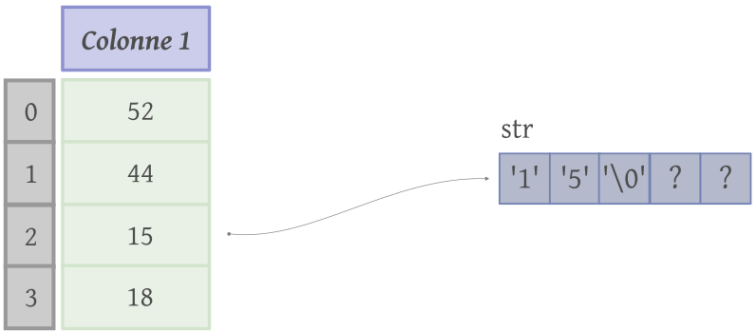


FIGURE 6 – Convertir le contenu de la ligne 3 (indice 2) en une chaîne de caractère.

**Exemple d'utilisation :**

```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);
...
```

**Sortie du programme :**

```
15
```

**Astuce** :

Pour implémenter cette fonction, vous pouvez vous aider de la fonction snprintf défini dans `<stdio.h>` de la librairie standard C.

**Exemple implémentassions :**

```
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;
        ...
    }
    ...
}
```

**Il est demandé de réfléchir à une fonction qui permet de convertir une donnée de type structuré en une chaîne de caractère.**

### 5.1.5. Afficher le contenu d'une colonne

La fonction suivante, doit permettre d'afficher le contenu d'une colonne. Pour chaque ligne elle doit aussi afficher le numéro de la ligne (indice de la case dans laquelle se trouve la valeur à afficher) suivi de la valeur contenue dans cette case. Si la valeur est nulle ($NULL$) alors elle doit pouvoir afficher la chaîne de caractères $NULL$.

**Prototype de la fonction :**

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

**Exemple d'utilisation :**

```
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);
```

**Sortie :**

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

### 5.1.6. Autres fonctions

En plus des fonctions précédentes, il faudra implémenter l'ensemble des fonctions qui permettent la réalisation des opérations suivantes :

- Retourner le nombre de d'occurrences d'une valeur $x$ ($x$ donné en paramètre).
- Retourner la valeur présente à la position $x$ ($x$ donné en paramètre).
- Retourner le nombre de valeurs qui sont supérieures à $x$ ($x$ donné en paramètre).
- Retourner le nombre de valeurs qui sont inférieures à $x$ ($x$ donné en paramètre).
- Retourner le nombre de valeurs qui sont égales à $x$ ($x$ donné en paramètre).

Il est à noter que d'autres fonctions utiles seront potentiellement à ajouter lorsque le besoin se présente dans la suite de ce projet.

## 5.2. Le CDataframe

Vous avez deux choix :

- Considérer un CDataframe comme étant un tableau de colonnes. Pour cela, il faudra suivre les instructions décrites dans la section 4.2.
- Considérer un CDataframe comme étant une liste doublement chaînée. Pour cela, il faudra suivre les instructions décrites dans la section 7.

Puis à partir de là, développer au minimum les fonctionnalités de base listées dans la description du projet.

# 6. Fonctionnalités avancées

Nous pouvons imaginer une multitude de fonctions avancées à ajouter au CDataframe en s'inspirant des fonctionnalités offerte dans le "Dataframe" de "Pandas" en Python.

Nous détaillerons dans ce qui suit la fonctionnalité qui permettra de trier des colonnes du CDataframe, mais il est possible de prévoir d'ajouter une ou plusieurs autres fonctionnalités telles que la concaténation ou encore la jointure.

## 6.1. Trier une colonne

Trier une colonne permet d'afficher ses valeurs dans un certain ordre (croissant ou décroissant) et permet aussi de vérifier l'existence (recherche) d'une valeur dans une colonne très rapidement en utilisant la technique de recherche dichotomique.

Le tri d'une colonne implique la changement de position des éléments dans la colonne. Dans le cas où cette colonne appartient à un *CDataframe*, alors il faut aussi déplacer des valeurs dans d'autres colonnes ce qui devient inefficace et fastidieux. Plus embêtant encore, si un nouveau tri est effectué sur le même *CDataframe* selon une autre colonne, alors l'on perd le laborieux travail de tri effectué précédemment.

