Université de Liège

INFO0947

Compléments de Programmation

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

Listes Doublement Chainées

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 Prérequis

Il est conseillé de débuter par les exercices concernant les listes simplement chainée. Faites-en au moins un avant de faire celui-ci.

6.1.2 Liste Doublement Chainée ?

Une *liste doublement chainée* (cfr. Chapitre 6, Slides 101 → 104), tout comme une liste chainée, se présente comme une succession de *cellules*, reliées l'une à l'autre. La majeure différence réside dans le fait que les cellules de la liste doublement chainée sont sont reliées par deux pointeurs (d'où le nom « **doublement** chainée ») : l'un pointe vers la cellule suivante et l'autre vers la cellule précédente.

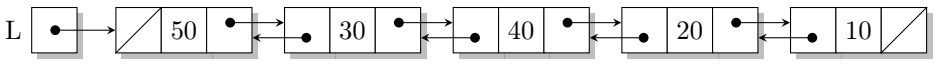


FIGURE 1 – Exemple de liste doublement chainée à 5 cellules.

On représente une liste doublement chainée en dessinant une succession de *cellules*, chacune comprenant trois parties :

— la partie utile (qui sert à stocker la donnée – dans l'exemple de la Fig. 1 des entiers) ;

— une partie liante qui permet de pointer vers la cellule suivante dans la liste (flèche vers la droite dans l'exemple de la Fig. 1) ;

— une partie liante qui permet de pointer vers la cellule précédente dans la liste (flèche vers la gauche dans l'exemple de la Fig. 1).

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 vers la cellule suivante contient la valeur particulière `NULL`. On peut aussi revenir en arrière dans la liste. On sait alors que la tête est atteinte quand la partie liante vers la cellule précédente contient la valeur particulière `NULL`.

En quoi cela diffère-t-il des tableaux ? Une liste (doublement) 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. L'avantage d'une liste doublement chainée est de pouvoir avancer vers la fin de la liste et de pouvoir aussi reculer jusqu'à son début facilement.

6.1.3 Notations sur les Listes

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

Comme vous pouvez le constatez, si vous avez suivi la recommandation de commencer par le rappel sur les listes simplement chainée, les notations sont identiques.

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 $L^+ = (e_0 \ldots e_{n-1})$	$L^- = ()$	-

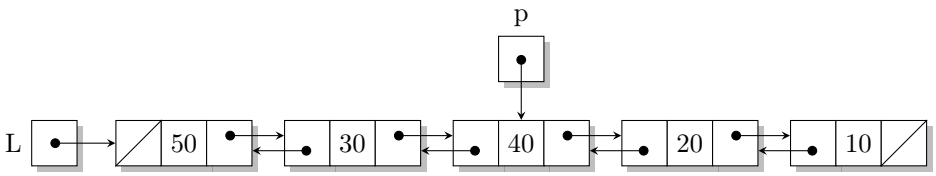


FIGURE 2 – Exemple de liste doublement chainée à 5 cellules.

6.1.4 Rappels sur l'Implémentation

6.1.4.1 Header (`list.h`)

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

typedef struct dllist_t DLList;

DLList *empty_list();

Boolean is_empty(DLList *L);

int length(DLList *L);

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

6.1.4.2 Module (`dlinked_list.c`)

La grosse différence entre les listes simplement et doublement chainée réside dans la présence ici des deux pointeurs `prev` et `next` qui permettent d'accéder respectivement à la cellule précédente et à la cellule suivante.

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

#include "dllist.h"

struct dllist_t{
  struct dllist_t *prev;
  void *data;
  struct dllist_t *next;
};

typedef struct dllist_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 !
  n_cell->prev = NULL; // Idem

  return n_cell;
}//end create_cell()

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

6.2 Énoncé

Écrivez une procédure qui affiche à l'écran tous les éléments d'une liste doublement chainée, `L`.

