TP n°5 - Listes chaînées, doublement chaînées

Notions abordées

- type de données liste chaînée, opérations associées, complexités
- manipulation de structures, des pointeurs vers des structures
- découverte de l'opérateur `->` (`p->champ` équivaut à `(*p).champ`)
- découverte de `typedef`
- découverte de `#define`

Toutes les fonctions doivent être commentées et testées. Elles doivent notamment être munies d'une description faisant suite à d'éventuelles hypothèses sur leurs arguments.

Listes chaînées

Exercice 1 Une nouvelle structure de données

• *Qu'est-ce que c'est ?*

La **liste** est une structure de données qui permet de stocker une collection dynamique d'éléments de manière ordonnée. Cela signifie que le nombre d'éléments à stocker n'est pas fixé initialement, à l'inverse des tableaux qui ont une taille fixée. En effet, la taille des tableau statique est fixée dès la compilation, et la taille des tableaux "dynamique" est fixée fixée par l'instruction `malloc`, à l'exécution, mais fixée quand même. Dans une liste, on doit pouvoir :

- ajouter un élément en fin de liste
- enlever un élément en fin de liste
- parcourir les éléments dans l'ordre

La **liste chaînée** est une implémentation de la structure de donnée abstraite décrite ci-dessus, reposant sur l'utilisation de pointeurs. Autrement dit, c'est un moyen concret, ici en C, de gérer une collection dynamique d'éléments ordonnés de sorte que les opérations de parcours, d'ajout et de suppression en fin soient efficaces.

• *Comment on fait ?*

L'idée de la liste chaînée est à l'opposée de celle du tableau : des éléments consécutifs dans la liste peuvent être stockés n'importe où en mémoire, pas du tout de manière consécutive. Ceci permet de faire grossir la structure sans problème : tant qu'il y a de la place quelque part en mémoire, on peut y stocker un nouvel élément, et ce sans avoir à déplacer les éléments pré-existants puisqu'on n'a aucune contrainte de proximité. En revanche, il faut quand même être capable de passer d'un élément au suivant. Pour cela, **on stocke avec chaque élément l'adresse de l'élément suivant**.

Formellement, on crée une structure de **cellule** qui comprend une valeur, et un pointeur vers la cellule suivante. Pour la cellule de fin de liste, la valeur de ce pointeur est `NULL`. Pour passer une liste chaînée en argument d'une fonction, il suffit alors de lui passer l'adresse de la première cellule. Ce qu'on appellera une liste chaînée est donc un pointeur vers une cellule.

Question 1 ✍
Représenter par un schéma une liste chaînée `l1` contenant les éléments 1,3,2,1,2,5. Pour cela chaque cellule sera représentée par deux carrés empilés, représentant respectivement les deux champs de la cellule. On inscrit donc la valeur contenue dans une cellule dans le carré du haut, et le pointeur vers la cellule suivante est représenté par une flèche qui part du carré du bas.
Représenter la variable `l1`, et ajouter un pointeur `l2` vers la liste des éléments de `l1` à partir du deuxième.

- *Type paramétré grâce à* ***`#define`*** *?*

Si l'on souhaite manipuler des listes d'entiers, on va déclarer la structure de cellule comme suit.

```
struct s_cellule{
  int val; //valeur de l'élément
  struct s_cellule * next; //adresse de la cellule suivante
  //ou NULL si c'est la dernière cellule de la liste
};
```

Mais si l'on souhaite manipuler des listes de caractères par exemple, on va plutôt déclarer la structure comme suit.

```
struct s_cellule{
  char val; //valeur de l'élément
  struct s_cellule * next; //adresse de la cellule suivante
  //ou NULL si c'est la dernière cellule de la liste
};
```

Et on pourrait vouloir aussi gérer des listes chaînées de flottants, de booléens, de couple d'entiers, de pointeurs... On se rend compte que le code pour ces différents types de listes est relativement indépendant du type des éléments, *i.e.* du type des valeurs des cellules. C'est pourquoi on utilisera ici une nouvelle directive au préprocesseur : *`#define motA motB`*, qui a pour effet de remplacer dans le code source toutes les occurrences de `motA` par `motB`. Comme *`#include`*, cette directive induit une opération purement textuelle, qui a lieu avant la compilation du code. Ainsi on ajoute en début de fichier la directive suivante (après les *`#include`*).

```
#define type_elem int
```

et on utilisera alors dans toute la suite `type_elem` comme type pour les éléments. La définition de la structure cellule devient alors

```
struct s_cellule{
  type_elem val; //valeur de l'élément
  struct s_cellule * next; //adresse de la cellule suivante
  //ou NULL si c'est la dernière cellule de la liste
};
```

Attention, cela ne veut pas dire qu'on a défini un nouveau type, le compilateur lira `int` et pas `type_elem`, mais le jour où l'on voudra faire des listes de booléens, il suffira de modifier la directive comme suit et de recompiler.

```c
#define type_elem bool
```

