Université de Liège

INFO0947

Compléments de Programmation

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

Listes Chainées – Copie Carbone

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.

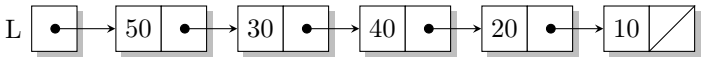


FIGURE 1 – Exemple de liste chainée à 5 cellules.

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

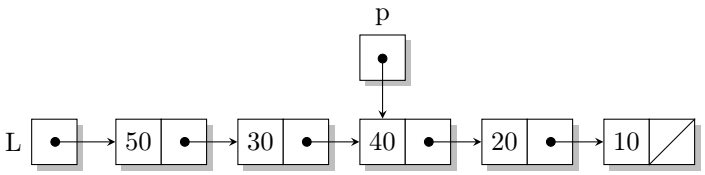


FIGURE 2 – Application des notations.

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

On considère une liste, donnée par son pointeur de début `L`. Écrivez une fonction qui recopie cette liste et retourne le pointeur de début de cette nouvelle liste. Il s'agit donc de réaliser une copie carbone de `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).

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

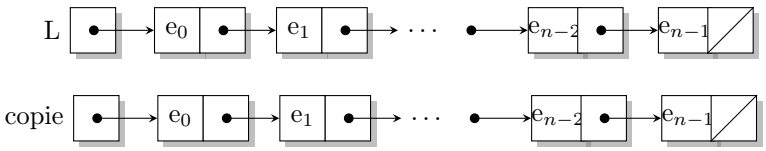
Spécifiez le problème. rendez-vous à la Sec. 6.3.3 pour la correction !

6.3.3 Spécification du problème

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

$$Pre : /$$

Postcondition On doit atteindre la situation représentée par le schéma suivant [4] :



Dans cette situation, un pointeur de début vers la liste copiée est retournée par la fonction. Il ne s'agit pas d'un pointeur vers la première cellule de `L`. Au contraire, le contenu des deux listes est identique, cellule par cellule. Il s'agit donc bien d'une copie carbone.

$$Post : copie^+ = L^+ \tag{1}$$
$$\wedge\, L^+ = L_0^+ \tag{2}$$
$$\wedge\, \forall (s,t), (L^+ = s^- \,||\, s^+ \wedge copie^+ = t^- \,||\, t^+ \wedge s \neq NULL \wedge t \neq NULL), s \neq t \tag{3}$$

Notre fonction s'appellera `copie` et renverra un pointeur. $copie^+$ représente donc la liste copiée. La Postcondition rassemble trois éléments :

1. Les listes L^+ et $copie^+$ ont les mêmes éléments ;
2. Les éléments de la liste pointée par L n'ont pas été modifiés ;
3. On a exprimé que le contenu des listes était identique mais il faut en outre exprimer que toutes les parties liantes sont différentes. Le mot « toutes » suggère donc une quantification universelle sur des parties liantes de L^+ et de $copie^+$. Les variables liées correspondantes (qui sont donc des pointeurs) seront appelée s et t. Il faut dire « pour tout s qui est un pointeur de la liste L^+ et t qui est un pointeur de la liste $copie^+$, s et t sont différents ». C'est exactement ce qui est exprimé ici puisque « p est une partie liante d'une liste L^+ » s'écrit : $L^+ = p^- \,||\, p^+$.
   Dans la condition du « pour tout », il faut empêcher que s et t soit égaux à `NULL` sinon la quantification universelle devient trivialement fausse à cause de l'existence d'une cellule de fin dans les deux listes ! Cela alourdit la notation mais c'est nécessaire.

Signature

```
1 List *copie(List *L);
```

Suite

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

4. Commencer par un dessin est toujours une bonne idée. Cela facilite l'écriture formelle (il suffit d'une simple traduction).

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*[5] sur eCampus.

