Programmation C
TP n° 6 : Listes doublement chaînées

Rappelons qu'une *liste doublement chaînée* est la donnée d'une suite de triplets – ou *nœuds* – chacun encapsulant :

- une valeur (ici, entière)
- un lien vers le nœud précédent – ou une valeur singulière de lien indiquant que ce nœud est le premier de la suite ;
- un lien vers le nœud suivant – ou une valeur singulière de lien indiquant que ce nœud est le dernier de la suite.

Ces triplets seront représentés en C à l'aide du type de structure suivant :

```c
typedef struct node {
  int val;
  struct node *prev;
  struct node *next;
} node;
```

Le champ `val` est bien sûr la valeur encapsulée. La notion de "lien vers" est implémentée via celle de "pointeur vers" (champs `prev` et `next`). Par convention, la "valeur de singulière de lien" indiquant l'absence de prédécesseur ou de successeur d'un nœud est la valeur nulle de pointeur, `NULL`.

La liste chaînée proprement dite sera représentée sous la forme d'un pointeur vers une structure encapsulant un couple de pointeurs :

- un pointeur vers le premier nœud de la liste – ou `NULL` si la liste est vide ;
- un pointeur vers le dernier nœud de la liste – ou `NULL` si la liste est vide.

Cette structure sera du type suivant :

```c
typedef struct {
  node *first;
  node *last;
} dl_list;
```

Conventions. Dans la suite, ce que nous appellerons *élément* de liste sera toujours un pointeur vers `node`. Lorsqu'un élément n'est pas `NULL`, son *prédecesseur* et son *successeur* sont respectivement les valeurs des champs `prev` et `next` de la structure pointée.

Ce que nous appellerons *liste* sera toujours un pointeur vers `dl_list`, pointant vers une structure `dl_list` dont les deux champs `first` et `last` seront appelés *premier* et *dernier* élément de la liste. Noter que ces deux champs valent tous les deux `NULL` si la liste est vide, et qu'ils sont égaux pour une liste singleton.

Exercice 1 : Préliminaires

Écrire `void print_dl(dl_list *list)`, affichant la suite des valeurs d'une liste en les séparant par des espaces simples, puis effectuant un retour à la ligne.

Ecrire également `void print_dl_rev(dl_list *list)` affichant cette suite à rebours (en commençant par le dernier élément de la liste, et en suivant les liens `prev`).

Nous vous invitons à tester le résultat ou l'effet de *chaque* fonction ci-dessous, en vous servant de celles-ci si nécessaire – si `print_dl_rev` n'affiche pas la suite inverse de `print_dl`, c'est qu'il y a un vrai problème.

Exercice 2 : Allocation et libération

Dans cet exercice comme dans les suivants, tout `malloc` devra être suivi d'un `assert` vérifiant le succès de l'allocation.

1. Écrire une fonction `dl_list *dl_alloc()` allouant une nouvelle liste vide.
2. Écrire une fonction `void dl_free(dl_list *list)` libérant la totalité de la mémoire allouée pour une liste : celle de ces éléments, ainsi que la structure allouée pour la liste elle-même.
3. Écrire une fonction `node *elmt_alloc(int val)` allouant un nouvel élément de liste dont la structure associée encapsulera la valeur spécifiée.
4. Écrire une fonction `void elmt_free(node *elmt)`. Si `elmt` vaut `NULL`, cette fonction ne fait rien ; sinon elle libère la mémoire allouée pour l'élément.

Exercice 3 : Fonction principale de cablage

1. Ecrire la fonction suivante :

```
1 void link_nodes(dl_list *list, node *elmt_l, node *elmt_r);
```

   Cette fonction effectuera les deux traitements suivants :

   1. Si `elmt_l` est différent de `NULL`, alors `elmt_l -> next` prend la valeur `elmt_r` ; sinon `list -> first` prend la valeur `elmt_r`.
   2. Si `elmt_r` est différent de `NULL`, alors `elmt_r -> prev` prend la valeur `elmt_l` ; sinon `list -> last` prend la valeur `elmt_l`.

