Projet Langage C

1 Modalité

Le projet est à faire par équipes de 2 étudiants.
Chaque équipe doit créer un dépôt git privé sur le gitlab de l'UFR

`https://gaufre.informatique.univ-paris-diderot.fr`

dès le début de la phase de codage et y donner accès en tant que 'Reporter' à tous les enseignants de C : Wieslaw Zielonka, Roberto Amadio, Pierre Charbit, Constantin Enea et Jules Chouquet. Le dépôt devra contenir un fichier 'equipe' donnant la liste des membres de l'équipe (nom, prénom, numéro étudiant et pseudo(s) sur le gitlab).
Vous devez utiliser le dépôt git de façon régulière pendant le développement du projet. Un projet avec très peu d'activité ou sans activité sur le dépôt git, où le projet complet et parfait apparaît à la veille de soutenance *ex nihilo*, sera traité avec une certaine suspicion. Et si dans une équipe un membre montre une activité soutenue sur git et l'activité de l'autre membre reste difficile à discerner, ce dernier sera interrogé de façon, disons, plus appuyée, pendant la soutenance.
La composition de chaque équipe devra être envoyée par mail à `zielonka@irif.fr` au plus tard **le 20 novembre 2019**, avec copie à chaque membre de l'équipe. Le mail devra avoir comme sujet `equipe projet C`.
Votre projet doit fonctionner sur les ordinateurs des salles de TP de l'UFR (2031 et 2032).
Le programme doit compiler avec les options `-Wall` (sans avertissement) et `-g` de `gcc`
En plus du programme demandé, vous devez fournir un `Makefile` pleinement fonctionnel. `make clean` devra supprimer tous les fichiers exécutables et les fichiers `*.o` de telle sorte que le `make` suivant permette de recompiler complètement le projet.
Si vous avez une question et que la réponse n'est pas contenue dans le présent document, merci de la poser sur le forum 'moodle' dédié du cours C. Seules les réponses postées sur ce forum feront foi au moment de la soutenance.

1.1 Soutenances

La soutenance se fera à partir du code contenu sur le gitlab. Pendant la soutenance chaque équipe devra cloner le projet et le compiler avec `make`. Si vous avez du mal à cloner le projet et que l'on passe 15 minutes à faire `clone` et `make`, vous aurez 15 minutes de moins pour la présentation de votre projet avec les conséquences évidentes sur la note. Donc vérifiez bien avant la soutenance que vous savez cloner votre projet sur une machine de l'UFR.
La date de la soutenance sera précisée ultérieurement (si les examens en L3 ont lieu en décembre, la soutenance du projet aura vraisemblablement lieu pendant la semaine du 6 janvier).

2 Listes chaînées : limiter le nombre de `malloc`

Le but du projet est d'implémenter des listes chaînées génériques en limitant le nombre d'appel à `malloc` et comparer l'efficacité de cette implémentation avec l'implémentation "classique".
La liste générique est une liste où l'on pourra stocker des données de n'importe quel type. Pour implémenter une telle liste il y a un problème à résoudre. Quand on définit un nœud d'une liste

```
struct node{
   struct node *next;
   struct node *previous;
   double data;
};
```

le compilateur ajoute dans la structure des octets de bourrage (padding) pour que le champ `data` soit placé dans la mémoire à l'adresse qui est multiple de `sizeof(double)`.
En fonction du type de données, on n'a pas les mêmes contraintes sur le placement en mémoire, les données de type `char` peuvent être stockées à n'importe quelle adresse, les données de type `int` peuvent être stockées à une adresse qui est multiple de `sizeof(int)`.
Par exemple sur les machines avec processeur Intel 64 bit, `sizeof(int)` est 8 donc les trois derniers chiffres de l'adresse d'une variable `int` sont toujours 0 (ce qui assure que l'adresse est un multiple de 8).
De même, pour le processeur Intel 64 bit `sizeof(long double)` est 16 donc l'adresse d'une variable de type `long double` doit être un multiple de 16 (4 derniers bits de l'adresse à 0). Notez que si l'adresse est multiple de 16 alors on pourra y mettre aussi bien un `int` qu'un `long double`.
Mais comment cela se passe pour `malloc` qui "ne sait pas" quelle donnée nous allons mettre dans la mémoire allouée ?
`malloc` retourne toujours l'adresse alignée de telle sorte qu'on peut mettre dans la mémoire allouée par `malloc` les données de n'importe quel type, il prévoit le pire de cas.
Dans la liste générique il faut aussi stocker les données de n'importe quel type, donc il faut s'assurer que l'adresse de données d'un nœud de la liste soit alignée de façon appropriée.
Nous définissons une `union` :

```
#include <stdint.h>

typedef union{
  intmax_t a;
  void *adr;
  long double c;
}align_data;
```