6.2.1 Méthode de résolution

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

1. Spécifier le problème complètement (Sec. 6.3) ;
2. Chercher un Invariant (Sec. 6.5) ;
3. Construire le programme complètement (Sec. 6.6) ;
4. Rédiger le programme final (Sec. 6.7).

Variante

Il est évidemment possible de résoudre ce problème de manière récursive. N'hésitez pas à le résoudre de cette manière et à en discuter sur eCampus.

6.3 Spécification

Il faut maintenant spécifier le problème.

Pour un rappel sur la spécification, voyez la Section 6.3.1
La correction de la spécification est disponible à la Section 6.3.3

6.3.1 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.

6.3.2 Suite de l'exercice

Spécifiez le problème. rendez-vous à la Sec. 6.3.3 pour la correction !

6.3.3 Spécification du problème

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

$$Pre : /$$

Postcondition A la fin de la procédure, la partie utile de toutes les cellules de la liste aura été affichée sur la sortie standard. Il suffit donc d'écrire :

$$Post : L^+ = L_0^+ \wedge L^+ \text{ imprimé}$$

Signature

```
1 void printDL(DLList *L);
```

Suite

Passez maintenant à la découpe en Sous-Problèmes : Sec. 6.4.

6.4 Découpe en Sous-Problèmes

Pourquoi devrions-nous envisager une découpe en Sous-Problèmes dans cet exercice ? C'est la question qu'il faut se poser. Y'a-t-il quelque chose qu'il faudra exécuter plusieurs fois et qu'il faudrait mettre en évidence ?

Suite de l'exercice

Trouvez ce qu'il convient de calculer à l'aide d'un Sous-Problème (SP) et spécifiez-le ! Rendez-vous à la Sec. 6.4.1.

6.4.1 Spécification du Sous-Problème

Une fois n'est pas coutume, il n'y a, à première vue, pas besoin de sous-problème pour résoudre l'exercice. Si vous avez pensez à un sous-problème, faites vous faire dépister une éventuelle *studentite aigüe*[3] sur eCampus.

Puisqu'il n'y a pas de sous-problème, passons donc à l'Invariant ! Continuons vers la Sec. 6.5.

3. Symptomes déjà décrits dans les TPs précédents, c'est quand on sort son Bazooka pour tuer une mouche

6.5 Invariant du Problème

Il faut maintenant trouver un Invariant permettant de résoudre le problème.

Pour un rappel sur l'Invariant, voyez la Section 6.5.1
La correction de l'Invariant Graphique est disponible à la Section 6.5.2
La correction de l'Invariant Formel est disponible à la Section 6.5.3

6.5.1 Rappels sur l'Invariant

Si vous voyez directement ce qu'il faut faire, continuez en lisant les conseils 6.5.1.1.

Petit rappel de la structure du code (cfr. Chapitre 2, Slide 28) :

```
// {Pre}
INIT
// {Inv}
while(B){
   // {Inv ∧ B}
   ITER
   // {Inv}
}
// {Inv ∧ ¬B}
END
// {Post}
```

La première chose à faire, pour chaque sous-problème, est de déterminer un Invariant. La meilleure façon de procéder est de d'abord produire un Invariant Graphique et de le traduire ensuite formellement. Le code sera ensuite construit sur base de l'Invariant.

Règles pour un Invariant Satisfaisant

Un bon Invariant Graphique respecte ces règles :

1. La structure manipulée est correctement représentée et nommée ;
2. Les bornes du problèmes sont représentées ;
3. Il y a une (ou plusieurs) ligne(s) de démarcation ;
4. Chaque ligne de démarcation est annotée par une variable ;
5. Ce qu'on a déjà fait est indiqué par le texte et utilise des variables pertinentes ;
6. Ce qu'on doit encore faire est mentionné.

Ce sera un encore meilleur Invariant dès que le code construit sur sa base sera en lien avec lui.

