Université de Liège

INFO0947

Compléments de Programmation

Un exercice dont vous êtes le Héros · l'Héroïne.

Listes Chainées – Occurrences d'une Valeur

Simon Liénardy Benoit Donnet

19 avril 2020

Préambule

Exercices

Dans ce « TP dont vous êtes le héros · l'héroïne », nous vous proposons de suivre pas à pas la résolution de trois exercices permettant la manipulation des Listes chainées, l'implémentation du TAD List utilisant des pointeurs. [1] L'un deux aborde les listes simplement chainées dans le cadre d'un algorithme itératif. Un autre aborde la même structure de données dans le cadre de la récursion. Enfin, un dernier aborde les listes doublement chainées.

Il est dangereux d'y aller seul [2] !

Partagez vos commentaires, questions, solutions alternatives sur le forum eCampus. N'hésitez jamais !

1. Parce qu'on peut utiliser des tableaux aussi : c'est comme cela que s'est implémenté en Python, par exemple.
2. Référence vidéoludique bien connue des Héros.

6.1 Commençons par un Rappel

Si vous avez déjà lu ce rappel, vous pouvez directement atteindre le point de l'énoncé de l'exercice (Sec. 6.2).

6.1.1 Liste Chainée ?

Une *liste chainée* (ou chaînée [3]) se présente comme une succession de *cellules*, reliée l'une à l'autre à l'aide d'un pointeur.

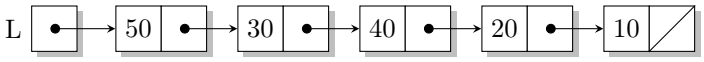

FIGURE 1 – Exemple de liste chainée à 5 cellules.

On représente une liste chainée en dessinant une succession de *cellules*, chacune comprenant deux parties : (*i*) la partie utile (qui sert à stocker la donnée – dans l'exemple de la Fig. 1 des entiers) et, (*ii*) la partie liante (qui permet de pointer vers la cellule suivante dans la liste). On accède à la première cellule de la liste grâce au pointeur de début (`L` dans la Fig. 1). On sait qu'on est sur la dernière cellule quand la partie liante contient la valeur particulière `NULL`

En quoi cela diffère-t-il des tableaux ? Une liste chainée est une structure séquentielle (à l'instar des fichiers), ce qui signifie qu'on ne peut accéder à une cellule qu'après avoir lu les cellules précédentes. On ne peut avancer, avec une liste chainée, que dans une seule direction (i.e., du début de la liste à la fin de la liste).

3. L'orthographe du français a été rectifiée en 1990 (Je –Simon– n'étais pas encore né). Bien que relativement timide, cette réforme simplifie l'orthographe d'un grand nombre de mots. En particulier, beaucoup d'accents circonflexes disparaissent, comme dans le mot « chaine ». Je vous invite à adopter cette écriture.

6.1.2 Notations sur les Listes

Faites en sorte de bien vous **familiariser** avec ces notations ! Si vous ne les comprenez pas, relisez le cours (Chap. 6, Slides 51 → 54).

Notion	**Notation**	**Application à la Fig. 2**
Liste	$L^+ = (e_0, e_1, e_2, e_3, \ldots, e_i, \ldots, e_{n-1})$	$L^+ = (50, 30, 40, 20, 10)$
Élément de rang i	e_i	élément de rang $2 = 40$
Longueur de L	$long(L) = n$	$long(L) = 5$
Sous-liste de L	$p^+ = (e_i, \ldots e_{n-1})$	$p^+ = (40, 20, 10)$
Sous-liste des élem avant L	$p_L^- = (e_0, e_1, e_2, \ldots, e_{i-1})$	$p^- = (50, 30)$
↪ Si pas d'ambigüité	$p_L^- = p^-$	
↪ Par convention	$L^+ = p^- \,\|\|\, p^+$	$L = (50, 30) \,\|\|(40, 20, 10)$
Liste vide	$L^+ = ()$	-
↪ Si $L^+ = (e_0 \ldots e_{n-1})$	$L^- = ()$	-