Pour résoudre ce problème d'une façon efficace, nous allons utiliser l'attribut **"index"** (tableau d'entiers) déjà prévu dans la structure de **COLUMN**. Celui-ci permet d'indiquer la position de chaque élément dans la colonne. Ainsi, dans un CDataframe, il y aura un index associé à chaque colonne comme illustré dans les Figures 7 et 8.

| *Colonne 1* | | *Colonne 2* | | .... | *Colonne n* | |
|---|---|---|---|---|---|---|
| 3 | 52 | 1 | Lima | | 0 | 1.158 |
| 2 | 44 | 0 | Bravo | | 2 | 9.135 |
| 0 | 15 | 3 | Zulu | | 1 | 6.588 |
| 1 | 18 | 2 | Tango | | 3 | 13.52 |

FIGURE 7 – État du CDataframe avec les index associés aux colonnes.

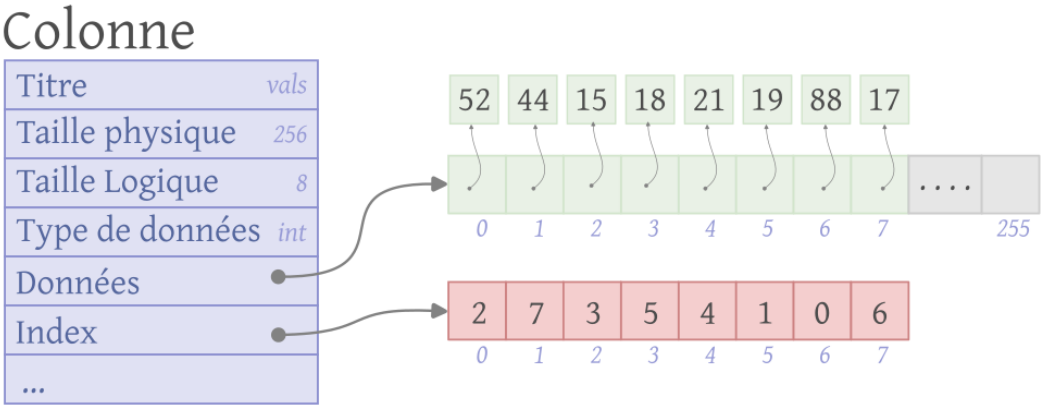


FIGURE 8 – Structure d'une colonne utilisant l'index.

### Comment cela va fonctionner ?

Initialement, la valeur de l'index attribuée à chaque donnée dans la colonne correspond au numéro de la ligne (en commençant par la valeur 0).

L'idée consiste en l'utilisation de cet index pour aider au tri du CDataframe de manière efficace. En fait, la méthode triviale à laquelle l'on peut penser naturellement serait d'appliquer un algorithme de tri sur une des colonnes du CDataframe. Or un algorithme de tri impose des permutations de valeurs sur une même colonne et pour une cohérence du CDataframe, il deviendrait nécessaire d'appliquer les mêmes permutations sur les autres colonnes du CDataframe. Si de plus, un tri doit se faire sur une autre colonne, alors le premier tri serait annulé.

L'association d'index aux colonnes du CDataframe apporte deux avantages :

— Pour trier une colonne, il ne sera plus obligatoire de permuter ses données. Il suffira de permuter les valeurs dans le tableau de l'index ;
— Pour trier plusieurs colonnes du CDataframe, il suffira de trier les index associés à chacune d'entre elles sans avoir à bouger les données de leurs cases initiales.

### Comment faire si une nouvelle donnée est insérée après une action de tri ?

Lorsqu'une colonne est triée, l'insertion d'une nouvelle donnée nécessite l'attribution d'un numéro séquentiel qui ne correspondra pas forcément à l'ordre de tri déjà établi. Afin d'éviter de re-trier la colonne, l'on se propose d'ajouter d'autres attributs à la structure **COLUMN** pour déterminer l'état de l'index. Les attributs à ajouter sont comme suit :

— **valid_index** : qui est un attribut qui peut prendre 3 valeurs. Il vaut 0 si la colonne n'est pas du tout triée, -1 si la colonne est partiellement triée (cela arrive lors de l'insertion d'une nouvelle valeur. Dans ce cas toute la colonne est triée sauf la dernière valeur.), 1 si la colonne est bien triée.
— **sort_dir** : c'est un attribut qui indiquera le sens de tri effectuée (ascendant ou descendant). Il prend 0 pour l'ordre ascendant, 1 sinon.
— **index_size** :

La structure de la colonne sera donc étendue comme suit :

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

**À noter que quelque soit la structure "COLUMN" utilisé précédent, l'ajout de ces nouveaux attributs liés à l'index n'impactera que la fonction de création d'une colonne où il faudra compléter leur initialisation.**

## 6.2. Trier une colonne

Pour trier une colonne, un algorithme de tri efficace est requis. Parmi les algorithmes de tri les plus efficaces l'on trouve :

— Le tri rapide (Quicksort) qui fonctionne bien si le tableau n'est pas trié ;
— Le tri par insertion qui fonctionne mieux que l'algorithme *Quicksort* dans le cas où le tableau est presque trié.