6.5.1.1 Quelques conseils préalables

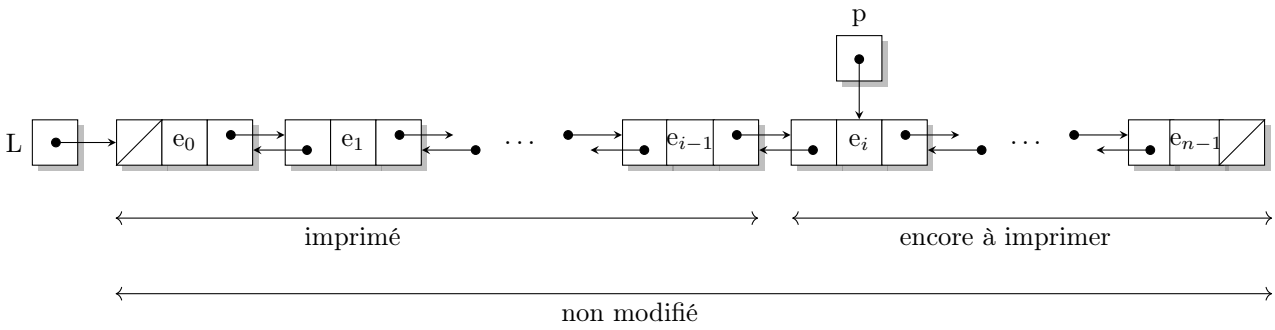
Un Invariant décrit particulièrement ce que l'on a déjà fait au fil des itérations précédentes. Dans le cadre d'une liste chainée, on possède une notation des éléments déjà vus : p^-. L'Invariant parlera donc **quasi toujours** de p^-. Ce qu'il reste à calculer correspond au traitement à appliquer à p^+.

Une bonne technique consiste à retravailler la Postcondition (cfr. Chapitre 2, Slide 29) et d'y faire apparaitre p^-. Il faut ensuite se poser la question : « à l'itération i, de quelle·s information·s ai-je besoin pour traiter le i^e élément que je m'apprête à lire dans la liste ? » (dans ce cas, on a évidemment $long(p^-) = i - 1$).

Suite de l'exercice

Pour l'Invariant Graphique 6.5.2
Pour l'Invariant Formel 6.5.3

6.5.2 Invariant Graphique

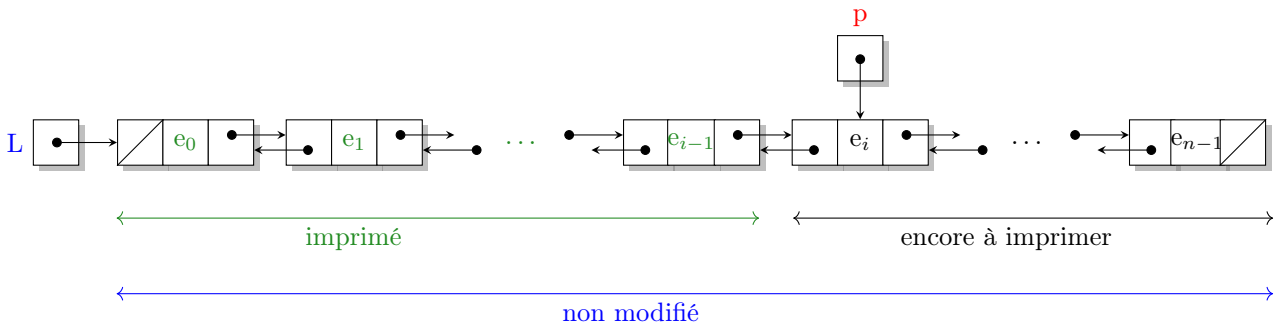


On commence par dessiner une liste, `L`. Puisqu'une liste (doublement) chainée est une structure séquentielle qui ne permet un parcourt que du début vers la fin (cfr. rappel) et qu'il ne faut pas perdre le pointeur de début, nous allons utiliser un pointeur temporaire pour le parcours : `p`.