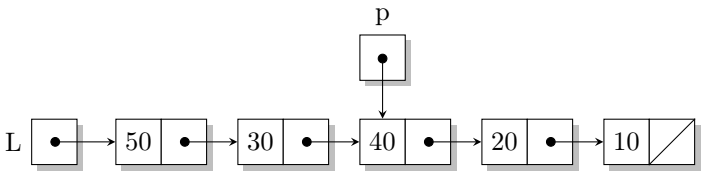

FIGURE 2 – Application des notations.

6.1.3 Rappels sur l'Implémentation

6.1.3.1 Header (`list.h`)

```c
#ifndef __LIST__
#define __LIST__
#include "boolean.h"

typedef struct list_t List;

List *empty_list();

Boolean is_empty(List *L);

int length(List *L);

//autres fonctions (voir Chap. 6, Slides 20-21)
#endif
```

6.1.3.2 Module (`linked_list.c`)

```c
#include <stdlib.h>
#include <assert.h>

#include "list.h"

struct list_t{
  void *data;
  struct list_t *next;
};

typedef struct list_t cell;

//Create a new cell
static cell *create_cell(void *data){
  cell *n_cell = malloc(sizeof(cell));
  if(n_cell==NULL)
    return NULL;

  n_cell->data = data;
  n_cell->next = NULL; // Extrêmement important !

  return n_cell;
}//end create_cell()

//Voir cours pour l'implémentation des autres fonctions/procédures
```

6.2 Énoncé

Spécifiez et construisez (de manière récursive) une fonction qui détermine le nombre d'occurrences d'une valeur donnée ($\in \mathbb{Z}$) dans une liste simplement chainée d'entiers, `L`.

6.2.1 Méthode de résolution

Nous allons suivre l'approche constructive vue au cours. Pour ce faire nous allons :

1. Formuler le problème récursivement (Sec. 6.3) ;
2. Spécifier le problème complètement (Sec. 6.4) ;
3. Construire le programme complètement (Sec. 6.5) ;
4. Rédiger le programme final (Sec. 6.6).

6.3 Formulation récursive

Tout d'abord, il convient de formaliser le problème de manière récursive.

Indice : relisez bien l'énoncé avant de progresser dans la suite !

Si vous voyez directement comment procéder, voyez la suite	6.3.3
Si vous êtes un peu perdu, voyez le rappel sur la récursivité	6.3.1
Si vous pensez avoir bien compris la récursivité mais que vous ne voyez pas comment l'appliquer dans le cas particulier des listes, voyez les conseils	6.3.2

6.3.1 Rappels sur la Récursivité

6.3.1.1 Formulation Récursive

On ne va pas se cacher derrière son petit doigt, c'est la principale étape et la plus difficile. Les étapes suivantes de résolution sont rendues beaucoup plus simples dès que la formulation récursive est connue. Comment procéder ?

En fait, il y a deux choses à fournir pour définir récursivement quelque chose :

1. Un cas de base ;
2. Un cas récursif.

Trouver le **cas de base** n'est pas toujours facile, certes, mais ce n'est pas le plus compliqué. En revanche, cerner le **cas récursif** n'est pas une tâche aisée pour qui débute dans la récursivité. Pourquoi ? Parce que c'est complètement contre-intuitif ! Personne ne pense récursivement de manière innée. C'est une *manière de voir le monde*[4] qui demande un peu d'entrainement et de pratique pour être maitrisée. Heureusement, **vous êtes au bon endroit** pour débuter ensemble.

Toutes les définitions récursives suivent le même schéma :

> Un X, c'est *quelque chose* **combiné** avec un un X *plus simple*.

Il faut détailler :

— Quel est ce *quelque chose* ?
— Ce qu'on entend par **combiné** ?
— Ce que signifie plus simple ?