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

5. 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

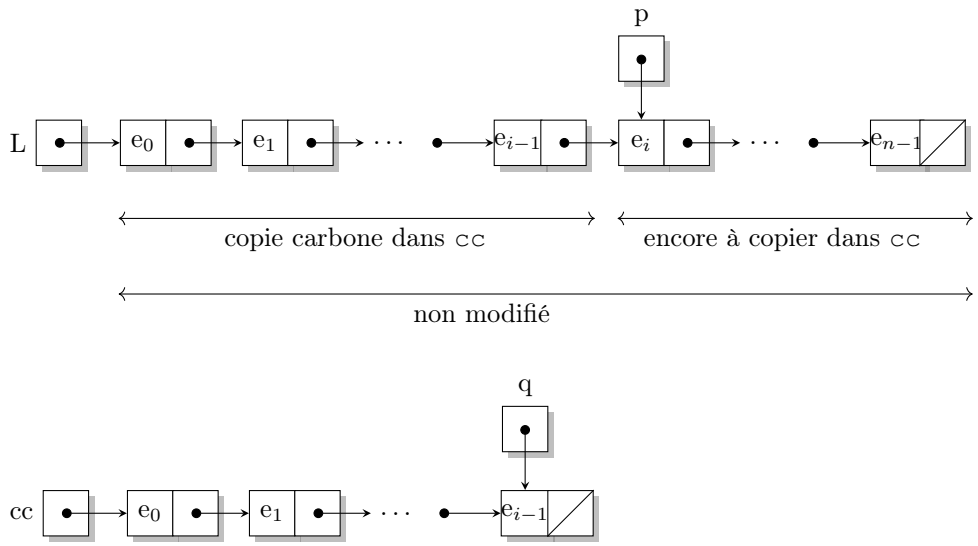
Un Invariant décrit particulièrement ce que l'on a déjà fait au fil des itérations précédentes. Dans le cadre d'une liste chainée, on possède une notation des éléments déjà vus : p^-. L'Invariant parlera donc **quasi toujours** de p^-. Ce qu'il reste à calculer correspond au traitement à appliquer à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



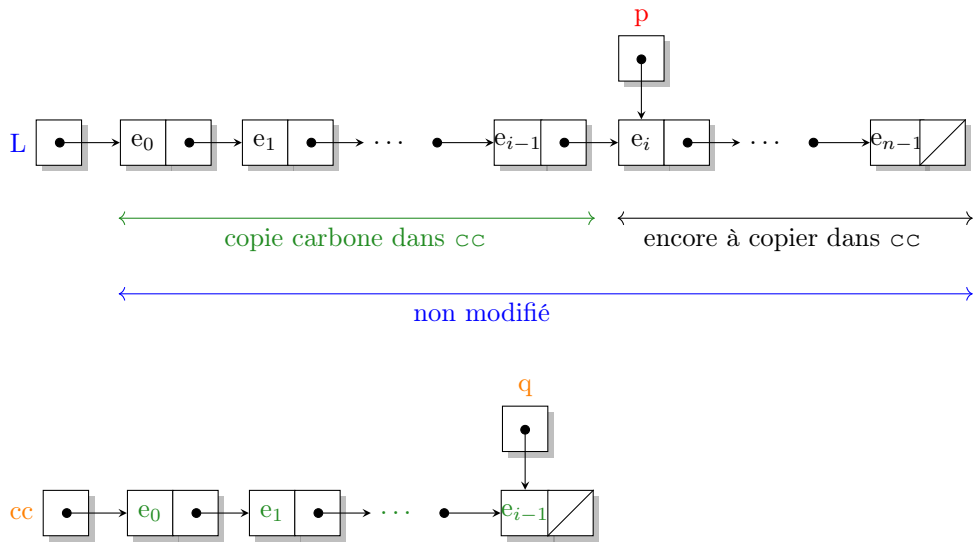
On commence par dessiner deux listes : `L`, la liste de départ, et `cc`, la liste **c**opie **c**arbone (partielle) de `L`. Puisqu'une liste chainée est une structure séquentielle qui ne permet un parcourt que du début vers la fin (cfr. Rappel) et qu'il ne faut pas perdre le pointeur de début, nous allons utiliser deux pointeurs temporaires : (*i*) `p` qui permet le parcourt de `L` et (*ii*) `q` qui permet d'ajouter en fin de la liste copiée `cc`. Dès lors, `q` doit toujours pointer sur la dernière cellule de `cc`.