Pour avoir une meilleure performance de notre tri, l'on souhaite tirer partie de ces deux techniques. Ainsi, si l'attribut **valid_index** est à 0 (colonne non triée), c'est l'algorithme *Quicksort* qui est à appliquer. Si l'attribut **valid_index** est à -1 (colonne partiellement triée), alors c'est le tri par insertion qui sera appliqué.

En plus des deux liens expliquant le fonctionnement des deux algorithmes de tri, voici ci-dessous leurs pseudo-algorithmes :

### Tri par insertion

**Algorithm 1:** Tri par insertion

**Data:** Un tableau $tab$ de taille $N$
**Result:** Le tableau $tab$ trié

```
for i ← 2 to N do
    k ← tab[i];
    j ← i − 1;
    while j > 0 and tab[j] > k do
        tab[j + 1] ← tab[j];
        j ← j − 1;
    end
    tab[j + 1] ← k;
end
```

### Tri rapide *(Quicksort)*

**Algorithm 2:** Tri rapide (Quicksort)

**Data:** Un tableau $tab[]$ de taille $N$
**Result:** Le tableau $tab[]$ trié

**Function** `Quicksort`($tab[]$, $gauche$, $droite$) :
  **if** $gauche < droite$ **then**
    $pi \leftarrow$ PARTITION($tab[]$, $gauche$, $droite$);
    `Quicksort`($tab[]$, $gauche$, $pi - 1$);
    `Quicksort`($tab[]$, $pi + 1$, $droite$);
  **end**
**Function** `Partition`($tab[]$, $gauche$, $droite$) :
  $pivot \leftarrow arr[droite]$;
  $i \leftarrow gauche - 1$;
  **for** $j \leftarrow gauche$ **to** $droite - 1$ **do**
    **if** $tab[j] \leq pivot$ **then**
      $i \leftarrow i + 1$;
      Échanger $tab[i]$ et $tab[j]$;
    **end**
  **end**
  Échanger $tab[i + 1]$ et $tab[droite]$;
  **return** $i + 1$;

Voici donc le prototype de la fonction de tri :

```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. Afficher le contenu d'une colonne triée

La fonction d'affichage d'une colonne précédemment implémentée permettait un affichage séquentiel des valeurs. Avec l'introduction de l'index, cette fonction ne remplira pas bien son rôle notamment lorsque la colonne est triée.

Il est donc demandé d'implémenter une autre fonction d'affichage qui permettra d'afficher les valeurs selon l'ordre séquentiel indiqué dans le tableau index.

**Prototype de la fonction :**

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

**Exemple d'utilisation :**

```
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);
```

**Exemple d'affichage :**

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

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

**Exemple d'utilisation avec des chaînes de caractères :**

```
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);
```

Exemple d'affichage :

```
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. Effacer l'index d'une colonne

Certes, la présence des index apporte une amélioration en performance sur les opérations de tri et de recherche d'information dans le CDataframe. Néanmoins, ils imposent un espace de stockage et une gestion supplémentaire. Il arrive donc de vouloir alléger cela en décidons de supprimer pour certaines colonnes le tableau d'index qui leur ai associé.

La fonction suivante permet de supprimer l'association d'un index à une colonne donnée. Cela implique la libération de la mémoire allouée au tableau des index, l'attribution de la valeur $NULL$ au pointeur *index* et la mise à jour de l'attribut *valid_index* qu'il faudra mettre à la valeur 0.

**Prototype de la fonction :**

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

## 6.5. Vérifier si une colonne dispose d'un index

Cette fonction permet de vérifier la présence d'un index en consultant l'état de l'attribut "**valid_index**. Elle retourne 0 si l'index n'existe pas, -1 si l'index est invalide, 1 sinon.