Question 2 ✍

Revenons à votre schéma. Maintenant qu'on a défini la structure de cellule et précisé le nom de ces deux champs, vous pouvez repérer sur le schéma précédent ce qui représente `(*l1)`, `(*l1).val`, `(*l1).next`

- *Définir un nouveau type avec* ***typedef***

Afin de ne pas avoir à taper `struct s_cellule` à chaque fois que l'on veut parler d'une cellule, on peut définir un nouveau type appelé `cellule` comme suit.

```c
typedef struct s_cellule cellule;
```

De même, pour ne pas avoir à taper `cellule*` à chaque fois qu'on veut introduire une liste, on définit le type `liste_c` (pour liste **c**haînée) comme suit.

```c
typedef cellule* liste_c;
```

Notez que la définition de ce type `liste_c` ne vous empêche pas d'utiliser le type `cellule *` lorsque vous trouvez ça plus parlant. L'étoile permet notamment de ce rappeler qu'il ne s'agit que d'une adresse, et que si on veut accéder à la valeur, il faut déréférencer avec l'opérateur *. Par exemple, si `p` est de type `cellule*`, alors on accède à la valeur de cette cellule par `(*p).val` ou `p->val`, et à l'adresse de la cellule suivante par `(*p).next` ou `p->next`.

Question 3 💻

Dans un fichier `liste_c.c`, déclarez les types `cellule` et `liste_c` comme proposé ci-avant. Toutes les fonction mentionnées dans cette **partie** devront être codées dans ce fichier. Avant de poursuivre, compilez votre fichier. *Si vous compilez comme d'habitude pour créer un exécutable, il faut penser à écrire une fonction* ***main****, même si elle ne comporte que l'instruction* ***return 0;****.*

Question 4 💻

Codez une fonction `create_cell` qui retourne l'adresse d'une cellule créée de manière persistante, dont le champ `val` a été initialisé selon la valeur passée en argument, et le champ `next` initialisé à `NULL`.

Question 5 💻

Pour tester votre fonction `create_cell`, créez dans le `main` une liste contenant les entiers 5, 6, 7, et 657 dans cet ordre. **Pensez impérativement** à libérer (avec `free`) l'espace réservé par un `malloc` dans les appels à `create_cell`. *Pour l'instant créez les cellules une par une explicitement, sans boucle while, avant de les lier deux à deux, là aussi sans boucle while.*

Question 6 💻

Pour que le test soit plus concluant, on aimerait afficher les éléments de la liste. Codez une fonction `affiche_liste` qui affiche un à un les éléments d'une liste. Afin que cette fonction soit indépendante du type des éléments, on relègue à une fonction `affiche_elem` le soin d'afficher un élément (*i.e.* cette fonction prend en argument un élément et l'affiche, c'est tout). Cette fonction est la seule qu'il faut modifier lorsque `type_elem` est modifié. Afin de ne pas l'oublier, je conseille de laisser en commentaire un petit avertissement sous le ***#define***.

Exercice 2 Création et suppression de listes chaînées

Question 1 💻
Codez une fonction `create_from_tab` qui crée une liste chaînée persistante contenant exactement les mêmes éléments (dans le même ordre) que le tableau qu'elle prend en argument, et qui retourne cette liste, c'est-à-dire qui retourne le pointeur vers la première cellule de la liste.
Attendez avant de tester votre fonction, au risque de réserver de l'espace mémoire sans le libérer ensuite.

Question 2 ✍
Essayez sur un schéma de voir quel algorithme permettrait de libérer une à une toutes les cellules d'une liste. Attention si vous supprimez une cellule sans enregistrer l'adresse de la cellule suivante, il n'y a plus aucun moyen de retrouver la suite de la liste... En cas de doute m'appeler pour me proposer votre algorithme schéma à l'appui.

Question 3 💻
Codez une fonction `free_liste_c` qui prend en argument une liste chaînée et qui libère tout l'espace occupé par ses cellules.

Question 4 💻
Testez vos fonctions dans le `main`.

Exercice 3 Informations tirées de listes chaînées

Question 1 💻
Codez une fonction `nb_elem` qui prend en argument une liste chaînée et qui retourne le nombre d'éléments de cette liste.

Question 2 💻
Codez une fonction `contient_elem` qui prend en argument une liste chaînée et une valeur de type `type_elem`, et qui teste si l'un des éléments de la liste a cette valeur.

Question 3 💻
Codez une fonction `sont_egales` qui prend en argument deux listes chaînées, et qui renvoie si leurs éléments sont exactement les mêmes et dans le même ordre. Comme lorsqu'on compare des chaînes de caractères, il ne s'agit pas de tester si les deux adresses sont les mêmes !

Exercice 4 Modification de listes chaînées

Question 1 💻
Codez une fonction `ajoute_elem_fin` qui prend en argument une liste chaînée et une valeur de type `type_elem`, et qui ajoute cet élément à la fin de la liste.

