UNIVERSITÉ D'ORLÉANS **L3 Info**
Informatique Année **2017 - 2018**

Série de Travaux Dirigés : **2** *- Les listes doublement chaînées circulaires - TDM NOTÉ*

Le TD est divisé en trois parties. La *première consiste* à mettre en place la structure pour manipuler une liste doublement chaînée circulaire d'entiers. Un programme de test est fourni pour tester les fonctions de manipulation de la liste. La *deuxième partie* a pour objectif de généraliser la liste à des éléments quelque soit leur type. Un programme de test est fourni pour vérifier cette nouvelle implémentation. La *dernière partie* est une application des listes pour modéliser la division cellulaire de l'algue Anabaena Catenula.

Exercice 1. La structure liste doublement chaînée circulaire pour les entiers

À partir de l'archive `TD2.tgz` disponible sur Celene, vous disposez des fichiers suivants :

— `listes_int.h` contenant les en-têtes des fonctions à développer et le typedef du type `liste` sur une structure `liste_struct` cachée définie dans le fichier suivant ;
— `listes_int.c` contenant les *includes* nécessaires, la structure `maillon`, la structure `liste_struct` et les squelettes de toutes les fonctions à développer ;
— `test_listes_int.c` et `test_listes_int_in.txt` pour tester le module `listes_int` ; et
— `Makefile` pour compiler et exécuter les différents tests.

Les fichiers contiennent la documentation et les commentaires nécessaires aux développements demandés.

Les structures `maillon_struct` et `liste_struct` sont définies sur la figure 1. Il faut compléter en conséquence la définition des structures dans le fichier `listes_int.c`.

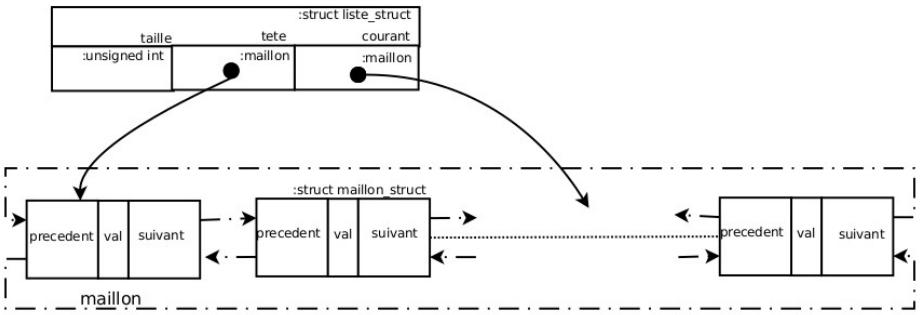

FIGURE 1 – Illustration de la liste doublement chaînée circulaire

Ainsi, un `maillon` permet de contenir une valeur entière et grâce à `suivant` et `precedent` d'accéder aux éléments voisins.

La structure `liste_struct` identifie ce qui est pour elle le début de la liste grâce au pointeur `tete`. Elle possède en outre un champ `taille` qui permet de connaître le nombre d'éléments de la liste (sans calcul) ainsi qu'un pointeur `courant` permettant le parcours de la liste.

Attention, les fonctions d'ajout ou de suppression sur `tete` modifient le début de la liste. Par exemple si la liste vaut `[2 3 5 6]` initialement, `tete` pointe 2. L'application de la fonction `liste_insertion_debut` sur cette liste avec l'entier -1 renvoie la liste `[-1 2 3 5 6]`. *Dessiner dans le compte rendu la structure représentant la dernière liste.*

L'application des fonctions `liste_insertion_avant` ou `liste_insertion_apres` permettent d'utiliser le pointeur `courant` pour insérer un élément au milieu de la liste. Si `courant` est `tete` `liste_insertion_avant` agit de la même manière que `liste_insertion_debut`.

Le programme `test_listes_int` permet de tester les fonctions d'ajout et de suppression d'éléments. Il contient également un parcours de la liste à partir de `courant`. Le `Makefile` doit être utilisé de la manière suivante :

— `make listes_int` permet de compiler et engendrer l'exécutable du programme `test_listes_int.c` fourni ;
— `make test_listes_int` exécute le test et compare avec le résultat à obtenir ; et
— `make memoire_listes_int` exécute le test avec valgrind et vérifie si la mémoire est correctement gérée.

Exercice 2. La liste doublement chaînée circulaire générique

Dans cette deuxième partie, vous devez rendre générique la liste précédente. Dorénavant les éléments seront de type `void*` :

On trouve dans le .h :

```
typedef struct liste_struct* liste;
```

On trouve dans le .c :

```
typedef struct maillon_struct * maillon;
struct maillon_struct {
  void* val;
  maillon suivant;
  maillon precedent;
};

struct liste_struct {
  unsigned int taille;
  maillon tete;
  maillon courant;
  void ( *copier ) ( void * val , void ** pt );
  void ( *afficher ) ( FILE * f , void * val );
  void ( *detruire ) ( void ** pt);
};
```

Il faut ajouter les fonctions nécessaires selon le type des éléments et qui permettent de les manipuler :

1. `copier` pour remplir le champ `val` de `maillon` par allocation de mémoire et recopie (ou autre),
2. `afficher` pour afficher le champ `val`, et
3. `detruire` pour détruire la valeur et désallouer l'espace mémoire (ou autre).

