TRAVAUX PRATIQUES VIII

Listes simplement chainées

L'objectif de ce TP est coder les listes simplement chainées en C.

On rappelle que les options suivantes doivent être utilisées à la compilation afin de détecter plus d'erreurs syntaxiques :

```
-Wall -Wextra
```

On rappelle que l'on peut ajouter les options suivantes pour obtenir de meilleurs messages d'erreurs à l'exécution :

```
-fsanitize=undefined,address
```

Il est fortement recommandé pour ce TP de *faire des dessins !*.

A Rappels et compléments pratiques

A.1 NULL

`NULL` (qui provient de `stdlib`) est le pointeur qui ne pointe sur rien.

A.2 assert

`void assert(bool expr)` est une fonction (qui provient de la librairie nommée... `assert` ; so much surprise) utile pour déboguer. Si l'expression booléenne passée en argument est vraie, rien ne se passe. Si elle est fausse, le programme s'interrompt immédiatement.

Par exemple, pour garantir qu'un pointeur ne vaut pas `NULL` dans une fonction, on peut débuter le code de celle-ci par `assert(ptr != NULL);` .

Définition 1 (Programmation défensive).
Lorsqu'une fonction commence par s'assurer que ses pré-conditions sont vérifiées, on parle de **programmation défensive** : on se *défend* contre un mauvais usage de la fonction.

B Révisions théoriques

Le fonctionnement de ces listes correspond à ce que l'on appelle des **listes simplement chainées**. On veut pouvoir :

- Créer la liste vide (`[]` en OCaml).
- Tester si une liste est vide.
- À partir d'une liste `l` non-vide, accéder au premier élément (en OCaml, filtrer en `x::l` et renvoyer `x`).
- À partir d'une liste `l` non-vide, accéder à la liste composée de tous les éléments sauf le premier (en OCaml, cela correspond à filtrer en `t::q` et renvoyer `q`).
- À partir d'une liste `l` et d'un élément `x`, créer une nouvelle liste qui contient `x` *puis* les éléments de `l` (l'équivalent de `x::l` en OCaml).

On utilisera l'implémentation vue en cours :

- La liste est découpée en cellules (aussi appellées chainons ou maillons).
- Chaque cellule stocke un élément, ainsi qu'un pointeur vers l'élément suivant. Cf Figure VIII.1.

Chaque cellule est allouée en mémoire. **Une liste est alors un pointeur vers une cellule.**

1. Sur la Figure VIII.1, ajouter le pointeur qui correspond à la liste `[12; 99; 37]`

La liste vide correspond au pointeur `NULL`. Sur la Figure, il est représenté comme un pointeur vers une case barrée [1] . Pour parcourir une liste, il faut donc « suivre les flèches » et lire les éléments au fur et à mesure, jusqu'à tomber sur la flèche `NULL` qui marque la fin.

Les opérations décrites précédemment fonctionnent ainsi :

- Pour créer une liste vide, il n'y a rien à faire : une liste vide est le pointeur vers aucun maillon, donc le pointeur `NULL`.
- Pour tester si un pointeur vers un maillon correspond à la liste vide, il faut tester si le pointeur pointe bien vers un maillon, çad tester s'il est non-`NULL`.
- Accéder à la tête ou à la queue de la liste demande de suivre le pointeur vers le maillon de tête, puis de renvoyer soit la valeur stockée dans le maillon soit le pointeur vers la suite.
- Créer une nouvelle liste en ajoutant un élément en tête est l'opération la plus intéressante. Pour cela... on va créer un nouveau maillon, le faire pointer vers l'ancienne liste, et renvoyer l'adresse de ce maillon. En particulier, on ne duplique pas l'ancienne liste !

2. Sur la Figure, ajouter :
   a. Un pointeur correspondant à la lsite `[37]` (i.e. la liste du schéma dont on a passé les deux premiers éléments de tête).
   b. Un pointeur correspondant à la liste `0::[99; 37]` (il faut créer un nouveau maillon).
   c. Un pointeur correspondant à la liste `6::[12; 99; 37]` (idem).
   d. `0::[12; 99; 37]` (idem).
   e. Un pointeur correspondant à la liste `20::6::[12; 99; 37]` (idem).

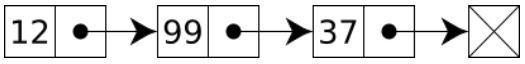

Figure VIII.1 – Des cellules chainées. Source : Wikipédia.

3. Comptenez le nombre total de maillons. Comparez avec la somme des longueurs des différentes listes crées. Commentez.

Cette façon d'implémenter des listes est appélée listes simplement chainée. Telle que décrite ci-dessus, il s'agit d'une structure de donnée fonctionnelle (les transformateurs renvoient une nouvelle structure de donnée au lieu de modifier celle passée en argument).

C Codons

Une cellule sera implémentée par le type suivant :

1. C'est une représentation standard, qui a ses intérêts pour des variantes de la structure de données, mais que je n'aime pas pour les listes simplement chainées telles que nous les faisons. D'ailleurs, je n'ai pas utilisée cette notation là dans le cours ; vous habituer au fait que les schémas diffèrent légèrement d'une source à l'autre est une bonne chose.

list.h

```c
struct cellule_s {
  int elem;
  struct cellule_s* next;
};
typedef struct cellule_s cellule;
```

Une liste sera donc un pointeur vers une cellule, c'est à dire un élément de type `cellule*` .

C.1 Let's a go !

Maintenant, c'est à vous : la question suivante vous laisse en autonomie sur le TP.

4. Implémentez les fonctions présentes dans le header, et testez-les depuis `test.c` . Vous n'êtes pas obligé·es de les implémenter dans l'ordre.

À titre informatif, voici l'ordre dans lequel je pense corriger les fonctions au tableau. C'est plus ou moins un ordre croissant de difficulté.

- `list_create` et `list_hd` et `list_tl`.
- Puis `print_list`.
- Puis `cellule_free` puis `list_free`.
- Puis `list_cons`.
- Puis le reste.

D Pour les plus rapides

5. Déclarez et créez une fonction permettant d'accéder au ième élément d'une liste.
6. Déclarez et créez une fonction permettant de transformer un tableau (et/ou une zone allouée) en liste. De même pour la réciproque (on prendra la taille de la liste en argument).
7. (Difficile) Déclarez et créez une fonction permettant de trier une liste.
   On devra renvoyer une copie de la liste triée, et non trier la liste donnée en argument.
   Indice : Le tri insertion est particulièrement adapté à cette structure de donnée.

Et ensuite ?

8. Avancez sur votre projet info perso, ou sur l'Advent of Code, ou sur Prologin, ou sur un DM d'info.