LISV Laboratoire d'ingénierie des systèmes de Versailles | UVSQ Université Paris-Saclay

Projet de programmation C

CDataframe

Halim Djerroud

Février 2025

Table des matières

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
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 :

Titre	Colonne 1	Colonne 2		Colonne n
	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.

— **Partie 3 :** Fichier CSV, listes chaînées et fonctionnalités avancées

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.

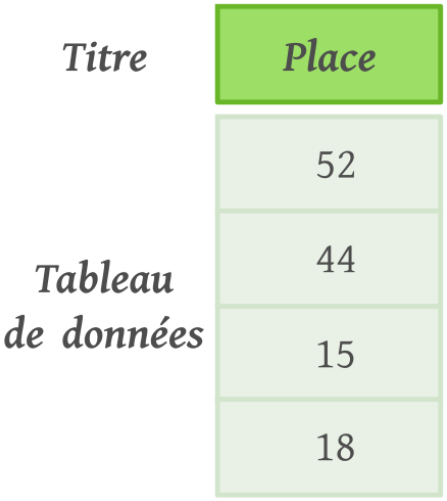

FIGURE 2 – Description d'une colonne.

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.

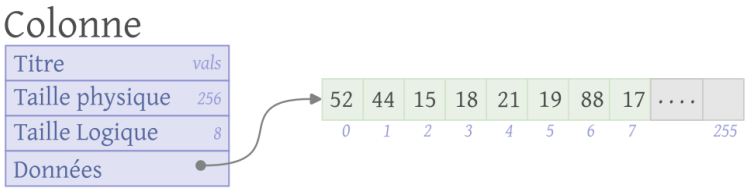

FIGURE 3 – Colonne d'un CDataframe.

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");
```

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 :

```
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.

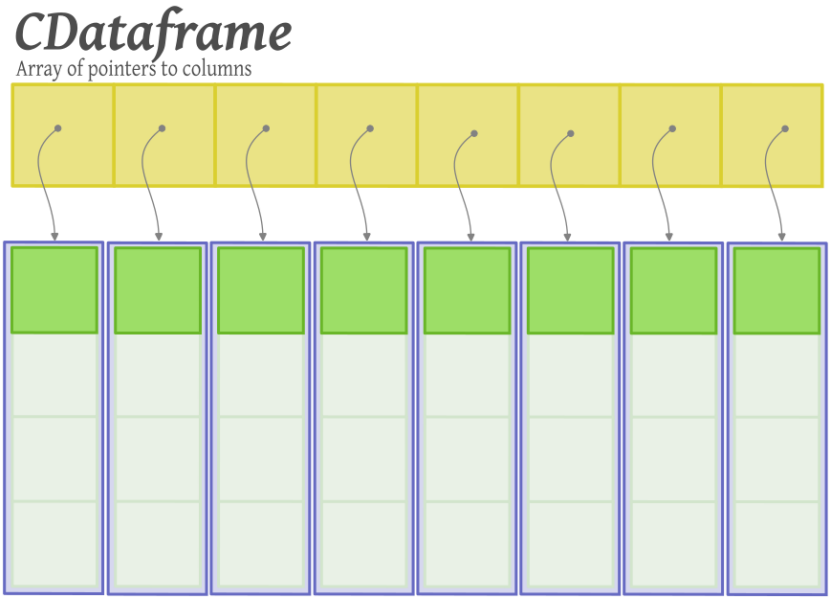

FIGURE 4 – Structure d'un CDataframe.

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
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.

5.0.1. La colonne

La figure 5 ci-après montre la structure d'une colonne.

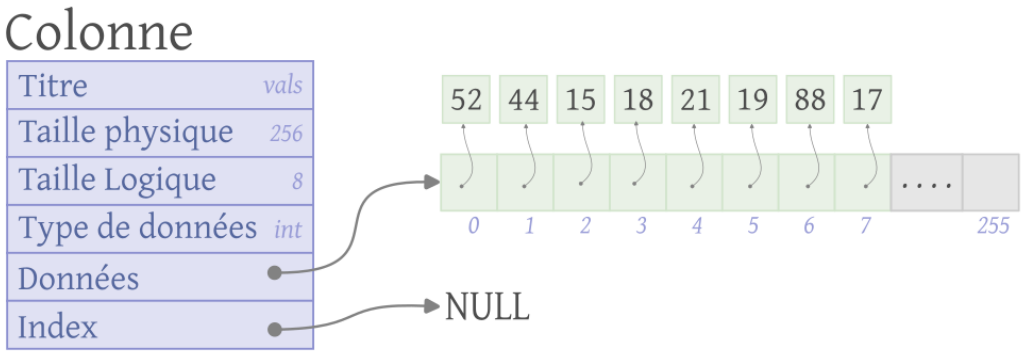

FIGURE 5 – Colonne.