Le pointeur `p` sert de ligne de démarcation sur le parcours de `L` (cfr. notations). La partie p^- a déjà été imprimée sur la sortie standard. La liste L^+ n'est pas modifiée.

La traduction de cet Invariant Graphique en Invariant Formel est disponible à la Sec. 6.5.3.

6.5.3 Invariant Formel



Pour décrire ce dessin en terme d'écriture formelle, la meilleure chose à faire est de repérer les différentes variables. Quelles sont-elles ? Il y a `L`, bien sûr, mais aussi `p`. Pour toutes ces variables, il faut se poser la question : « quelles propriétés ai-je représenté dans mon dessin ? ». Il ne faut pas oublier ce qui ne change (ou varie pas), point-clé de l'Invariant. S'inspirer de la version formelle de la Postcondition est aussi une (très) bonne idée.

On arrive à cet Invariant :

$$Inv : L^+ = L_0^+ \tag{1}$$
$$\wedge\, L^+ = p^- \,||\, p^+ \tag{2}$$
$$\wedge\, p^- \text{ a été imprimé} \tag{3}$$

1. Conservation des valeurs dans la liste `L`.
2. Présentation de `p` : c'est un pointeur vers une cellule de `L`.
3. Description du résultat : les valeurs de p^- ont été imprimées sur la sortie standard.

6.6 Construction du Code

Voici les différentes étapes de l'Approche Constructive (Chapitre 2, Slides 25 → 37)

INIT De la Précondition, il faut arriver dans une situation où l'Invariant est vrai.

B Il faut trouver d'abord la condition d'arrêt $\neg B$ et en déduire le gardien, B.

ITER Il faut faire progresser le corps de la boucle puis restaurer l'Invariant.

END Vérifier si la Postcondition est atteinte si $\{Inv \wedge \neg B\}$, sinon, amener la Postcondition.

Fonction t Il faut donner une fonction de terminaison pour montrer que la boucle se termine (cfr. INFO0946, Chapitre 3, Slides 97 → 107).

Tout ceci doit être justifié sur base de l'Invariant, à l'aide d'assertions intermédiaire. Notez que l'ordre dans lequel vous faites ces étapes n'a pas d'importance !

Suite de l'exercice

La suite de l'exercice est disponible à la Sec. 6.6.1.

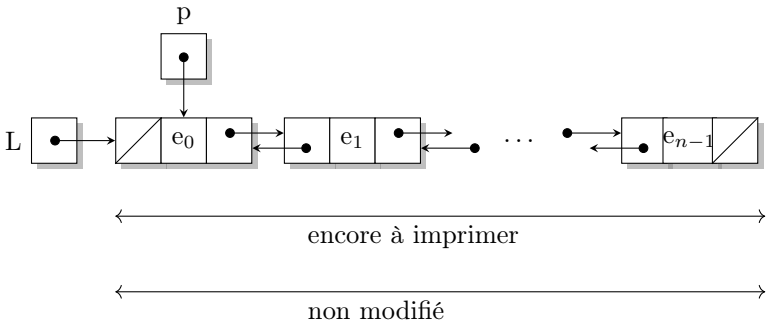
6.6.1 Approche constructive pour notre problème

6.6.1.1 Programmation défensive

On accepte tout en entrée puisque la liste vide est représentée par un pointeur `NULL`. On ne doit donc rien faire pour s'assurer que la Précondition est respectée.

6.6.1.2 INIT

Approche graphique Nous avions l'habitude dans les précédents Invariants[4] de dessiner la situation initiale en translatant une barre de démarcation. Ici, c'est le pointeur `p` qui tient ce rôle. Nous allons donc chercher à le positionner sur une cellule particulière de sorte que « ce qui a déjà été fait » (en l'occurrence ici : imprimer le contenu de cellule) corresponde à « rien ». La flèche « encore à imprimer » doit donc s'étendre sur toute la liste. On obtient alors le dessin suivant :



La correspondance entre `L` et `p` nous indique à quelle valeur initialiser ce dernier.