qui contient tous les types de données primitifs en C dont la taille peut être maximale [1].
Par la définition de `union` nous savons que si l'adresse de la mémoire est un multiple de `sizeof(align_data)` alors nous pouvons sans risque placer à cette adresse des données de tout type.
Sur la machine `lulu` de l'UFR

1. `intmax_t` est un type défini dans `stdint.h` qui désigne les entiers de taille maximale sur la machine

`http://www.informatique.univ-paris-diderot.fr/wiki/doku.php/wiki/linux`

(et les processeurs Intel 64 bit en général) `sizeof(align_data)` vaut 16.
Maintenant vous savez comment fait `malloc`, il retourne toujours l'adresse qui est multiple de `sizeof(align_data)` et grâce à ça, nous pouvons y mettre des données de tout type sans courir le risque de problèmes de non alignement d'adresse.
(Les même contraintes sur l'adresse de données impliquent aussi que la mémoire allouée par `malloc` soit d'au moins `sizeof(align_data)` octets même si vous demandez moins. En pratique, `malloc` stocke d'autres informations, donc la mémoire minimale réellement allouée par `malloc` est bien plus grande que `sizeof(align_data)`.)
Ceci nous ramène au sujet de ce projet. Le coût de `malloc` peut être non négligeable, surtout si l'on doit faire des dizaines de milliers ou de millions de `malloc`. L'idée est donc d'allouer une grande tranche de mémoire une fois pour toutes, et de gérer soit même le placement d'éléments de la liste chaînée dans cette mémoire.
La taille de la mémoire allouée sera un multiple de `sizeof(align_data)` et `malloc` assure que l'adresse du début de la mémoire allouée est aussi un multiple de `sizeof(align_data)`.
Un nœud de la liste chaînée sera représenté par la structure

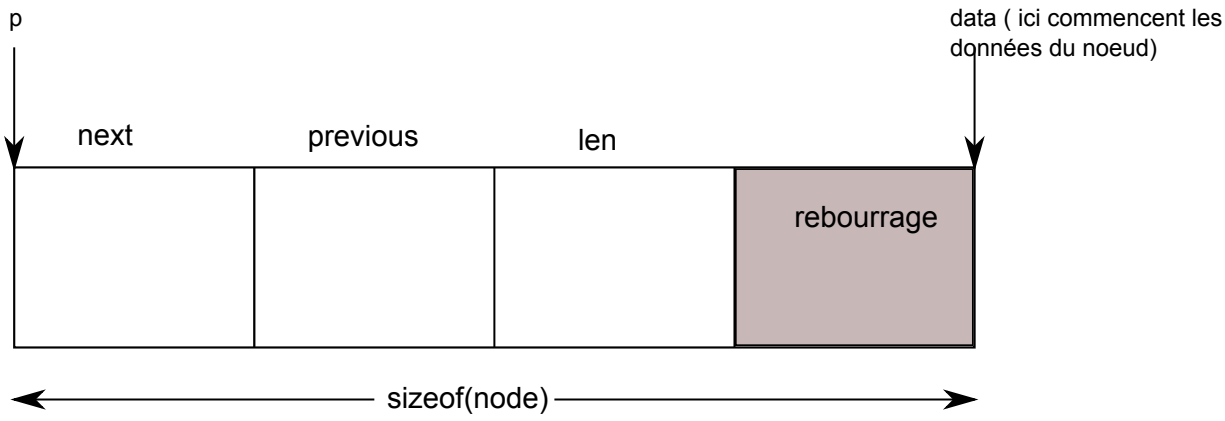
```
#include <stddef.h>
typedef struct{
  ptrdiff_t next;
  ptrdiff_t previous;
  size_t len;
  align_data data[];
}node;
```