Une colonne doit stocker des données de différents types nous allons commencer par énumérer les types possibles :

— entiers naturels : $[0, 2^{32} - 1]$
— entiers relatifs : $[-2^{31}, 2^{31} - 1]$
— petits entiers naturels : $[0, 2^{16} - 1]$
— petits relatifs : $[-2^{15}, 2^{15} - 1]$
— grands entiers naturels : $[0, 2^{64} - 1]$
— grands relatifs : $[-2^{63}, 2^{63} - 1]$
— très petits entiers naturels : $[0, 2^{8} - 1]$ (généralement utilisé pour stocker des caractères)
— très petits relatifs : $[-2^{7}, 2^{7} - 1]$

— flottants simple précision : codé sur 32 bits
— flottants double précision : codé sur 64 bits
— Chaînes de caractères : tableau de caractère
— Objet quelconque : structures

Les données stockées par une colonne peuvent être de n'importe quel type cité ci-dessus, mais une colonne ne mélange pas les types, c'est-à-dire, si une colonne est destinée à stocker des entiers alors elle ne stockera que des entiers ou une valeur `NULL` pour indiquer l'absence de valeur. Étant donnée que le langage C est typé, il nous faut créer une structure pour chaque type que nous souhaitons stocker, ce qui fastidieux. Heureusement le langage C propose un pointeur générique noté `(void *)` qui permet de pointer sur n'importe quel variable de n'importe quel type. Attention il ne peut pas contenir de valeurs, mais seulement des adresses (car les adresse font toutes la même taille) de plus il est interdit de dé-référencer un pointeur `(void *)`, il faut au préalable le *caster* (forcer son type) afin le rendre un pointeur sur un type bien connu. Pour cela nous avons besoin de connaître le type de valeurs que notre colonne doit stocker. Pour nous simplifier la tâche on peux utiliser une énumération comme suit :

```
enum enum_type {NULLVAL = 1 , UINT, INT, USHORT, SHORT, ULONG,
                LONG, UCHAR, CHAR, FLOAT, DOUBLE, STRING, OBJECT };
typedef enum enum_type Enum_Type;
```

Quand au données elle serons stockées dans un tableau de pointeurs de type `(void *)`. Mais une autre façon de faire une plus élégante est possible. Il existe en C les unions (notées `union`), elle permettent de stocker au même endroit différentes valeurs types, donc au lieu de mettre un pointeur `(void *)` on dire que notre tableau de valeur est un pointeur sur une union, comme illustré dans le code ci-après :

```
union column_type{
  unsigned int             uint_value;
  signed   int             int_value;
  unsigned short           ushort_value;
  signed   short           short_value;
  unsigned long long int   ulong_int;
  signed   long long int   long_int;
  unsigned char            uchar_value;
  signed   char            char_value;
  float                    float_value;
  double                   double_value;
  char*                    string_value;
  void*                    object_value;
};

typedef union column_type Column_Type ;
```

Donc une colonne peut être définie comme suit. L'index sert au tri de la colonne il sera traité dans les prochaines sections.

```
struct column {
  char* column_name;
  //logical size
  unsigned int size;
  //physical size
```

```
    unsigned int max_size;
    Enum_Type column_type;
    // pointer of array of values
    Column_Type** data;
    // array of index
    unsigned long long int *index;
    // index valid
    //  0 : no index
    // -1 : invalid index
    //  1 : valid index
    char valid_index;
    // index size
    unsigned int index_size;
    // sort direction Ascending or Descending
    // 0 : ASC
    // 1 : DESC
    char sort_dir;
};
typedef struct column Column;
```

5.1. Créer une colonne

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

Prototype de la fonction :

```
/**
 * Create a new column
 * @param1 : column type
 * @return : pointer to the new column
 */
Column* create_column(Enum_Type type);
```

Exemple d'utilisation :

```
Column* mycol = create_column(CHAR);
```

5.2. 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 quelle type (dont l'adresse est donnée en paramètre) dans le tableau de donnée `data` .

