TD Méthodologie de la Programmation
Feuille d'exercices n°6

Sebastien.Varrette@imag.fr (Bureau BR.4.07)

Le but de ce programme est d'implementer une bibliothèque gérant des *listes chaînées*. On utilisera une programmation modulaire et une compilation utilisant l'utilitaire `Makefile`. Ainsi, les fonctions et les définitions de structures seront ainsi définies dans les fichiers `linked_list.c` et `linked_list.h`. Le fichier `main.c` contiendra la fonction `main()` qui illustrera l'utilisation de la bibliothèque implémentée.

1 Listes chainées d'entiers

Une liste chaînée utilise un cas particulier de structures dites "auto-référées", c'est à dire dont un des membres pointe vers une structure du même type.
il est possible de représenter une liste d'éléments de même type par un tableau (ou un pointeur). Toutefois, cette représentation, dite contiguë, impose que la taille maximale de la liste soit connue a priori (on a besoin du nombre d'éléments du tableau lors de l'allocation dynamique). Pour résoudre ce problème, on utilise une représentation chaînée : l'élément de base de la chaîne est une structure appelée cellule qui contient la valeur d'un élément de la liste et un pointeur sur l'élément suivant. Le dernier élément pointe sur le pointeur `NULL` (défini dans `stddef.h`). La liste est alors définie comme un pointeur sur le premier élément de la chaîne.
Considérons par exemple la structure `toto` possédant 2 champs :

1. un champ `data` de type `int` ;
2. un champ `next` de type pointeur vers une `struct toto`.

La liste est alors un objet de type pointeur sur une `struct toto`. Grâce au mot-clef `typedef`, on peut définir le type `list`, synonyme du type pointeur sur une `struct toto` (en utilisant l'instruction `typedef`). On peut alors définir un objet `list l` qu'il convient d'initialiser à `NULL` (pour symboliser une liste vide). Cette représentation est illustrée dans la figure 1.

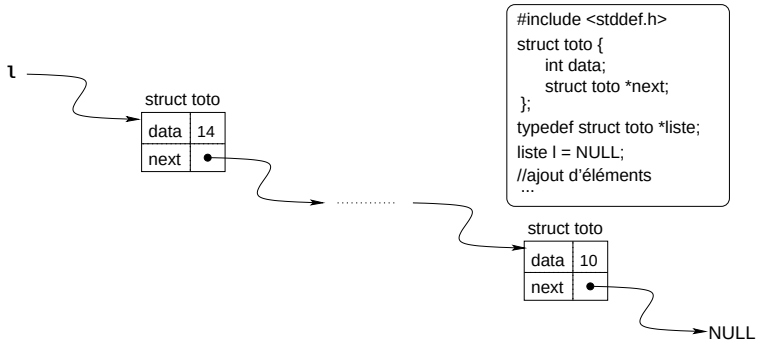

Fig. 1 – Representation d'une liste chaînée

Un des avantages de la représentation chaînée est qu'il est très facile d'insérer un élément à un endroit quelconque de la liste.

1.1 Préliminaires

Définir les structures utilisées et le prototype des fonctions implémentées dans le fichier `linked_list.h` Créer le fichier `linked_list.c` qui contiendra le corps des fonctions de la bibliothèque. Créer de même le fichier `main.c` qui contiendra la fonction `main()` et qui inclut évidemment `linked_list.h` via la directive `#include`. Enfin, écrire le fichier de configuration `Makefile` qui permettra de compiler ce projet.

1.2 Ajout d'éléments

De façon général, l'ajout d'un nouvel élément dans une liste chaînée s'effectue en trois étapes :

1. recherche de l'endroit où la nouvelle cellule devra être insérée ;
2. création de la nouvelle cellule (par `malloc`) et mise à jour de ses champs. Au niveau de l'allocation en mémoire, on prendra garde de réserver le bon nombre d'octets (en reprenant l'exemple précédent, `sizeof(struct toto)` et non `sizeof(struct toto *)`) et de convertir le résultat du `malloc` vers le bon type (pointeur vers la structure).
3. mise à jour des champs des cellules voisines.

Ecrire la fonction `list insertInHead(int elt, list L) ;` qui insère un nouvel élément `elt` en tête de la liste `L` et renvoit la liste résultante.
De même, écrire la fonction `list insertInTail(int elt, list L) ;` qui insère un nouvel élément `elt` en queue de la liste `L` et renvoit la liste résultante.

1.3 Impression

Ecrire la fonction `void printList(liste L) ;` qui affiche à l'écran le contenu de la liste `L`. Voici par exemple le résultat de l'application de cette fonction sur la liste (14,27,11) :

```
14 -> 27 -> 11 -> NULL
```

1.4 Suppression d'éléments

Il est **primordial** d'utiliser `free` dans les listes chaînées. Sinon, la mémoire risque d'être rapidement saturée (par exemple lors d'ajouts et de suppressions successives qui ne libèrent pas l'espace alloué). Ainsi, de façon similaire à l'ajout, la suppression d'une cellule dans une liste chaînée s'effectue en trois étapes :

1. Recherche de la cellule à supprimer ;
2. mise à jour des champs des cellules voisines ;
3. libération de la cellule avec `free`.

Ecrire la fonction `list deleteHead(list L) ;` qui supprime l'élément de tête (s'il existe !) et renvoit la liste résultante. Utiliser cette fonction pour définir la fonction récursive `list deleteList(list L) ;` qui supprime l'intégralité de la liste `L`.
Ecrire la fonction `list deleteTail(list L) ;` qui supprime l'élément en queue (s'il existe !) et renvoit la liste résultante.