Si on satisfait à ces exigences, on aura défini X récursivement puisqu'il intervient lui-même dans sa propre définition.

Un exemple mathématique, pour illustrer ce canevas de définition récursive :

> n ! $= n \times$ (n-1) !

On veut définir ici la factorielle de n ($n!$). On dit que c'est n (notre *quelque chose*) combiné par la **multiplication** à une factorielle plus simple : $(n-1)!$

6.3.1.2 Programmatiquement parlant

Par définition, un programme récursif s'appelle lui-même, sur des données *plus simple*. La tâche du programmeur consiste donc à déterminer comment **combiner** le résultat de l'appel récursif sur ces données plus simples avec les paramètres du programme pour obtenir le résultat voulu.

Dans la rédaction d'un programme récursif, on a donc **toujours** accès à un sous-problème particulier : le programme lui-même. Il convient par contre de respecter deux règles (cf. slides Chap. 4, 23 → 26) :

Règle 1 Tout algorithme récursif doit distinguer plusieurs cas, dont l'un au moins ne doit pas comporter d'appel récursif ;

Règle 2 Tout appel récursif doit se faire avec des données plus « proches » de données satisfaisant une condition de terminaison

4. On dit aussi *paradigme*.

Les cas non-récursifs sont appelés **cas de base**.

Les conditions que doivent satisfaire les données dans ces cas de base sont appelées **conditions de terminaison**.

6.3.1.3 Construction récursive d'un programme

Pour construire un programme récursif, il convient de respecter ce schéma. Notez bien qu'ici n'interviendra pas la notion d'invariant tant qu'il n'y a pas de boucle en jeu [5] !

1. Formuler récursivement le Problème ;
2. Spécifier le problème. Le plus souvent, il suffit d'utiliser la formulation récursive préalablement établie ;
3. Construire le programme final en suivant l'approche constructive. La plupart du temps, c'est trivial, il suffit de suivre la formulation récursive.

Et les sous-problème dans tout ça ? Si le problème principal semble trop compliqué, il se peut qu'une découpe en sous-problèmes soit pertinente. Ce sont alors ces sous-problèmes qui seront récursifs et dont le programme pourra être rédigé en suivant les trois points ci-dessus.

De toute façon, chaque programme récursif possède au moins un sous-problème : lui-même ! Par définition de la récursion. Attention que par l'approche constructive, avant d'appeler n'importe quel sous-problème, il convient d'assurer que sa précondition est respectée. Après l'appel, sa postcondition sera respectée par l'approche constructive.

6.3.2 Liste et récursivité – Conseils

Une première constatation utile est que la liste est elle-même définie de manière récursive, en effet,

Une liste, c'est *une cellule* liée à une liste *plus petite*.

Traduit formellement, cela donne :

$$L^+ = (data) \;||\; L \rightarrow next^+$$

La définition récursive d'une opération sur une telle structure récursive suivra très souvent la définition récursive de de cette structure. [6] En d'autres termes, cela veut dire que le cas de base de l'opération que nous allons définir devra s'occuper du cas de base de la liste (i.e., une liste vide est une liste) le cas récursif s'occupera du contenu de la cellule de tête d'une part et de la suite de la liste, d'autre part.

6.3.2.1 Suite de l'exercice

À vous ! Formalisez le problème de manière récursive et passez à la section 6.3.3.

5. Peut-on mêler Invariant et récursion ? Bien sûr : souvenez-vous en lorsqu'on vous parlera de l'algorithme de tri `quicksort`.
6. Spoilers pour la suite de votre cursus : on peut établir des schémas généraux d'algorithmes récursifs définis sur ces structures récursives, on appelle cela de la *récursivité structurelle*.

6.3.3 Mise en commun de la formulation récursive

Sur base de l'indice donné (voir Sec. 6.3.2), nous allons définir le cas de base et le cas récursif de la fonction *Comptage*.

