CRO TD6 : listes triées

Manipulation de pointeurs, listes chaînées, allocation dynamique
1 séance, sur machine

Ce TD a pour but de faire un retour sur les listes en C pour se familiariser avec la manipulation des pointeurs associés aux structures. Nous allons étudier l'insertion dans une liste triée, puis ceux qui auront le temps pourront aborder la fusion de liste.

1 Introduction

Nous allons utiliser le type suivant pour les listes chaînées d'entier :

```
struct model_elem {
   int elem ;
   struct model_elem* suivant;
};

typedef struct model_elem ELEMLISTE;

typedef ELEMLISTE *LISTE;
```

QUESTION 1 ► Est ce que tout le monde est bien **sûr** de comprendre la définition du type ci-dessus.

QUESTION 2 ► Nous vous avons préparé une archive contenant un Makefile et une fonction `afficherListe`. téléchargez-là sur Moodle, exécutez les commandes suivantes :

```
make
./liste
```

et parcourez le code pour être bien sûr de comprendre. Il contient deux fonctions qui ne font rien et dont il faudra remplir le corps.

2 Insertion dans une liste triée : deux moyens

Il existe deux moyens de définir une fonction `f` pour modifier une variable : soit on affecte à la variable le résultat de la fonction `f` (et dans ce cas la fonction elle-même ne modifie par la variable e.g. `a=f(a);` ou `x=x+1;`) soit on passe la variable en paramètre *par référence* à la fonction (et dans ce cas, la fonction modifie la variable) : `inc(&x)` avec (par exemple) pour définition de `inc` :

```
void inc(int *param)
{
  *param=*param+1;
}
```

Notons que pour les tableaux, on ne peut pas renvoyer un tableau comme résultat d'un fonction puisqu'un tableau n'est pas une L-value. On est donc obligé, pour le modifier, de le passer en paramètre d'une fonction. Vous avez déjà fait ça dans le TP2 par exemple (passer un tableau en paramètre et le modifier), et vous savez qu'on n'a pas besoin d'ajouter l'opérateur '&' devant le tableau passé en paramètre. Effectivement, l'opérateur adresse ('&') est implicite pour un tableau : le nom du tableau indique l'adresse du premier élément.
Pour les listes par contre, on peut faire les deux, puisque les listes sont des pointeurs (qui sont des L-values). On peut modifier une liste soit en la renvoyant comme résultat d'une fonction, soit en passant en paramètre l'*adresse de la liste* (la liste étant elle-même un pointeur vers un `ELEMLIST`, on passe donc *un pointeur vers un pointeur vers un* `ELEMLIST`)

QUESTION 3 ▶ Écrire le code C d'une fonction **récursive** [1] qui insère un nouvel entier dans la liste triée. Cette fonction fait l'allocation d'un nouvel élément et l'insère à sa place dans la liste triée (du plus petit au plus grand). On utilisera le prototype suivant pour cette fonction :
`LISTE insertionListeTriee(LISTE liste1,int val)`
Tester votre fonction avec une liste simple

QUESTION 4 ▶ Écrire maintenant une autre fonction qui fait la même chose (i.e alloue un nouvel élément et l'insère à sa place dans la liste triée) mais qui prend en paramètre *un pointeur vers une liste* et modifie la liste pointée par ce pointeur. On utilisera le prototype suivant :
`void insertionListeTriee2(LISTE *pliste1,int val)`
Notez que j'ai appelé l'argument `pliste1` pour bien me rappeler que c'est un pointeur vers une liste et non pas une liste .... vous avez bien compris la différence ? On essaye autant que possible de nommer les variables avec un nom qui correspond à leurs type (e.g. `LISTE *pliste1` ou bien `LISTE liste1`). Tester votre nouvelle fonction.

QUESTION 5 ▶ Écrire une fonction `viderListe` qui supprime tous les éléments d'une liste et renvoie un entier qui est le nombre d'éléments supprimés (pensez à libérez la mémoire pour les éléments supprimés). Testez votre fonction, utilisez le prototype suivant :
`int viderListe(LISTE *pliste1)`
Si vous le pouvez, testez avec `valgrind` que votre fonction ne fait pas de fuite mémoire.

3 Exercice corrigé : Au fait, vous maîtrisez le `typedef` ?

QUESTION 6 ▶ c'est quoi la différence entre :

- `typedef int (*p1d)[10];`
- `typedef int *p1d[10];`

Pour comprendre le `typedef` :

- le mot clé `typedef` est suivit d'une déclaration qui a la même syntaxe qu'une déclaration de variable, mais c'est une déclaration de **type**.
- On rappelle d'abord que l'opérateur `*` est associatif à **droite** (cf le poly ou google), on `int **a` revient à `int (*(*a));`
- Il faut connaître les priorité des opérateurs. L'opérateur `'[]'` est plus prioritaire que l'étoile dont la deuxième définition revient à :

  `typedef int *p1d[10];` ⇔ `typedef int (*(p1d[10]));`

- Donc pour bien comprendre le type à partir de `typedef int (*(p1d[10]));`, on fait :
  - `(*(p1d[10]))` est de type `int`
  - donc `(p1d[10]))` est de type `pointeur vers un int`
  - l'opérateur `'[]'` indique que ce qui est précède est un tableau plutot qu'un simple scalaire donc
  - `p1d` est de type `tableau (de taille 10) de pointeurs vers int`
- Pour l'autre `typedef int (*p1d)[10];`
  - `(*p1d)[10]` est de type `int`
  - `(*p1d)` est de type `tableau (de taille 10) d'int`
  - `p1d` est de type `pointeur vers un tableau (de taille 10) d'int`

1. Quand on manipule des listes (ou des arbres), façonnés avec des structures et des pointeurs, on utilise quasiment **toujours** des fonctions récursives.

4 Pour ceux qui ont fini : Fusion de listes triées

Écrire une fonction qui fusionne deux listes triées en une seule liste triée. Le prototype sera le suivant :
`void fusion (LISTE *pliste1, LISTE *pliste2)`
À la fin de l'exécution de la fonction, la liste pointée par `pliste1` contiendra tous les éléments (triés) des deux listes et la liste pointée par `pliste2` sera vide.
Comme souvent dans les programme manipulant des pointeurs sur des structures, on n'a pas besoin de recréer les éléments (avec `malloc`) ou de les libérer (avec `free`). Il suffit de réarranger la manière dont sont connectés les éléments entre eux...