Rappelons que le type `ptrdiff_t` est utilisé pour la différence de pointeurs.
Les champs `next` et `previous` permettront de retrouver les éléments suivant et précédent sur la liste [2], le champ `len` donne la taille du nœud. Nous précisons cela plus loin.
Le champ `data` est un tableau de taille 0 qui se trouve à la fin de `node`. Il est tout à fait légitime et utile d'avoir un tableau de taille 0 à la fin d'une structure, le champ `data` sert juste à obtenir le bon alignement de l'adresse de données.
Pour voir le placement de la structure `node` en mémoire, supposons qu'on utilise une machine avec un processeur Intel 64 bit (par exemple, `lulu`). Soit `p` l'adresse du (premier octet de) `node`. Cette adresse est un multiple de `sizeof(align_data)` (== 16 sur cette machine). Sur le même processeur `ptrdiff_t` et `size_t` prennent 8 octets chacun, donc les champs `next`, `previous` et `len` commencent respectivement à l'adresse :

- le champ `next` a la même adresse que `p`,
- `(char *)p + 8` pour `previous`,
- `(char *)p + 16` pour `len`.

où le retypage `(char *)` est fait pour que le calcul d'adresse soit en octets. L'adresse du champ `data`, même si `data` est de longueur 0, sera alignée sur un multiple de `sizeof(align_data)`. Pour obtenir une telle adresse après le champ `len` il y aura 8 octets de rembourrage (padding). De cette façon, le champ `data` de longueur 0 se situe à l'adresse `(char *)p + 32`.
Le dessin suivant représente un placement de `node` en mémoire.

2. on implémente une liste doublement chaînée



Sur d'autres processeurs, le placement des champs peut être différent, mais il y aura toujours à la fin un "trou" de rembourrage.
Pour le voir il suffit d'exécuter sur une machine de l'UFR [3] le code suivant :

```
node *p=malloc(sizeof(node));
printf( "%p %p %p %p %p\n", p, &p->next, &p->previous, &p->len, &p->data);
```

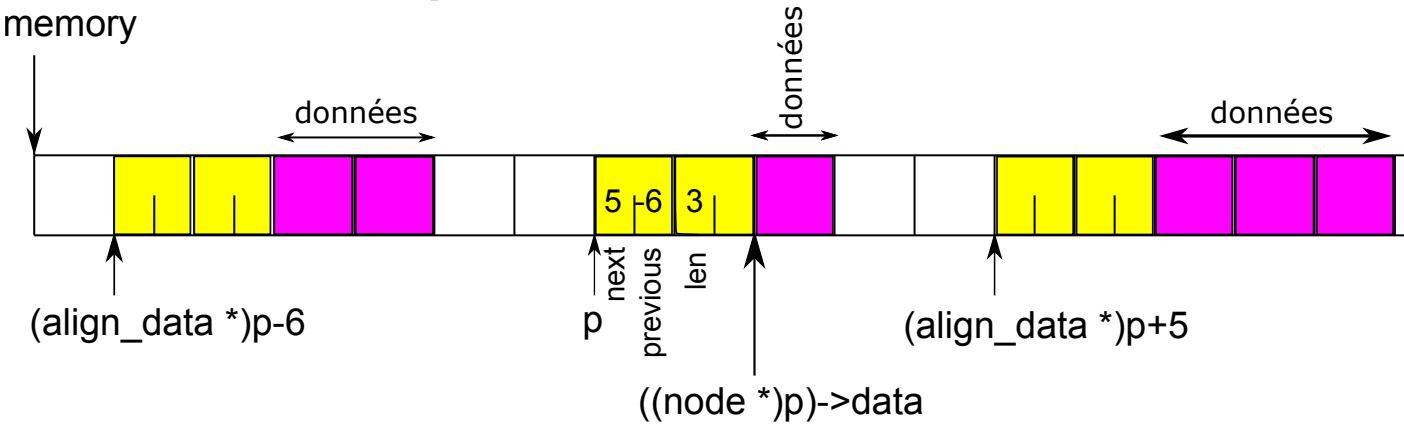
qui affiche les différentes adresses.
Pour la suite je suppose qu'on a une fonction

```
size_t nb_blocs(size_t o)
```

qui prend comme argument le nombre d'octets. La fonction `nb_blocs` calcule le nombre de blocs de taille
`sizeof(align_data)` suffisants pour stocker `o` octets (donc si `n == nb_blocs( o )` alors

`(n-1)*sizeof(align_data) < o <= n*sizeof(align_data)`).