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

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  inser_value(Column* col, void * value);
```

Exemple d'utilisation :

```
Column* mycol = create_column(CHAR);
char a = 'A', c = 'C';
inser_value(mycol, &a);
inser_value(mycol, NULL);
inser_value(mycol, &c);
```

La fonction `inser_value(Column* col, void * value)`, doit vérifier si la taille logique du tableau de donnée n'a pas atteint sa taille maximale (même valeur que la taille physique). Dans le cas contraire la fonction doit augmenter la taille physique grâce à la fonction `void *realloc(void *ptr, size_t size);` et mettre à jour la valeur de la taille physique.

Attention : La fonction `void *realloc(void *ptr, size_t size);)` 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. Il faire attention aussi lors de la première allocation c'est à dire quand la taille physique est égale à zéro, c'est la fonction `malloc` il faut utiliser, car la fonction `realloc` ne permet pas de réserver de l'espace mais uniquement étendre un espace déjà existant.

On remarque que la fonction prends en paramètre un pointeur sur la valeur à insérer. Il ne faut pas directement insérer ce pointeur dans le tableau de donnée mais il faut veiller une créer une nous nouvelle valeur. Car l'utilisateur peux changer le contenu de sa valeur, mais ne souhaite pas forcément modifier le contenu de la valeur insérée précédemment. Voir l'exemple d'utilisation suivant :

```
Column* mycol = create_column(INT);
for(int i = 0 ; i < 100 ; i++){
    inser_value(mycol, &i);
}
```

Pour vous aider à faire cela on vous proposer par exemple suivant :

```
int inser_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++;
        ...
    }
```

5.3. Libérer l'espace allouée par une colonne

Une fonction qui permet de libérer la mémoire allouée par une colonne. Attention cette fonction doit libérer tout les espaces mémoire allouée par la colonne.

Prototype de la fonction :

```
    /**
     * @brief: free the space allocated by a column
     * @param1: positioner to the column
     */
     void delete_column(Column* col);
```

5.4. Afficher une valeur

La fonction suivante permet de de récupérer une valeur d'une colonne à une position donnée, dans une chaîne de caractère.

```
    /**
     * @brief: free the space allocated by a column
     * @param1: pointer to the column
     * @param2: The pointer of the value to retrieve
     * @param3: pointer to the string in which the value will be written
     * @param4: Maximum size of the string
     */
    void print_value(Column* col, unsigned long long int i, char* str, int size);
```

Par exemple dans la colonne illustrée dans l'exemple suivant :

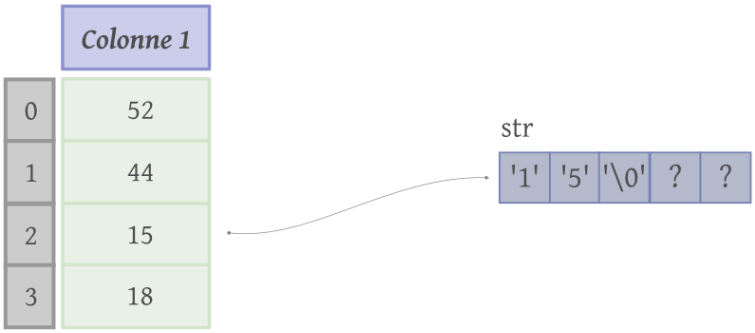

FIGURE 6 – Convertir le contenu de la ligne 2 dans une chaîne de caractère.

Exemple d'utilisation

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

Sortie du programme :

```
15
```

Astuce : Pour implémenter cette fonctions, vous pouvez vous aidez de la fonction `snprintf`, défini dans `string.h` de la librairie standard C.

Exemple implémentassions :

```
void print_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;
        ...
    }
    ...
}
```

5.5. Afficher le contenu d'une colonne

La fonction suivante, doit permettre d'afficher le contenu d'une colonne. Pour chaque ligne elle doit doit aussi afficher le numéro de ligne (numéro de la case dans la quelle elle se trouve) suivi de la valeur du contenu. Si la valeur est nulle (`NULL`) alors on se contenance d'afficher la chaîne de caractères `NULL`.

Prototype de la fonction :