Question 2 💻
Codez une fonction `ajoute_elem_debut` qui prend en argument une liste chaînée et une valeur de type `type_elem`, et qui ajoute cet élément au début de la liste. On note que cela change nécessairement la première cellule, donc la valeur de la liste, qui est l'adresse de la première cellule, doit être modifiée.

Pour cela, on veillera à passer la liste par référence à cette fonction, et non pas par valeur.

Question 3 💻 (bonus dans un premier temps)
Codez une fonction `supprime_occ` qui prend en argument une liste chaînée et une valeur de type `type_elem`, et qui supprime de la liste toutes les occurrences de cette valeur. On note que la première cellule de la liste est alors sujette à suppression, et que la valeur de la liste, qui est l'adresse de la première cellule, est donc sujette à modification, et pourrait même se retrouver à `NULL`. On veillera donc à passer la liste par référence, et à avertir l'utilisateur sur le risque de mettre à `NULL` la liste. De plus, les cellules supprimées de la liste devront être libérées en mémoire.

Question 4 ✍
Quelle est la complexité de l'opération d'ajout en début de liste avec `ajoute_elem_debut` en fonction de la taille de la liste ?
Et celle de l'ajout en fin de liste avec `ajoute_elem_fin` ?
Voyez-vous un moyen de réduire cette complexité, quitte à améliorer la structure utilisée. Préparez des schémas pour expliquer votre idée puis appelez-moi pour en discuter.

Listes doublement chaînées

Exercice 5 Structure, constructeur et libérateur

Les listes doublement chaînées sont un autre moyen d'implémenter les listes. L'idée ressemble à celle des listes chaînées, avec un stockage modulaire des éléments dans des cellules, mais cette fois chaque valeur est stockée avec à la fois l'adresse de la cellule suivante et celle de la cellule précédente. Une cellule est donc constituée d'une valeur et de deux pointeurs. De plus une liste doublement chaînée n'est pas seulement un pointeur vers la première cellule, mais un couple de deux pointeurs : l'un vers la première cellule, l'autre vers la dernière cellule.

Question 1 💻
Dans un nouveau fichier, déclarer les types `cellule_d` et `liste_d` permettant d'implémenter des listes doublement chaînées. Comme pour les listes simplement chaînées, le type des éléments sera donné par un paramètre fixé par une directive `#define`.

Question 2 💻
Codez des fonctions `create_cell_d`, `create_from_tab`, `free_liste_d` adaptées aux listes doublement chaînées.

Question 3 ✍ puis 💻
Que dire des fonctions permettant l'affichage des listes simplement chaînées ? Comment les modifier pour qu'elles fonctionnent pour des listes doublement chaînées.

Exercice 6 Ajout et suppression en début et fin de liste

Question 1 💻
Codez les fonctions suivantes pour les listes doublement chaînées.

- `ajoute_elem_debut`
- `ajoute_elem_fin`
- `supprime_elem_debut`
- `supprime_elem_fin`

Question 2 ✍
Donnez les complexités des fonctions `ajoute_elem_debut`, `ajoute_elem_fin`, `supprime_elem_debut` et `supprime_elem_fin`. Comparez-les aux complexités de fonctions analogues sur les listes simplement chaînées.

Exercice 7 Ajout et suppression quelconques

Vu qu'on dispose à la fois de l'adresse de la cellule suivante et de celle de la précédente, on peut en fait supprimer ou insérer un élément en milieu de liste. Dans la suite, on appellera cellule intermédiaire une cellule qui n'est ni la première ni la dernière de la liste.

Question 1 ✍
En supposant que `pc` pointe vers une cellule de type `cellule_d`, donner une expression booléenne en C qui vaut `true` si et seulement si la cellule pointée est une cellule intermédiaire.

Question 2 🖳
Donner le code d'une fonction `supprime_cell` qui supprime en temps constant une cellule passée en argument, en supposant qu'il s'agit d'une cellule intermédiaire dans une liste.

Question 3 🖳
Donner le code d'une fonction `ajoute_elem_apres_cell` qui prend en argument une cellule intermédiaire dans une liste, et une valeur de type `type_elem` et qui ajoute entre cette cellule et la suivante une nouvelle cellule contenant la valeur passée en argument, et ce en temps constant.

Question 4 🖳
Afin de pouvoir simplifier les ajouts et suppressions à une position donnée, donnez le code d'une fonction `cell_from_pos`, qui à partir d'une liste et d'un entier $k \in [1..n]$ (où n est le nombre d'éléments de la liste), renvoie l'adresse de la k-ième cellule de la liste.

Question 5 🖳
En faisant appel aux fonctions précédentes, proposez une fonction `ajoute_elem_pos` (resp. une fonction `supprime_pos`), qui ajoute un élément à une position donnée dans la liste (resp. qui supprime l'élément à une position donnée dans la liste).