Le pointeur `p` sert de ligne de démarcation sur le parcours de `L` (cfr. notations – Sec. 6.1.2). La partie p^- a déjà été copiée (copie carbone) dans `cc`. La liste L^+ n'est pas modifiée.

Suite de l'exercice

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` et `cc`, bien sûr, mais aussi `p` et `q`. 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 Formel :

$$Inv : L^+ = L_0^+ \tag{1}$$
$$\wedge\, L^+ = p^- \,||\, p^+ \tag{2}$$
$$\wedge\, cc^+ = q^- \,||\, q^+ \tag{3}$$
$$\wedge\, q- > next^+ = (\ ) \tag{4}$$
$$\wedge\, cc^+ = p^- \tag{5}$$
$$\wedge\, \forall (s,t), (L^+ = s^- \,||\, s^+ \wedge cc^+ = t^- \,||\, t^+ \wedge s \neq NULL \wedge t \neq NULL), s \neq t \tag{6}$$

1. Conservation des valeurs dans la liste `L`.
2. Présentation de `p` : c'est un pointeur vers une cellule de `L`.
3. Présentation de `q` : c'est un pointeur vers une cellule de `cc`.
4. Propriété de `q` : il doit toujours être sur la dernière cellule de la liste `cc`.
5. Description du résultat : les valeurs de p^- et cc^+ sont identiques.
6. Description du résultat : pas de pointeurs non-nuls communs aux listes `L` et `cc` (tout est expliqué à la Sec. 6.3).

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

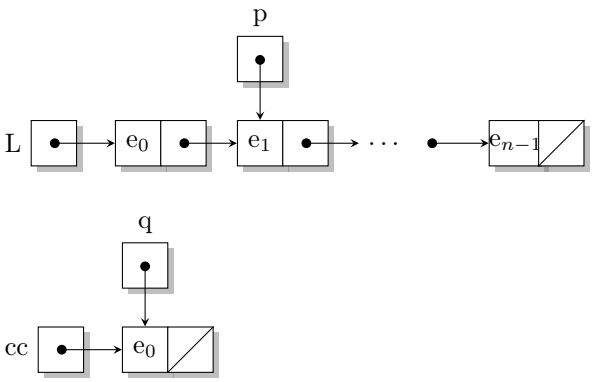
La suite de l'exercice est disponible à la Sec. 6.6.1.

6.6.1 Approche constructive pour notre problème

6.6.1.1 INIT

Il est **obligatoire** de se positionner dans une situation où l'Invariant est vrai. Dès qu'on l'observe, on voit, dans l'écriture formelle, que l'on parle de `q->next`. Cela veut donc dire que, pour entrer dans l'itération, la liste `cc` doit contenir au moins un élément. Nous traiterons donc le cas spécial où `L` est vide plus tard.

Représentons cette situation initiale :



Il faut donc copier le contenu de la première cellule et initialiser tous les pointeurs en conséquence :

```
// {Pre ∧ L+ ≠ ( )}
List *p = L->next;
List *cc  = create_cell(L->data);
// {L+ = p− || p+ ∧ cc+ = p− ∧ ∀(s,t),(L+ = s− || s+ ∧ cc+ = t− || t+ ∧ s ≠ NULL ∧ t ≠ NULL), s ≠ t}
List *q = cc;
// {cc+ = q− || q+ ∧ q− > next+ = ( )
//  ∧ L+ = p− || p+ ∧ cc+ = p− ∧ ∀(s,t),(L+ = s− || s+ ∧ cc+ = t− || t+ ∧ s ≠ NULL ∧ t ≠ NULL), s ≠ t ∧ L+ = L0+ ⇒ Inv}
```