```
/**
 * @brief: Display the contents of a column
 * @param: pointer to the column to display
 */
void print_col(Column* col);
```

Exemple d'utilisation :

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

Sortie :

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

5.6. Informations sur une colonne

Prototype de la fonction :

```
/**
 * @brief: Display the information of a column
 * @param1: pointer to the column to display
 */
void info_column(Column* col);
```

Exemple d'affichage :

```
xx
```

6. Trier une colonne

Trier un valeur des colonnes permet d'afficher les valeurs dans un certain ordre (croissant ou décroissant) et permet aussi de verrier 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 *dataframe* alors il faut aussi déplacer des valeurs dans d'autres colonnes ce qui devient inefficace et fastidieux. Plus embêtant encore si on re-tri un *dataframe* selon une autre colonne on perd le laborieux travail de tri effectué précédemment.

Pour résoudre ce problème d'une façon efficace on ajouter un index (tableau d'entier) qui nous indique la position de chaque élément dans la colonne. Ainsi il est possible d'avoir un index par colonne. Un exemple d'utilisation d'un index est illustré dans la la figure suivante.

Toute fois, cette technique à dés limites, par exemple lors d'insertion d'une nouvelle valeur l'index devient invalide.

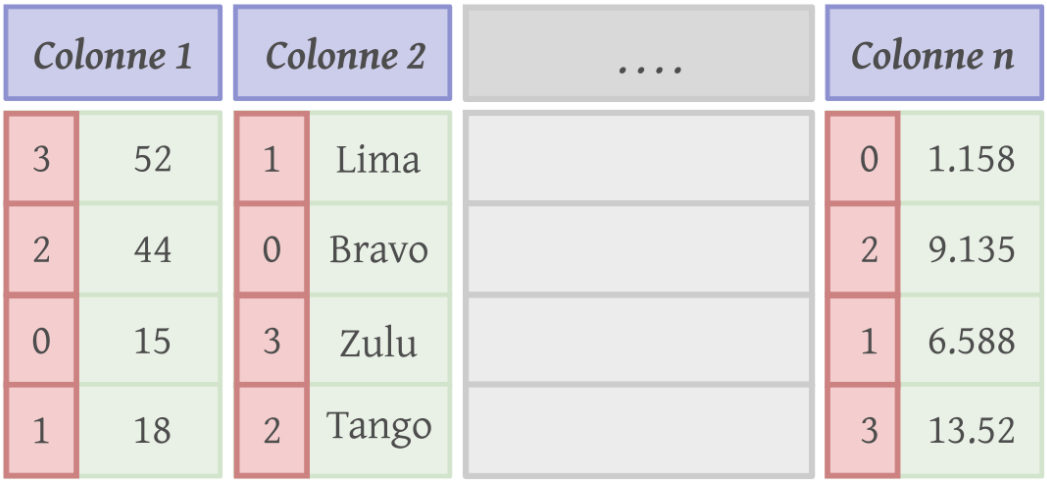

FIGURE 7 – Colonne.

Afin d'implémenter cette technique nous allons ajouter à notre colonnes un tableau

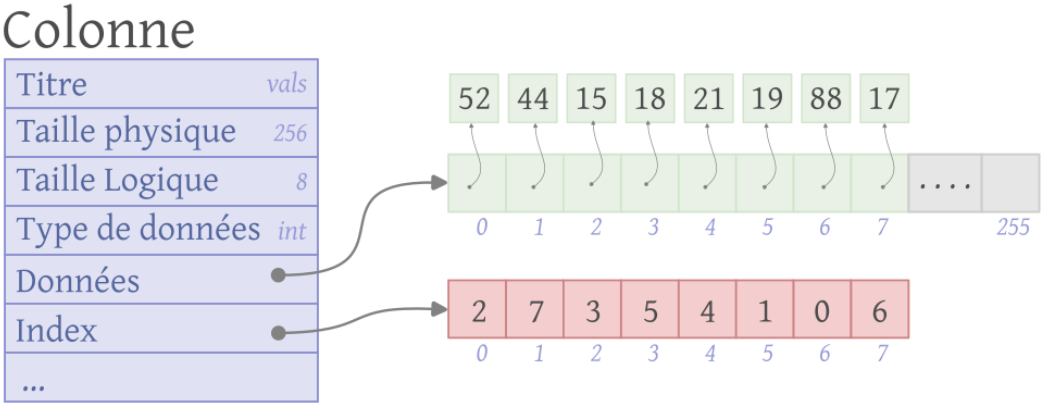

FIGURE 8 – Colonne.

On ajoute à notre définition de la colonne un tableau d'entier qui permet de stocker l'index, la taille de l'index et aussi une variable qui nous permet de nous indiquer l'état de l'index (pas d'index, index invalide, ou index valide).

— Pas index : signifie que la colonne n'est pas triée
— Index invalide : 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 colonne.
— Index valide : La colonne est bien triée

Il faut aussi indiquer le sens de tri, il existe deux sens de tri possible, croissant ou décroissant.