Vous devez écrire vous même cette fonction.
Je suppose aussi que la mémoire pour stocker la liste a été allouée avec un seul appel à `malloc`, et si l'utilisateur a prévu k octets de mémoire, alors vous devez allouer `nb_blocs(n)*sizeof(align_data)` octets. Dans la suite, `memory` désigne le pointeur qui donne l'adresse de la mémoire allouée.
Le dessin ci-dessous représente une partie de la mémoire. Il y a trois nœuds de la liste visibles sur le dessin. Pour chaque nœud les blocs en violet sont de blocs de données.



3. sur votre portable ça doit être la même chose, il y a une grande chance que ce soit aussi l'architecture Intel 64 bit.

Chaque rectangle représente un bloc de taille `sizeof(align_data)` octets.
`p` est un pointeur vers un nœud courant. Le dessin présente aussi le nœud précédent et suivant de celui-ci.
Je suppose que c'est une machine avec architecture Intel 64 bit, le nœud courant possède une en-tête (en jaune) dont les valeurs des différents champs sont :

- `(node *)p -> next == 5`
- `(node *)p -> previous == -6`
- `(node *)p -> len == 3`

L'en-tête occupe 2 blocs sur ce processeur. L'en-tête est suivie d'un bloc de données [4].
Je suppose que le calcul se fait en nombre de blocs de taille `sizeof(align_data)`.
Les valeurs dans les champs `next` et `previous` nous informent que

- `(align_data *)p + 5` est l'adresse (du premier octets) du nœud suivant sur la liste, tandis que
- `(align_data *)p - 6` est l'adresse du nœud précédent.

Comment obtenir l'adresse du champ de données pour le nœud pointé par `p` ? Il y a trois possibilités :

(1) `(char *)p + sizeof(node)` – on prend l'adresse `p` et on se décale `sizeof(node)` octets à droite,

(2) `(node *)p + 1` – on prend l'adresse `p` et on se décale d'un nœud à droite,

(3) `(node *)p -> data` – puisque c'est le champ `data` d'un `node`.

Notez que l'on obtient chaque fois la même adresse, mais que le type de pointeur est chaque fois différent. Bien sûr, les retypages `(char *)` et `(char *node)` sont nécessaires uniquement si `p` est un autre type de pointeur.
Dans votre code vous ne devez rien supposer sur les valeurs `sizeof(align_data)` et `sizeof(node)`, la seule chose qui est toujours vraie est que `sizeof(node)` est un multiple de `sizeof(align_data)}` (mais pas forcement multiple de 2 comme sur l'architecture Intel 64 bit).

Il reste la question de ce qu'il faut mettre dans `next` dans le dernier nœud de la liste et dans `previous` dans le premier nœud de la liste. La seule possibilité raisonnable est d'y placer 0, pour tous les autres nœuds les valeurs de ces champs peuvent être positives ou négatives mais jamais 0.

2.1 Gestion de la mémoire libre

La mémoire libre, non occupée par les nœuds de la liste, peut être gérée de plusieurs manières différentes. L'algorithme d'allocation de mémoire pour un nœud et l'algorithme de libération de mémoire dépendent bien évidemment de votre gestion de la mémoire libre.
On donne ici quelques suggestions mais vous êtes libres de choisir comment vous gérez la mémoire libre.
On appellera **tranche** une suite de blocs libres consécutifs de la mémoire.

4. donc au total il y a 3 blocs dans ce nœud, d'où la valeur de champ `len`

2.1.1 Liste chaînée de tranches

On peut maintenir une liste chaînée de tranches en mettant au début de chaque tranche la structure

```
typedef struct{
   ptrdiff_t suivant;
   size_t    nb_blocs;
} entete_tranche;
```

Avec le champ `suivant` et le pointeur vers la tranche courante, on trouvera l'adresse de la tranche suivante, comme pour la liste de nœuds. Pas de champ `previous`, une liste simplement chaînée suffit. On peut supposer sans risque que
`sizeof(entete_tranche) <= sizeof(node)`
par contre rien ne permet d'assurer que
`sizeof(entete_tranche) <= sizeof(align_data)`.
Le champ `nb_blocs` nous renseigne sur le nombre de blocs dans la tranche courante.
Pendant l'allocation d'un nouveau nœud, on parcourt la liste de tranches libres pour trouver une tranche suffisamment grande pour accueillir le nouveau nœud. Après avoir mis le nouveau nœud dans la tranche, soit il reste assez de place dans la tranche pour obtenir une tranche plus petite soit toute la mémoire dans la tranche sera attribuée au nouveau nœud (comme dans l'exercice de TP).
Avec une liste chaînée de tranches il n'y a aucun moyen simple de fusionner deux tranches, même quand elles sont l'une à côté de l'autre dans la mémoire.
Par contre on pourra toujours fusionner les tranches à la demande de l'utilisateur, voir la fonction décrite dans la section 2.3.4.