Approche formelle Il faut initialiser les variables de telles sortes que l'Invariant est vérifié. Une façon de faire consiste à regarder dans l'Invariant Formel ce qui correspond à ce qui a déjà été fait dans les itérations précédentes et d'exprimer que rien n'a encore été fait.

Dans notre cas, c'est « p^- a été imprimé ». Il faut exprimer que rien n'a été imprimé, soit :

$$(\ ) \text{ a été imprimé} \Rightarrow p^- = (\ )$$

En substituant $p^- = (\ )$ dans l'Invariant, on trouve :

$$L^+ = (\ ) \,||\, p^+ = p^+$$

Qui nous indique que `p` doit être initialisé à la valeur de `L`.

```
1 // {Pre}
2 DLList* p = L;
3 // {L+ = p− || p+ ∧ p− = ( ) ⇒ p− a été imprimé ⇒ Inv}
```

6.6.1.3 B

Déterminer le gardien demande de d'abord déterminer la condition d'arrêt, $\neg B$. Ceci est possible en comparant la Postcondition et l'Invariant (ne sont repris que les éléments utiles) :

4. C'était particulièrement vrai dans le cours INFO0946. N'hésitez pas à continuer à jouer avec le Gli. Attention, il n'est pas (encore) possible de dessiner des listes chainées (ou des fichiers) avec le Gli.

Inv : p^- a été imprimé

$Post$: L^+ a été imprimé

On doit doit donc s'arrêter lorsque $L^+ = p^-$. On sait en outre, par l'Invariant que $L^+ = p^- \,||\, p^+$. Pour que $L^+ = p^-$, il faut que $p^+ = (\ )$ qui est notre condition d'arrêt. En d'autre terme `p == NULL` est notre condition d'arrêt et le gardien est sa négation logique : `p != NULL` ou, plus simplement `p` puisque `NULL` a la valeur standard 0 qui correspond à faux. Toutes les autres valeurs de pointeurs sont donc vraies.

6.6.1.4 ITER

Il faut progresser vers la Postcondition, en d'autre termes, imprimer le contenu de la cellule pointée par `p`.

```
while(p){
    // {Inv ∧ B ⇒ p+ = (data) || p → next+ ∧ p−  imprimé ∧ Pre_printf}
    printf("%c ", p->data);
    // {Post_printf ⇒ p− ||(p− > data)  imprimé ∧ L+ = L+_0}
    p  = p->next;
    // {p−  imprimé L+ = L+_0 ⇒ Inv}
}
```

6.6.1.5 END

Il n'y a rien à faire puisque $\{Inv \wedge \neg B \equiv Post\}$.

6.6.1.6 Fonction de terminaison

Il faut déterminer une grandeur faisant partie du programme qui diminue à chaque itération. La longueur de ce qui reste à imprimer diminue comme cela. Il vient :

$$t = long(p^-)$$

Le programme final est disponible à la Sec. 6.7.

6.7 Programme final

```
1  void printDL(DLList *L){
2     // {Pre}
3     DLList* p = L;
4     // {L+ = p− || p+ ∧ p− = ( ) ⇒ p− a été imprimé ⇒ Inv}
5
6     while(p){
7        // {Inv ∧ B ⇒ p+ = (data) || p → next+ ∧ p− imprimé ∧ Pre_printf}
8        printf("%c ", p->data);
9        // {Post_printf ⇒ p− ||(p− > data) imprimé ∧ L+ = L+_0}
10       p  = p->next;
11       // {p− imprimé L+ = L+_0 ⇒ Inv}
12    }
13
14    // {Inv ∧ ¬B : L+ = p− || p+ ∧ p− imprimé ∧ p+ = () ⇒ L+ imprimé}
15    // {⇒ Post}
16 }
```