```
struct column {
  ...
  unsigned long long  *index;
  unsigned int index_size;
  // index valid
  //  0 : no index
  // -1 : invalid index
  //  1 : valid index
  char valid_index;
  char sort_dir;
};
```

6.1. Trier une colonne

La fonction suivant permet de trier une colonne selon un ordre croissant ou décroissant.

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

Pour trier une colonne on a besoin d'un algorithme de tri efficace. Parmi les algorithmes de tri les plus efficaces 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 meilleur performance de notre tri, on souhaite tirer parti de ces deux techniques. Par exemple si l'index est absent, c'est à dire que le colonne n'est pas trié alors on utilisera l'algorithme de tri rapide. Si l'index est invalide, ce qui arrive quand on insère une nouvelle valeur dans une colonne déjà trié, alors on utilisera le tri par insertion.

6.1.1. Tri par insertion

Algorithm 1: Tri par insertion

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

```
1 for i ← 2 to N do
2 |   k ← tab[i];
3 |   j ← i − 1;
4 |   while j > 0 and tab[j] > k do
5 |   |   tab[j + 1] ← tab[j];
6 |   |   j ← j − 1;
7 |   end
8 |   tab[j + 1] ← k;
9 end
```

6.1.2. Tri rapide *(Quicksort)*

Algorithm 2: Tri rapide (Quicksort)

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

```
1  Function Quicksort(tab[], gauche, droite) :
2  |   if gauche < droite then
3  |   |   pi ← Partition(tab[], gauche, droite);
4  |   |   Quicksort(tab[], gauche, pi − 1);
5  |   |   Quicksort(tab[], pi + 1, droite);
6  |   end
7  Function Partition(tab[], gauche, droite) :
8  |   pivot ← arr[droite];
9  |   i ← gauche − 1;
10 |   for j ← gauche to droite − 1 do
11 |   |   if tab[j] ≤ pivot then
12 |   |   |   i ← i + 1;
13 |   |   |   Échanger tab[i] et tab[j];
14 |   |   end
15 |   end
16 |   Échanger tab[i + 1] et tab[droite];
17 |   return i + 1;
```

Question de réflexion : *En utilisant uniquement l'algorithme de tri rapide, et si le pivot est bien choisi, il n'est pas nécessaire de faire l'implémentation du tri par insertion. Expliquer pourquoi ? et Expliquer quel est la meilleur position du pivot pour notre dataframe ?*

6.2. Afficher le contenu d'une colonne triée

Prototype de la fonction :

```
/**
 * @brief: display the contents of a sorted column
 * @param1: pointer to a column
 */
void print_col_by_index(Column* col);
```

Exemple d'affichage :

```
xx
```

6.3. Effacer l'index d'une colonne

La fonction suivante permet de supprimer un index, cela support la libération de la mémoire précédemment allouée et aussi la mise à jour de valeurs par défaut.

Prototype de la fonction :

```
/**
 * @brief: remove the index of a column
 * @param1: pointer to the column
 */
void erase_index(Column* col);
```

6.4. Vérifier si une colonne dispose d'un index

Cette fonction permet de vérifier si l'index d'une colonne.

Prototype de la fonction :

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

6.5. Mettre à jour un index

Cette fonction permet de mettre à jour un index, dans les fait il suffit juste d'appeler la fonction `sort`.

Prototype de la fonction :

```c
/**
 * @brief: update the index
 * @param1: pointer to the column
 */
void update_index(Column* col);
```

6.6. Recherche dichotomique

En utilisant la recherche dichotomique il faut vérifier si une valeur donnée en paramètre existe bien dans une colonne.

Prototype de la fonction :

```c
/**
 * @brief: Test if a value existe 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);
```

7. Bonus

7.1. Appliquer une fonction à une colonne

Cette fonction permet d'appliquer une fonction sur l'ensemble des éléments d'une colonne deux à deux. Cela permet par exemple de calculer toutes sort de choses par exemple la moyenne, la somme, l'écart type, etc.