2.1.2 Tableau de tranches

On maintient un tableau de structures de taille fixe, chaque structure permet de trouver l'adresse de la tranche et sa taille. Par exemple

```
#define NTRANCHES 1024
typedef struct {
   ptrdiff_t decalage;
   size_t    nb_blocs;
} tranche;
tranche *tab_tranches = malloc( NTRANCHES * sizeof(tranche) ) ;
```

Le champ `decalage` donnera le décalage de l'adresse de la tranche par rapport au début de la mémoire. En supposant que `memory` est l'adresse de la mémoire où on stocke la liste. Alors `(align_data *)memory + tab_tranches[i].decalage`
donnera l'adresse du début de la i-ème tranche (si le décalage est compté en blocs de taille `sizeof(align_data)`).
Le champ `nb_blocs` nous donnera le nombre de blocs libres dans la tranche.
Il est conseillé de maintenir le tableau `tab_tranches` trié dans l'ordre croissant de la valeur `decalage`.
Supposons qu'on supprime un nœud de la liste. Cela nous donne une nouvelle tranche. Avec la recherche binaire on trouve facilement la position où il faut mettre la nouvelle tranche dans le tableau `tab_tranches`. En plus ou pourra tout de suite vérifier si la

nouvelle tranche peut ou ne peut pas fusionner avec les tranches voisines. De cette façon il n'y aura jamais de tranches libres qui se touchent et qu'on n'arrive pas à fusionner.
Ici pour, simplifier les choses, j'ai assumé que le tableau `tab_tranches` est de taille fixe. Quand le nombre de tranches devient trop élevé nous sommes obligé de compactifier la mémoire à l'aide de la fonction `ld_compactify` de la section 2.3.4 Bien sûr il serait bien plus intéressant d'implémenter `tab_tranches` comme un tableau dynamique qui grossit en cas de besoin.
La structure `head` de la section 2.2.1 permettra de stocker soit un pointeur vers le tableau `tab_tranches` soit le tableau lui-même.

2.1.3 Pas de gestion de tranche libre

Enfin, pour ceux qui ont du mal à implémenter la gestion des tranches libres, il est toujours possible de ne pas le faire. On maintient juste l'adresse d'une seule tranche à la fin de la mémoire qu'on utilise pour allouer la mémoire pour un nouveau nœud de la liste.
Quand on supprime un nœud, on "oublie" la mémoire qu'il a occupée et on ne fait aucune tentative pour la réutiliser.
C'est une solution viable quand on alloue une grande mémoire initiale.
Cela peut-être aussi la première version de votre programme pour obtenir une implémentation simple mais fonctionnelle. Par contre, il est vraiment essentiel dans ce cas d'écrire la fonction de compactification, (section 2.3.4).

2.2 Fonctions à implémenter

Vous devez implémenter les fonctions décrites dans cette section. Vous pouvez ajouter d'autres fonctions qui peuvent vous aider dans votre tâche, mais les fonctions auxiliaires doivent être invisibles par l'utilisateur (elle doivent être `static`).
Toutes les fonctions doivent être regroupées dans un fichier `projet2019.c` – le nom du fichier ne doit pas être modifié. Vous créez le fichier `projet2019.h` correspondant avec les prototypes de ces fonctions. Vous n'avez pas de droit de modifier les noms des fonctions demandées. Le fichier `projet2019.c` ne doit pas contenir de `main()`, et dans le fichier `projet2019.c` on met uniquement les fonctions de gestion de listes.

2.2.1 ld_create

```
void * ld_create( size_t nboctets )
```

La fonction crée toutes les structures pour la liste. Vous définissez une structure

```
typed struct{
   void *memory;     //pointeur vers la memoire
   ptrdiff_t first; //ptrdiff_t ou pointeur
   ptrdiff_t last;  //ptrdiff_t ou pointeur
   ptrdiff_t libre; //ptrdiff_t ou pointeur si la liste de tranches
   //toute autre information
} head;
```

où vous devez stocker

- le pointeur `memory` vers la mémoire utilisée pour la liste.