La création de la liste dépend désormais de ces 3 fonctions.

Toujours à partir de l'archive `TD2.tgz`, vous trouverez les fichiers :

— `listes_generiques.h` qui contient les en-têtes des fonctions de manipulation de la liste et

— `listes_generiques.c` qui contient le squelette des fonctions à implémenter sachant que la structure `maillon` et les fonctions correspondantes ne sont visibles que de `listes_generiques.c`.

On retrouve donc les mêmes fonctions à implémenter sachant que désormais `val` est de type `void*`. Certaines sont inchangées par rapport aux listes d'entiers. Par contre les fonctions de la structure `maillon` prennent en paramètre un pointeur sur une des trois fonctions `copier`, `afficher` et `detruire` en fonction de leur nature. Par exemple,

```
static void maillon_ajouter_avant(maillon m, int val)
```

devient

```
static void maillon_ajouter_avant(maillon m, void * val, void (*copier)(void* val, void** pt))
```

afin de construire la valeur `val` du maillon qu'on ajoute en utilisant la bonne fonction de copie.

Vous devez mettre en place toute la *documentation* du module `listes_generiques` et (ré)implémenter les fonctions de `listes_generiques.c` à partir des nouveaux profils.

Le programme `test_listes_generiques.c` est le programme principal qui vous permet de tester votre liste générique sur des entiers. Les commandes associées sont :

1. `make listes_generiques` compile et
2. `make test_listes_generiques` exécute et compare les résultats engendrés avec ceux attendus.
3. `make memoire_listes_generiques` exécute le test avec valgrind et vérifie si la mémoire est correctement gérée.

Dessiner dans le compte-rendu les structures représentant la liste avec les chaînes de caractères "abc" puis "" lorsque les chaînes sont représentées par `chaine` du TD n°1. *Précisez dans le compte-rendu* ce que sont alors les trois fonctions à fournir pour gérer la généricité.

Exercice 3. L'algue Anabaena Catenula

Le principe de la division cellulaire de l'algue Anabaena Catenula est le suivant. Toutes les cellules sont définies par une taille comprises entre 4 et 9 et une orientation à gauche ou à droite. Initialement, on part d'une cellule de taille 4 et orientée à gauche. Ensuite à chaque itération, soit la taille est incrémentée de un si la cellule a une taille inférieure stricte à 9 soit la cellule est divisée. La division consiste à découper la cellule en deux nouvelles cellules. Si la cellule est orientée à gauche, elle engendre une nouvelle cellule orientée à gauche de taille 4 suivie d'une nouvelle cellule orientée à droite de taille 5. Si la cellule est orientée à droite, elle engendre une nouvelle cellule orientée à gauche de taille 5 suivie d'une nouvelle cellule orientée à droite de taille 4.

L'objectif de cette partie est de représenter l'algue et son évolution à l'aide d'une liste doublement chaînée circulaire.

Dans le dossier `TD2` le fichier `algues.c` contient la structure de données `algue`, le profil des fonctions à compléter et le programme principal permettant de tester vos implémentations. Vous n'avez pas à modifier le programme principal mais il reste à définir :

1. `void copier_algue(void* val, void** pt)`, `void afficher_algue(FILE* f, void* val)` et `void detruire_algue(void **pt)` pour pouvoir créer une liste doublement chaînée circulaire de cellules de l'algue Anabaena Catenula,
2. `liste* algue_liste_init(void(*_copier)(void* val, void** pt), void(*_afficher)(FILE* f, void* val), void(*_detruire)(void **pt))` pour créer une liste initialisée avec une première cellule de taille 4 et orientée à gauche, et
3. `void algue_iteration(liste* l)` qui réalise un parcours de la liste afin de faire évoluer chaque cellule selon la règle décrite ci-dessus.

Les commandes disponibles à partir du Makefile sont

1. `make algues` pour compiler
2. `make test_algues` pour exécuter et comparer les résultats engendrés avec ceux attendus.
3. `make memoire_algues` exécute le test avec valgrind et vérifie si la mémoire est correctement gérée.

Compte-rendu

Dans un fichier `cr.pdf` :
— donner la composition du binôme,
— écrire l'état d'achèvement du TP et commenter les structures de données (auriez vous choisi de faire autrement ...).
— répondre aux questions données dans le sujet
— conclure par un bilan plus personnel.

Rendu

La commande

`make rendu`

engendre un fichier archive `rendu.tgz` qui contient les `*.c`, `*.h`, `cr.pdf` et `[Mm]akefile` de votre répertoire.

Ce fichier qui doit être remonté sous Celene, au plus tard 15 minutes après la fin du TP.

Aucun autre format ni nom de fichier n'est accepté pour la remise.