**Prototype de la fonction :**

```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. Mettre à jour un index

Cette fonction permet de mettre à jour un index, dans les faits il suffit juste d'appeler la fonction `sort`. Celle-ci peut être appelée lors de l'insertion d'une nouvelle valeur après avoir appliqué une opération de tri.

**Prototype de la fonction :**

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

## 6.7. Recherche dichotomique

Cette fonctionnalité permet de recherche une valeur donnée en paramètre dans une colonne triée.

**Prototype de la fonction :**

```
/**
 * @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);
```

# Troisième partie
# Un CDataframe parfait

Cette partie aborde des fonctionnalités avancées que l'on peut appliquer au CDataframe.

— Apporter plus d'efficacité en implémentant le CDataframe avec une liste linéaire chaînée au lieu d'un tableau de colonne.

— Ajouter la possibilité de charger les données depuis un fichier **.csv** pour éviter la saisie de plusieurs lignes de données.

— Permettre de stocker le contenu d'un CDataframe dans un fichier **.csv**.

## 7. Implémentation du CDataframe

Pour apporter plus de souplesse au stockage des colonnes du CDataframe, nous proposons de l'implémenter comme étant une liste chaînée.

L'idée est que chaque colonne soit stocké dans un maillon. Pour assurer un meilleur accès, nous pensons qu'une structure en liste bi-directionnelle serait l'idéale (Figure 9) où l'accès à la liste se fait à la fois à partir de l'entête que de la queue, et qu'à partir de chaque maillon, il est possible d'accéder à la fois au maillon suivant qu'au maillon précédent.

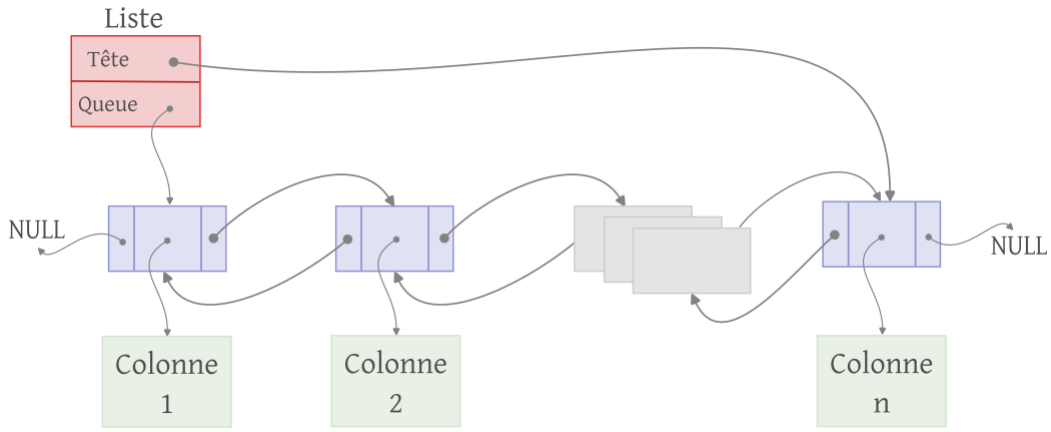


FIGURE 9 – Un CDataframe en liste chaînée.

Les structures nécessaires pour l'implémentation de la liste sont les suivantes :

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

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

```c
typedef LIST CDATAFRAME;
```

**Un implémentation d'une liste chaînée générique est donnée en annexe A. Libre à vous de réutiliser la liste donnée ou de créer votre propre implémentation.**

#### Conseil

Nous vous proposons de séparer votre code en quatre fichiers.

- **cdataframe (.h/.c) :** fonctions de manipulation du cdataframe. Seules ces fonctions sont utilisables par l'utilisateur final de la librairie. Donc le fichier entête correspondant doit être bien écrit et bien documenté.
- **column (.h/.c) :** fonctions de gestion des colonnes
- **list (.h/.c) :** liste chaînée donnée en annexe
- **sort (.h/.c) :** algorithmes de tri.

## 7.1. Fonctions à implémenter

### 7.1.1. Création du CDataframe

**Prototype de la fonction :**

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

**Exemple d'utilisation :**

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

### 7.1.2. Suppression d'un CDataframe

Proposer une fonction qui permet de supprimer un Dataframe. Cette fonction doit supprimer toutes les colonnes du CDataframe et libérer toute la mémoire allouée précédement avant de supprimer le CDataframee donné en paramarètre.

**Prototype de la fonction :**

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

### 7.1.3. Supprimer une colonne

Proposer une fonction qui permet de supprimer une colonne d'un CDataframe en indiquant le titre de la colonne.

**Prototype de la fonction :**

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

### 7.1.4. Compter le nombre de colonnes

Cette fonction permet de compter le nombre de colonnes dans le CDataframe.

**Prototype de la fonction :**