2. Vérifier à la main, en vous aidant de diagrammes si nécessaire, que si `elmt_l`, `elmt_r` sont tous les deux non nuls et éléments successifs de `list`, et si `elmt` est non nul, alors :

```
1 link_nodes(list, elmt_l, elmt);
2 link_nodes(list, elmt, elmt_r);
```

   insère `elmt` dans la liste, entre `elmt_l` et `elmt_r`.

3. De même, vérifier que si `elmt` est non nul, alors

```
1 link_nodes(list, elmt, list -> first);
2 link_nodes(list, NULL, elmt);
```

   ajoute toujours `elmt` en tête de la liste en mettant à jour tous les champs nécessaires, que la liste soit vide ou non – si elle est vide, `elmt` devient son unique élément.

4. De même, vérifier que

```
1 link_nodes(list, list -> last, elmt);
2 link_nodes(list, elmt, NULL);
```

   ajoute toujours `elmt` en fin de liste en mettant à jour tous les champs nécessaires, que la liste soit vide ou non.

Exercice 4 : Primitives d'ajout

1. Écrire la fonction suivante :

```
node *after(dl_list *list, node *elmt);
```

Si `elmt` vaut `NULL`, la fonction doit renvoyer le premier élément de la liste ; sinon, elle doit renvoyer le successeur de `elmt`.

2. En combinant `elmt_alloc`, `after` et deux appels de `link_nodes`, écrire :

```
void insert_after(dl_list *list, node *elmt, int val);
```

Cette fonction doit insérer dans `list` un nouvel élément encapsulant la valeur `val`. Si `elmt` vaut `NULL`, le nouvel élément est ajouté en début de liste, sinon il est inséré immediatement après `elemt`. Les liaisons entre les éléments, ainsi que les premier et dernier éléments de la liste devront, si nécessaire, être mis à jour de manière appropriée.

Remarque. Les fonctions `after` et `link_nodes` permettent d'écrire cette fonction sans aucun `if ... else` dans le corps de celle-ci. N'oubliez pas de la tester.

3. Symétriquement, écrire :

```
node *before(dl_list *list, node *elmt);
```

Si `elmt` vaut `NULL`, la fonction doit renvoyer le dernier élément de la liste ; sinon, elle doit renvoyer le prédecesseur de `elmt`.

Tester à nouveau votre fonction en reprenant le fragment de code ci-dessus : après la dernière liaison, `before(list, elmt_0)` devrait renvoyer `elm_1`, et `before(list, elmt_1)` devrait renvoyer `NULL`.

4. Symétriquement encore, en combinant `elmt_alloc`, `before` et deux appels de `link_nodes`, écrire :

```
void insert_before(dl_list *list, node *elmt, int val);
```

Cette fonction doit insérer dans `list` un nouvel élément encapsulant la valeur `val`. Si `elmt` vaut `NULL`, le nouvel élément est ajouté en fin de liste sinon il est inséré immediatement avant `elemt` (mêmes remarques que pour `insert_after` sur les mises à jour et l'écriture).

Exercice 5 : Primitives de suppression

1. En combinant `emlt_free`, deux appels de `after` et un seul appel de `link_nodes`, écrire la fonction suivante (toujours sans aucun `if ... else` dans le corps de celle-ci) :

```
void delete_after(dl_list *list, node *elmt);
```

Si `elmt` est différent de `NULL` et n'est pas le dernier élément de la liste, cette fonction doit retirer de la liste son successeur, en libérant son espace-mémoire.

Si `elmt` vaut `NULL`, la fonction doit supprimer le premier élément de la liste si celle-ci est non vide.

Dans tous les autres cas, cette fonction ne fait rien.

2. Symétriquement, en combinant `emlt_free`, deux appels de `before` et un seul appel de `link_nodes`, écrire la fonction suivante (toujours sans aucun `if ... else` dans le corps de celle-ci) :

```c
void delete_before(dl_list *list, node *elmt);
```

Si `elmt` est différent de `NULL` et n'est pas le premier élément de la liste, cette fonction doit retirer de la liste son prédécesseur, en libérant son espace-mémoire.

Si `elmt` vaut `NULL`, la fonction doit supprimer le dernier élément de la liste si celle-ci est non vide.

Dans tous les autres cas, cette fonction ne fait rien.