```c
/**
 * @brief  : Tester l'existance d'une valeur dans une colonne
 * @param1 : pointeur sur la colonne
 * @param2 : pointeur sur la fonction à appliquer
 * @return : un poiteur sur la nouvelle valeur allouée qui contient le résultat
 */
void *applay_function_column_vertical(Column *col_1,
                                      Enum_Type dftype,
                                      void *(*fct)(void *, void *));
```

7.2. Appliquer une fonction à à plusieurs colonnes

Cette fonction permet d'appliquer une fonction sur deux colonnes er créer une nouvelle colonne.

```
/**
 * @brief  : Appliquer une fonction sur 2 colonne pour créer une 3eme
 * @param1 : pointeur sur la colonne 1
 * @param1 : pointeur sur la colonne 2
 * @param2 : pointeur sur la valeur a chercher
 * @return : Pointeur sur la nouvelle colonne crée
 */
Column* applay_function_column(Column* col_1,
                               Column*  col_2,
                               Enum_Type dftype,
                               void *(*fct)(void *,void*));
```

8. Implémentation d'un dataframe

Jusqu'à présent nous n'avons pas implémenté de fonctions qui permettent de créer un dataframe car, ce dernier est composé de colonnes. Maintenant que nous avons nos briques de bases, nous proposons d'implémenter les fonctions suivantes :

Nous pensons qu'une solution idéale serai de stocker les entêtes dans une liste chaînées, ce qui y remédierai aux inconvénients cités précédemment.

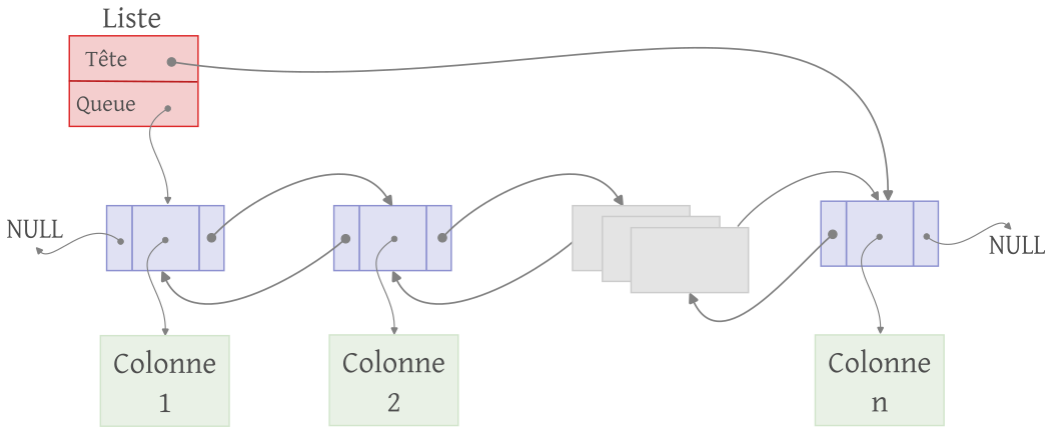

FIGURE 9 – Colonne.

Donc un *dataframe* ce n'est rien d'autre d'une liste chaînées, on peux la définir comme suit :

```
typedef list dataframe;
```

La liste chaînée quand à elle, elle peut être défini comme suit :

```
/**
 * Élément lnode
 */
typedef struct lnode_ {
    void *data;  // un pointeur vers une colonne de n'importe quel type
    struct lnode_ *prev;
    struct lnode_ *next;
} lnode;


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

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

Conseil *Nous vous proposons de séparer votre code en quatre fichiers.*

- **dataframe (.h/.c) :** fonctions de manipulation du dataframe. 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 :** fonctions de gestion des colonnes
- **list :** liste chaînée donnée en annexe
- **sort :** algorithmes de tri.