```
/**
 * @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. Autres fonctions

Si vous avez choisi d'utiliser cette structure du CDataframe, il faudra lui permettre d'offrir au minimum les fonctionnalités supplémentaires suivantes :

1. Alimentation
   - Création d'un CDataframe vide
   - Remplissage du CDataframe à partir de saisies utilisateurs
   - Remplissage en dur du CDataframe
2. Affichage
   - Afficher tout le CDataframe
   - Afficher une partie des lignes du CDataframe selon une limite fournie par l'utilisateur
   - Afficher une partie des colonnes du CDataframe selon une limite fournie par l'utilisateur
3. Opérations usuelles
   - Ajouter une ligne de valeurs au CDataframe
   - Supprimer une ligne de valeurs du CDataframe
   - Ajouter une colonne au CDataframe
   - Supprimer une colonne du CDataframe
   - Renommer le titre d'une colonne du CDataframe
   - Vérifier l'existence d'une valeur (recherche) dans le CDataframe
   - Accéder/remplacer la valeur se trouvant dans une cellule du CDataframe en utilisant son numéro de ligne et de colonne
   - Afficher les noms des colonnes
4. Analyse et statistiques
   - Afficher le nombre de lignes
   - Afficher le nombre de colonnes
   - Nombre de cellules égales à $x$ ($x$ donné en paramètre)
   - Nombre de cellules contenant une valeur supérieure à $x$ ($x$ donné en paramètre)
   - Nombre de cellules contenant une valeur inférieure à $x$($x$ donné en paramètre)

## 8. Fichiers

Lorsqu'on développe un outil de manipulation de données, il est important de lui permettre d'accéder aux fichiers (généralement le formation utilisé est le format (**CSV)** en lecture et en écriture afin de charger / stocker un grand nombre de données.

Un fichier CSV (Comma-Separated Values) est un format de fichier couramment utilisé pour stocker des données tabulaires, telles que des données de feuilles de calcul ou des bases de données. Il est largement utilisé car il est simple à comprendre et à manipuler. Dans un fichier CSV, chaque ligne du fichier représente une ligne de données, et les valeurs dans chaque ligne sont séparées par un délimiteur, généralement une virgule (","), bien que d'autres délimiteurs comme les points-virgules (";") ou les tabulations ("$\widehat{\text{}}$") puissent également être utilisés. Les valeurs nulles sont notées par `"NULL"` ou tout simplement laissées vides.

**Exemple de contenu d'un fichier data.csv :**

```
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
```

Afin d'éviter la saisie manuelle de données ou les initialisations en dur du CDataframe, nous souhaitons dans ce projet pouvoir charger les données à partir d'un fichier **.csv**. Aussi, lorsque des données sont saisies par l'utilisateur, nous souhaitons offrir la possibilité de les stocker dans un fichier **.csv**.

**Exemple d'utilisation :**

Affichage de la ligne 2 à la ligne 9 du fichier précédent :

```
ENUM_TYPE cdftype [] = {INT,STRING,FLOAT};
CDATAFRAME *cdf = load_from_csv("data.csv", cdftype, 3);
// exemple de fonction qui permet un affichage partiel du CDataframe
print_dataframe_by_line(df,2,9);
```

**Affichage attendu :**

```
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. Charger un CDataframe depuis un fichier CSV

Cette fonction permet de charger les données présentes dans un fichier CSV. Elle prend en paramètre le nom du fichier et un tableau de types qui décrit les types de chaque colonne du fichier CSV.

**Prototype de la fonction :**

```
/**
 * @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);
```

Vous trouverez sur ce lien les instructions nécessaires pour l'ouverture et la lecture d'un fichier CSV en langage C.

## 8.2. Exporter un CDataframe dans un fichier CSV

Cette fonction permet d'exporter un CDataframe sous forme d'un fichier CSV. Le format d'export est le même que le fichier d'import, c'est-à-dire que les colonnes sont séparées par des virgules ',' et les linges sont séparées par des saut de lignes ''. La première ligne représente le nom des colonne. Le chemin du fichier est donné sous forme d'une chaîne de caractère en paramètre.

```
/**
 * @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);
```

Vous trouverez sur ce lien les instructions nécessaires pour l'ouverture et l'écriture sur un fichier CSV en langage C.

# A. Implémentation d'une liste chaînée

## A.1. Fichier entête (`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. Fichier source (`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;
}
```