Cas de base Une liste vide est une liste. C'est donc notre cas de base. Combien y a-t-il d'éléments dans une liste vide ? Zéro !

$$Comptage((\,), X) = 0$$

Cas récursif Nous nous basons maintenant la propriété suivante :

$$L^+ = (data) \,||\, L \rightarrow next^+$$

Le nombre d'occurrences de X dans cette liste est donc le nombre d'occurrence de X dans $(data)$ additionné au nombre d'occurrences de X dans $L \rightarrow next^+$.

Il faut donc additionner 0 ou 1 suivant la valeur de *data* au nombre de X dans $L \rightarrow next^+$. Avons-nous une fonction qui nous permettent de représenter facilement le nombre de X dans $L \rightarrow next^+$? Oui : *Comptage* !

$$Comptage(L, X) = ((L \rightarrow data = X) + Comptage(L \rightarrow next, X)$$

Synthèse Au final, la formulation récursive de *Comptage* est la suivante :

$$Comptage(L, X) = \begin{cases} 0 & \text{si } L^+ = (); \\ (L \rightarrow \text{data} = X) + Comptage(L \rightarrow \text{next}, X) & \text{sinon.} \end{cases}$$

6.3.3.1 Suite de l'exercice

Vous commencez à connaitre le fonctionnement de la résolution : il faut maintenant passer aux spécifications. Voir la Sec. 6.4.

6.4 Spécification

Il faut maintenant spécifier le problème.

Pour un rappel sur la spécification, voyez la Section 6.4.1
La correction de la spécification est disponible à la Section 6.4.3

6.4.1 Rappel sur les spécifications

Voici un rappel à propos de ce qui est attendu :

La Précondition C'est une condition sur les données. Pour l'établir, il faut se demander quelles doivent être les données du problème. Ici, dans notre exemple, cela parle de tableau. Quelles conditions devons-nous imposer sur ce tableau ? N'hésitez pas à vous référer au cours INFO0946, Chapitre 6, Slides 54 → 64).

La Postcondition C'est une condition sur le résultat du problème, si tant est que la Précondition est remplie (si tel n'est pas le cas, on pratique la *programmation défensive* (voir INFO0946, Chapitre 6, Slides 65 → 70) et on arrête l'exécution du programme.

La signature est souvent nécessaire lors de l'établissement de la Postcondition des fonctions (voir INFO0946, Chapitre 6, Slide 11). En effet, le résultat retourné par l'invocation de la fonction est représenté par le nom de la fonction dans la Postcondition. Le nom de la fonction doit donc être défini avant d'en parler.

6.4.2 Suite de l'exercice

Spécifiez le problème. rendez-vous à la Sec. 6.4.3 pour la correction !

6.4.3 Spécification du problème

Précondition Il n'y a, ici, rien de particulier. On accepte, en entrée, n'importe quelle liste, même vide.

$$Pre : /$$

Postcondition On remarque que le but du programme est de calculer le nombre d'occurrences d'une variable `X` dans la liste `L`. Il n'y a aucune raison de ne pas réutiliser la notation fort convenable définie dans la formulation récursive :

$$Post : comptage = Comptage(L, X)$$

Alerte : Studentite aigüe !

Si vous n'avez pas une Postcondition aussi simple, c'est que vous avez contracté une *studentite aigüe*, la maladie caractérisée par une inflammation de l'étudiant qui cherche midi à quatorze heures.

Heureusement, ce n'est pas dangereux tant que vous êtes confiné ! Faites une petite pause dans cette résolution. Prenez le temps de vous oxygéner, en faisant un peu d'exercice physique par exemple. Trois répétitions de dix pompes devraient faire l'affaire.

Signature Les entiers se représentent avec le type `int`.

```
1 int comptage(List *L, int X);
```

Suite

Passez maintenant à la construction du code : Sec. 6.5.

6.5 Construction du Code

Il faut maintenant construire le programme en suivant *l'approche constructive*. La Section suivante vous fournit un petit rappel.

Le corrigé est disponible à la Sec. 6.5.2.