Une fonction qui permet de créer un *dataframe* vide.

Prototype de la fonction :

```
/**
 * Création d'un dataframe
 */
dataframe* create_dataframe(Enum_Type *dftype, int size);
```

Exemple d'utilisation :

```
Enum_Type dftype [] = {INT,CHAR,INT};
dataframe *df = create_dataframe(dftype, 3);
```

Prototype de la fonction :

```
/**
 * Suppression d'une colonne en indiquant son nom
 */
void delete_column(dataframe* df, char* col_name);
```

Prototype de la fonction :

```
/**
 * xxx
 */
int set_col_names(dataframe* df, char* tabs[], int size);
```

Prototype de la fonction :

```
/**
 * xxx
 */
int get_dataframe_cols_size(dataframe* df);
```

Prototype de la fonction :

```
/**
 * xxx
 */
int insert_line_dataframe(dataframe* df, void * velues[], int size);
```

Prototype de la fonction :

```
/**
 * xxx
 */
int get_dataframe_lines_size(dataframe* df);
```

Prototype de la fonction :

```
/**
 * Affichage des informations
 */
void info_dataframe(dataframe* df);
```

La fonction suivant va permettre d'afficher l'entête du *dataframe*, c'est à dire afficher uniquement le nom des colonnes, si le nom de la colonne n'est pas renseigné alors la on se contenance d'afficher la chaîne de caractère `"null"`. L'affichage se fera sur une seule ligne et les éléments sont séparés par des tabulations.

Prototype de la fonction :

```
/**
 * xxx
 */
void print_header_dataframe(dataframe* df);
```

La fonction suivante nous permettra d'afficher le contenu un *dataframe*. Plus exactement elle va nous permettre d'afficher des lignes consécutives d'un *dataframe*. La ligne de début et la ligne de fin sont donnée en paramètre. Par exemple l'appel de la fonction avec la ligne de départ *first* égale 5 et la ligne de fin *last* égale 10. La fonction affichera les lignes 5 incluse à 10 incluse. Dans le cas où les paramètres donnés débordent du *dataframe* alors uniquement des lignes réellement existante serons affichées.

Prototype de la fonction :

```
/**
 * xxx
 */
void print_dataframe_by_line(dataframe* df,
                             unsigned long long int fisrt,
                             unsigned long long int last);
```

Les trois fonctions suivantes sont des variantes de la fonction précédente. D'ailleur cette fonction peut tre réutilisée pour chacune de ces fonctions. La première fonction se contenante d'affichier l'ensemble du dataframe. La seconde fonction permet affichier uniquement les 10 premières lignes du dataframe donnée en paramètre. Finalement la dernièrte fonction affichera les 10 dernières lignes du dataframe.

Prototype des fonctions :

```
/**
 * xxx
 */
void print_dataframe(dataframe* df);

/**
 * xxx
 */
void print_head_dataframe(dataframe* df);

/**
 * xxx
 */
void print_tail_dataframe(dataframe* df);
```

Prototype de la fonction :

```
/**
 * xxx
 */
dataframe* load_from_csv(const char* file_name,
const Enum_Type *dftype,
int size);
```

Les fichiers tableur sont généralement stockés simplement dans des fichiers texte en prenons certaines précautions : Les lignes sont séparée par des sauts de lignes '' ou les points virgules ';'. La première ligne représente le titre des colonnes. Et finalement les valeurs sont séparées par des virgules ','. Les valeurs nulles sont notées par "NULL" ou tout simplement laissées vides.

Par exemple le fichier exemple ci-après représente les données du tableau montrée dans la Figure 8.

Exemple fichier CSV :

```
Place,Code,Indice-p
52,Lima,1.158
44,Bravo,6.135
15,Zulu,
18,Tango,
```

Prototype de la fonction :

```
/**
 * xxx
 */
dataframe* load_from_csv_auto(const char* file_name);
```

Exemple d'utilisation :

```
ddd
```

Sortie :

```
ddd
```

Cette fonction permet d'exporter un *dataframe* sous forme d'un fichier CSV. Le format d'exporter 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 chmin du fichier est donné sous forme d'une chaîne de caractère en paramètre.

```
/**
 * Exporter vers un fichier CSV.
 */
void save_into_csv(dataframe* df, const char* file_name);
```

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;
}
```