- `first` et `last` – il faut stocker les adresses du premier et du dernier élément de la liste. Mais à la place d'adresses, on peut stocker le décalage par rapport à l'adresse `memory`. A vous de choisir.
- `libre` pour l'adresse ou décalage pour le début de la liste de tranches si la mémoire libre est gérée par la liste de tranches. Dans l'implémentation avec le tableau de tranches on pourra remplacer ce champ par le pointeur vers ce tableau.

Vous ajouterez dans la structure tous ce qui vous sera utile : la taille de la mémoire pointée par `memory`, le nombre de blocs libres, le nombre d'éléments de la liste etc. . .
Ensuite, la fonction `ld_create` alloue la mémoire pour la liste et l'initialise éventuellement (cela dépend de la gestion de tranches libres). L'argument `nb_octets` de `ld_create` spécifie la taille de la mémoire, mais il faut arrondir cette valeur vers un multiple de `sizeof(align_data)`. Donc la mémoire réellement allouée sera de
`nb_blocs( nboctets ) * sizeof(align_data)` octets.
`ld_create` retourne le pointeur vers la structure `head`.

Dans les fonctions suivantes, l'argument `void *liste` désigne toujours un pointeur vers la structure `head` retourné par `ld_create`.

2.2.2 ld_first

```
void *ld_first(void *liste)
```

retourne le pointeur vers le premier nœud de la liste, ou NULL si la liste est vide.

2.2.3 ld_last

```
void *ld_last(void *liste)
```

Idem pour le dernier nœud.

2.2.4 ld_next

```
void *ld_next(void *liste, void *current)
```

`current` est un pointeur qui pointe soit vers `head` (i.e. `liste==current`) soit vers un nœud. Dans le premier cas, `ld_next` retourne la valeur de
`ld_first( liste)`. Dans le deuxième cas elle retourne le pointeur vers le nœud qui suit `current` sur la liste. Si `current` pointe vers le dernier nœud sur la liste, la fonction retourne NULL. Si la liste est vide et `liste==current`, la fonction retourne NULL.

2.2.5 ld_previous

```
void *ld_previous(void *liste, void *current)
```

Idem mais pour le nœud précédent de `current`.

2.2.6 ld_destroy

```
void ld_destroy(void *liste)
```

La fonction détruit la liste en libérant la mémoire de `head` et de `memory`.

2.2.7 void *ld_get

```
size_t ld_get(void *liste, void *current, size_t len, void *val)
```

`current` est un pointeur vers un nœud (donc différent de `liste`). `ld_get` copie `len` octets du champ `data` du nœud `current` vers l'adresse `val`. Si le nombre d'octets demandé est supérieur au nombre d'octets dans le champ `data` du nœud, on limite la copie aux octets qui sont dans `data`. La fonction retourne le nombre d'octets copiés.

2.2.8 ld_insert_first

```
void * ld_insert_first(void *liste, size_t len, void *p_data)
```

insère un nouveau nœud comme le premier élément de la liste.
Les paramètres `len` et `p_data` sont les mêmes que dans la fonction `ld_create_node`.

2.2.9 ld_insert_last

```
void * ld_insert_last(void *liste, size_t len, void *p_data)
```

Idem mais pour insérer le dernier nœud.

2.2.10 ld_insert_before

```
void * ld_insert_before(void *liste, void *n, size_t len, void *p_data)
```

insère un nouveau nœud dans la liste. Les paramètres `len` et `p_data` sont les mêmes que dans la fonction `ld_insert_first`. `n` est un pointeur vers un nœud, `ld_insert_before` insère le nouveau nœud avant le nœud `n`.

2.2.11 ld_insert_after

```
void * ld_insert_after(void *liste, void *n, size_t len, void *p_data)
```

idem pour insérer un nouveau nœud après le nœud pointé par `n`.

Toutes les fonction d'insertion retournent `NULL` si l'insertion est impossible. Elles retournent le pointeur vers le nouveau nœud si l'insertion réussit.

2.3 ld_delete_node

```
void * ld_delete_node(void *liste, void *n)
```

supprime le noeud pointé par `n`. La fonction retourne `liste` quand la suppression réussit et `NULL` sinon.