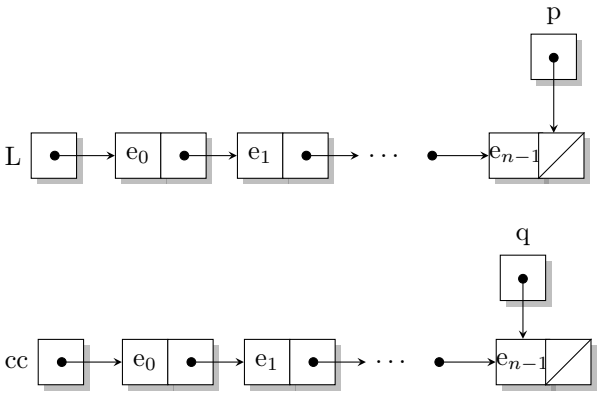
6.6.1.2 Cas particulier : liste vide

Si la liste est vide, il n'est pas question d'entrer dans la boucle. Il convient donc de simplement retourner un pointeur NULL, qui correspond bien à la liste vide qui est donc correctement copiée !

```
if (is_empty(L))
    // {L+ = ()}
    return empty_list();
    // {Post : copy+ = L0+}
```

6.6.1.3 B

On commence par définir la condition d'arrêt en comparant la Postcondition (Sec. 6.3.3) et l'Invariant (Sec. 6.5.2).



On s'aperçoit que le pointeur temporaire `p` doit être arrivé au bout de la liste `L` (remarquez que dans le dessin, il « pointe » bien vers NULL). Ceci se manifeste quand la partie liante de la cellule courante vaut `NULL`.

La condition d'arrêt est donc :

$$B \equiv p^{+} = ()$$

Dit autrement :

$$B \equiv p == NULL$$

On en déduit, naturellement, le gardien de boucle :

$$\neg B \equiv p \neq NULL$$

Alerte : Erreur classique

Sur base du raisonnement donné ci-dessus, il est (souvent) tentant de définir la condition d'arrêt comme étant :

$$p \rightarrow \text{next} == NULL$$

Ce qui est une traduction directe de la phrase « la partie liante de la cellule courante vaut `NULL` ». C'est incorrect. Une telle condition d'arrêt amènerait, imanquablement, à s'arrêter un tour trop tôt. Pour s'en convaincre, il suffit de transformer la condition d'arrêt en gardien de boucle et de voir ce qu'il se passerait sur une liste assez simple (e.g., deux cellules). La dernière cellule ne serait clairement pas copiée dans `cc`.

6.6.1.4 ITER

Nous nous trouvons dans une situation où l'Invariant est vrai et ou le gardien est vrai également, cela signifie qu'on est pas encore arrivé au bout de la liste L. Il faut progresser en copiant la cellule pointée par p, la mettre à la fin de la liste cc grâce à q et ensuite, il faut rétablir l'Invariant.

Note importante Dans les assertions intermédiaires suivantes, pour plus de lisibilité, on ne rappellera pas à chaque ligne que

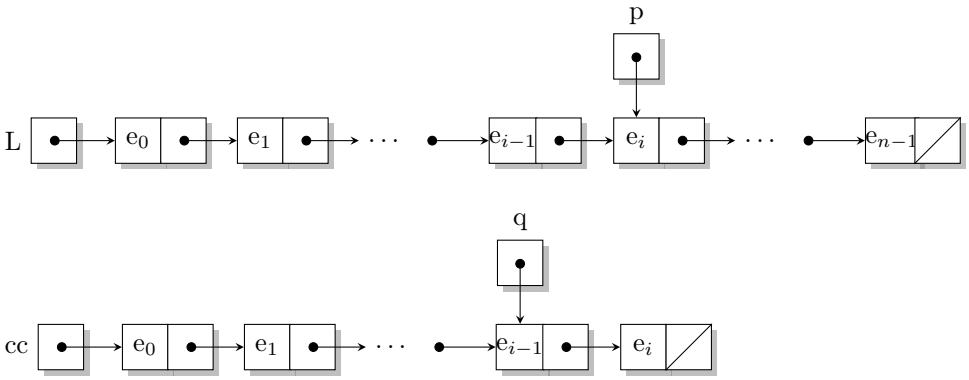
$$L^+ = L_0^+ \wedge \forall (s,t), (L^+ = s^- \,||\, s^+ \wedge cc^+ = t^- \,||\, t^+ \wedge s \neq NULL \wedge t \neq NULL), s \neq t \wedge L^+ = p^- \,||\, p^+ \wedge cc^+ = q^- \,||\, q^+$$

