Travaux Dirigés Système n°1

Cours d'Informatique de Deuxième Année

— Licence 2ème Année, Deuxième Semestre MIAS —

Listes chaînées génériques

Le but de ce TD est d'écrire des *fonctions C génériques* pour la gestion de listes chaînées. L'intérêt de fonctions génériques est d'avoir des fonctions qui sont adaptées à tous les types de données sans qu'il soit nécessaire de les réécrire ou même de les modifier !

▸ Exercice 1. (Définitions des types)

Tout d'abord, nous allons écrire un mécanisme de gestion de listes chaînées. Le type des éléments contenus dans la liste chaînée sera void. L'utilisation de void * nous permet de faire des listes de « n'importe quoi ». Nous allons utiliser la structure suivante : définie comme suit :*

```
struct cellule {
 void* data;
 struct cellule *suivant;
};
```

Le void data correspond à la donnée inclue dans la cellule courante et suivant correspond à la cellule suivante. On définie maintenant le type Liste de la façon suivante :*

```
typedef struct cellule* Liste;
```

▸ Exercice 2. (Préliminaires : Fonctions utilitaires) *On va écrire maintenant quelques fonctions de gestion de listes chaînées suivantes :*

1. *Liste* creation_liste() ; qui créée une liste vide.*
2. *void liberation_liste(Liste** liberee) ; qui libère la mémoire occupée par la liste*
3. *void ajout_en_tete_de_liste(List* liste, void* element) ; qui ajoute une élément en tête de la liste.*
4. *void *retire_tete_de_liste(Liste **liste) ; qui retire le premier élément de la tête de liste.*

Il est très important de noter que quel que soit le type des éléments de la liste, l'ensemble du code ne change pas ! On peut donc dire que ce code est parfaitement générique. Mais ceci va bientôt changer...

Remarque : *Ecrivez une fonction main permettant de tester l'ensemble des fonctions que vous avez écrites.*

▸ Exercice 3. (Va-t-on perdre la généricité ?) *À partir de maintenant, on suppose qu'on dispose des deux fonctions suivantes :*

1. *int compare_elements(void* e1, void* e2) ; qui renvoie 0 s'ils sont égaux, un nombre positif si l'élément 1 est plus grand que l'élément 2 et un nombre négatif si l'élément 1 est plus petit que l'élément 2.*
2. *void affiche_element(void* element) ; qui affiche sur la sortie standard l'élément correspondant.*

Vous ne devez PAS les écrire pour le moment !

On vous demande maintenant d'écrire les fonctions suivantes en considérant que les deux fonctions ci-dessus existes :

1. *void affiche_liste(Liste* liste) ; qui affiche les éléments de la liste.*
2. *void* recherche_element_liste(Liste* liste, void* element) ; qui recherche l'élément dans la liste. Si cet élément est trouvé la fonction le renvoie sinon elle retourne NULL.*
3. *void liste_ajoute_element(Liste** liste, void* element) ; qui ajoute l'élément uniquement s'il* ***N'est PAS déjà*** *dans la liste !*

IMPORTANT : *A chaque fois qu'il est nécessaire de comparer des éléments, n'utilisez pas simplement le test d'égalite == mais plutôt la fonction compare_elements.*

▸ Exercice 4. (Application) *Nous allons maintenant supposer que les données contenues dans la liste chaînée sont des chaînes de caractères. Écrire les fonctions compare_elements et affiche_element correspondantes, puis vérifier le bon fonctionnement de l'ensemble (dont l'insertion de doublons et vérifiez que lorsque vous appelez la fonction liste_affiche, ils ont été bien insérés qu'une seule fois et donc qu'il n'y a pas de doublons).*

▸ Exercice 5. (Retour de la généricité)

Comme vous l'avez certainement remarqué, l'introduction des deux fontions compare_elements et affiche_element nous a fait perdre le caractère "universel" de nos fonctions !

Pour résoudre ce problème nous allons donc introduire une nouvelle structure contenant ces fonctions. L'idée est la suivante :

```
struct type {
 int (*compare)(void* e1, void* e2);
 void (*affiche)(void* e1);
};

typedef struct type Type;
```

Pour illustrer un peu cette écriture, voici un petit exemple :

```
struct point {
 int x, y;
};

typedef struct point Point;

int compare_point(void* e1, void* e2) {
 Point *p1 = e1;
 Point *p2 = e2;

 return (p1->x == p2->x && p1->y == p2->y);
}

void affiche_point(void* element) {
 Point *point = element;

 printf("x = %d, y = %d\n", point->x, point->y);
}

Type type_point = {
  compare_point,
  affiche_point
};
```

Nous devons aussi modifier la structure cellule pour y inclure cette notion de "type" :

```
struct cellule {
void* tete;
struct cellule* queue;
Type* type;
};
```

Et modifier quelques fonctions. Par exemple la fonction de création aura maintenant le prototype suivant :

```
List* creation_liste(Type* type);
```


D'autre part, les fonctions qui utilisent les fonction de comparaison et d'affichage doivent être modifiées afin d'utiliser les fonctions contenues dans la structure type. Adapter l'exemple sur les noms pour vérifier que tous ces nouveaux concepts sont bien correctement compris et implémentés.

▸ **Exercice 6. (Dictionnaire)** *On demande d'écrire un programme dico qui prend au moins deux arguments au moyen du mécanisme argc, argv en paramètre de la fonction main. Ces paramètres sont un fichier contenant une liste de mots (par exemple /usr/share/dict/french), et un ou plusieurs mots. Le programme doit charger complètement le dictionnaire en mémoire, puis indiquer pour chaque mot s'il existe ou pas dans le dictionnaire.*

Par exemple la ligne d'exécution suivante :

```
dico /usr/share/dict/french abandon kaouète zouave zapoyoko
```

Aura la sortie suivante :

```
abandon appartient au dictionnaire
kaouète n'appartient pas au dictionnaire
zouave appartient au dictionnaire
zapoyoko n'appartient pas au dictionnaire
```

▸ **Exercice 7. (Temps de recherche...)**

Comparer avec la commande time le temps nécessaire pour rechercher un mot, selon qu'il est au début, à la fin, au milieu, etc. du dictionnaire.

```
time dico /usr/share/dict/french mot
```

Ne pourrait-on pas améliorer les temps de recherche en utilisant une autre structure de donnée qu'une liste ?