2.3.1 ld_total_free_memory

```
size_t  ld_total_free_memory(void *liste)
```

retourne le nombre d'octets libres dans la mémoire `memory`. Cela permettra de voir si la mémoire commence à être saturée.

2.3.2 ld_total_useful_memory

```
size_t  ld_total_useful_memory(void *liste)
```

retourne la taille de la mémoire libre qui peut être encore utilisée pour créer de nouveaux nœuds (si on ne réutilise pas la mémoire libérée cela peut être beaucoup moins que ce que retourne la fonction précédente qui compte aussi la mémoire libre mais non utilisable).

2.3.3 ld_add_memory

```
void *ld_add_memory(void *liste, size_t nboctets)
```

agrandit la mémoire de `nb_octets`.
Comme pour la création de la liste, on arrondit `nb_octets` vers un multiple de `sizeof(align_data)`.
Il est impossible de diminuer la taille de mémoire. La fonction retourne NULL en cas de problème et `liste` sinon.

2.3.4 ld_compactify

```
void *ld_compactify(void *liste)
```

compacte la liste en mettant les nœuds au début de la mémoire avec une seule tranche de blocs libres à la fin de la mémoire.
(La fonction devra sans doute allouer une nouvelle mémoire (de la même taille que l'ancienne) pour y recopier tous les nœuds l'un après l'autre dans l'ordre de parcours sur la liste.)
La fonction retourne NULL en cas de problème (échec de `malloc`), sinon elle retourne `liste`.

3 Tester le programme

Vous devez écrire un ou plusieurs programmes qui testent les fonctions *de façon exhaustive*.
Faites les essais avec peu de mémoire pour tester la saturation de la mémoire et les fonctions `ld_add_memory` et `ld_compactify`.
Testez avec des listes homogènes (tous les nœuds de même taille) et des listes dont les nœuds sont de taille variable.
Testez de longues, très longues séquences d'opérations d'insertion et de suppression.
Pour générer automatiquement une longue suite d'opérations vous utiliserez `random`. Prenons trois appels consécutifs à `random` :

```
oper = random();
pos = random();
val = random();
```

Les trois valeurs peuvent être interprétées de la façon suivante :
- si `oper % k != 0`, alors c'est une insertion sinon une suppression [5],
- `pos % longueur_liste` donne la position d'insertion/suppression (il faudra maintenir la longueur de la liste),
- `val` la valeur à insérer (non utilisée pour la suppression).

Avec ce mécanisme on pourra tester vos fonctions sur des suites de centaines de milliers (millions ?) d'opérations et voir le temps d'exécution du programme avec la commande `time`.
Il serait très intéressant de comparer le temps d'exécution de longues suites d'insertions/-suppressions dans votre implémentation avec le temps d'exécution pour une liste chaînée "classique" qui fait `malloc` pour créer chaque nouveau nœud (vous pouvez reprendre le code de cours/TP, c'est le seul moment où on autorise le plagiat).
Y a-t-il un gain de temps quand on utilise votre implémentation ? Ou au contraire ?
La comparaison a un sens si on teste les programmes avec la même suite d'insertions/-suppressions. Mais si on précède la suite d'appels à `random` par `srandom(a)` avec la même valeur de `a` on obtient la même suite de nombres aléatoires.

4 Le débougueur et fuites de mémoire

4.1 Le débougueur gdb

On peut éviter beaucoup de problèmes et accélérer le développement du projet si on apprend les rudiments d'utilisation de débougueur `gdb`. Le minimum nécessaire est décrit sur `https://www.rocq.inria.fr/secret/Anne.Canteaut/COURS_C/gdb.html`. Surtout la section 4 de ce document peut s'avérer très utile : comment trouver les erreurs quand le programme termine prématurément avec `segmentation fault`.
Pour utiliser `gdb` n'oubliez pas de compiler avec l'option `-g`.

4.2 Fuites de mémoire

Vous pouvez [6] vérifier avec `Valgrind`

`http://valgrind.org`.

l'absence de fuites de mémoire. C'est très simple, voyez le tutoriel

`http://valgrind.org/docs/manual/quick-start.html`

5. avec la valeur de k on peut contrôler les fréquences relatives de deux opérations, par exemple si $k == 7$ il y aura 7 fois plus d'insertions que de suppressions

6. et en fait vous devez pour que votre projet soit considéré comme abouti, par la nature du projet les fuites de mémoire diminuent largement l'utilité pratique de votre implémentation