Ce n'est évidemment pas incorrect de le recopier à chaque ligne [6].

```
1 while(p != NULL){
2    // {Inv ∧ B}
3    q->next = create_cell(p->data);
4    // {cc+ = p− ||(p− > data) ∧ q− > next+ = (p− > data)}
5    q = q->next;
6    // {cc+ = p− ||(p− > data) ∧ q− > next+ = ( )}
7    p = p->next;
8    // {cc+ = p− ∧ q− > next+ = ( ) ⇒ Inv}
9 }
```

Dans les assertions, dès qu'on atteint $\{Inv\}$, on ne fait plus rien dans la boucle et on passe à l'itération suivante.

La ligne de code 3 s'illustre de la façon suivante :



6.6.1.5 END

En combinant $\neg B$ et l'Invariant, on a :

$$p^+ = (\,) \wedge (L^+ = L_0^+ \wedge cc^+ = q^- \,||\, q^+ \wedge q- > next^+ = (\,) \wedge L^+ = p^- \,||\, p^+ \wedge cc^+ = p^- \\ \wedge \forall (s,t), (L^+ = s^- \,||\, s^+ \wedge cc^+ = t^- \,||\, t^+ \wedge s \neq NULL \wedge t \neq NULL), s \neq t)$$

Il vient que :

$$(L^+ = p^- = cc^+ = L_0^+ \wedge \forall (s,t), (L^+ = s^- \,||\, s^+ \wedge cc^+ = t^- \,||\, t^+ \wedge s \neq NULL \wedge t \neq NULL), s \neq t) \Rightarrow Post$$

Il n'y a donc rien à faire à part retourner le pointeur `cc`

6.6.1.6 Fonction de terminaison

La liste chainée n'a pas une taille infinie. Trouver ce qui décroit à chaque itération est simple : $long(p^+)$ (voir la notation).

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

Le programme final est disponible à la Sec. 6.7.

6. Idée confinement : faites le donc si vous vous ennuyez ;-)

6.7 Programme final

```
1  List *copy(List* L){
2     //{Pre}
3
4     if (is_empty(L))
5        // {L+ = ()}
6        return empty_list();
7        // {Post : copy+ = L0+}
8
9     // {Pre ∧ L+ ≠ ( )}
10    List *p = L->next;
11    List *cc  = create_cell(L->data);
12    // {L+ = p− || p+ ∧ cc+ = p− ∧ ∀(s, t), (L+ = s− || s+ ∧ cc+ = t− || t+ ∧ s ≠ NULL ∧ t ≠ NULL), s ≠ t}
13    List *q = cc;
14    // {cc+ = q− || q+ ∧ q− > next+ = ( )
15    //  ∧ L+ = p− || p+ ∧ cc+ = p− ∧ ∀(s, t), (L+ = s− || s+ ∧ cc+ = t− || t+ ∧ s ≠ NULL ∧ t ≠ NULL), s ≠ t ∧ L+ = L0+
16    //  ⇒ Inv}
17
18    while(p != NULL){
19       // {Inv ∧ B}
20       q->next = create_cell(p->data);
21       // {cc+ = p− ||(p− > data) ∧ q− > next+ = (p− > data)}
22       q = q->next;
23       // {cc+ = p− ||(p− > data) ∧ q− > next+ = ( )}
24       p = p->next;
25       // {cc+ = p− ∧ q− > next+ = ( ) ⇒ Inv}
26    }
27    // {Inv ∧ ¬B ⇒ L+ = p− = cc+ = L0+}
28    return cc;
29    // {Post}
30 }
```