6.5.1 Approche constructive et récursion

Les principes de base de l'approche constructive ne changent évidemment pas si le programme construit est récursif :

- Il faut vérifier que la **Précondition** est respectée en pratiquant la **programmation défensive** ;
- Le programme sera construit autours d'une instruction conditionnelle qui discrimine le·s cas de base des cas récursifs ;
- Chaque **cas de base** doit amener la **Postcondition** ;
- **Le cas récursif** devra contenir (au moins) un **appel récursif** ;
- Pour tous les appels à des sous-problème (y compris l'appel récursif), il faut vérifier que la **Précondition** du module que l'on va invoquer est respectée
- Par le principe de l'approche constrictive, la **Postcondition** des sous-problèmes invoqués est respectée après leurs appels.

Il faut respecter ces quelques règles en écrivant le code du programme. La meilleure façon de procéder consiste à suivre d'assez près la formulation récursive du problème que l'on est en train de résoudre. Le code est souvent une « traduction » de cette fameuse formulation. Il ne faut pas oublier les **assertions intermédiaires**. Habituellement, elles ne sont pas tellement compliquées à fournir.

Qu'en est-il de `INIT`, `B`, `CORPS` et `FIN` ?

Essayez de suivre ! Ces zones correspondent au code produit lorsqu'on écrit une boucle ! S'il n'y a pas de boucle, il n'y aura pas, par exemple, de gardien de boucle !

Et la fonction de terminaison ?

Là non plus, s'il n'y a pas de boucle, on ne peut pas en fournir une. **Par contre**, on s'assure qu'on a pas oublié le·s **cas de base** et que les différents **appels récursifs** convergent bien vers un des cas de base (la convergence vers un des cas de base est appelée *condition de terminaison* – Chapitre 4, Slides 22 → 27).

Suite de l'exercice

À vous maintenant de construire le programme par l'approche constructive. Un corrigé est disponible à la Sec. 6.5.2.

6.5.2 Approche constructive

6.5.2.1 Programmation défensive

Il n'y a ici rien à faire, puisque notre fonction peut accepter en entrée n'importe quelle liste.

6.5.2.2 Cas de base

On gère le cas de base où $L = (\ )$.

```
// {Pre}
if(is_empty(L))
  // {L = ( )}
  return 0;
  // {Post}
```

6.5.2.3 Cas récursif

Il y a un seul cas récursif (cfr. Sec. 6.3.3). Il suffit de comparer la partie utile de la première cellule (`L->data`) avec la variable d'intérêt (`X`). On pourrait ici introduire une structure conditionnelle mais on peut, adéquatement, tirer profit de l'évaluation des expressions de comparaison. [7] Ainsi, si l'expression `L->data==X` est vraie, elle sera évaluée à 1 (0 sinon). Il suffit donc d'ajouter l'évaluation de cette expression à l'appel récursif.

```
else
  // {L ≠ ( ) ⇒ L+ = (data, _)}
  // {Pre_REC}
  return (L->data == X) + comptage(L->next, X);
  // {Post_REC : comptage = Comptage(L− > next, X)}
  // { ⇒ Post}
```

Au niveau des assertions intermédiaires, on vérifie (trivialement) la Précondition avant l'appel à `comptage` et après l'appel, sa Postcondition est vérifiée : on obtient donc facilement la Postcondition du code appelant.

Suite de l'exercice

Le programme final est donné à la Section suivante : 6.6.

7. Voir INFO0946, Chapitre 1, Slide 22.

6.6 Programme final

```
1  int comptage(List *L, int X){
2     // {Pre}
3     if(is_empty(L))
4     // {L = ( )}
5     return 0;
6     // {Post}
7     else
8     // {L ≠ ( ) ⇒ L+ = (data, _)}
9     // {Pre_REC}
10    return (L->data == X) + comptage(L->next, X);
11    // {Post_REC : comptage = Comptage(L− > next, X)}
12    // { ⇒ Post}
13 }
```