

Compléments de Programmation

Benoit Donnet
Année Académique 2023 - 2024



Agenda

- Chapitre 1: Raisonnement Mathématique
- Chapitre 2: Construction de Programme
- Chapitre 3: Introduction à la Complexité
- Chapitre 4: Récursivité
- Chapitre 5: Types Abstraits de Données
- **Chapitre 6: Listes**
- Chapitre 7: Piles
- Chapitre 8: Files
- Chapitre 9: Elimination de la Récursivité

Agenda

- Chapitre 6: Listes
 - Principe
 - Spécification Abstraite
 - Implémentation
 - Algorithmique
 - Bestiaire

Agenda

- Chapitre 6: Listes
 - Principe
 - ✓ Idée
 - ✓ Vocabulaire
 - ✓ Opérations
 - Spécification Abstraite
 - Implémentation
 - Algorithmique
 - Bestiaire

Idée

- Comment faire pour conserver en mémoire une quantité inconnue d'information?
- On veut pouvoir ajouter/supprimer des éléments en fonction des besoins
 - la structure doit donc être *dynamique*
- Solution
 - Liste

Idée (2)

- Une liste est une *suite finie, éventuellement vide*, d'éléments de *même type* repérés par leur **rang** dans la liste
- Chaque élément de la liste est rangé à une certaine **place**
- Exemples
 - $L = (4, 1, 7, 3, 1)$
 - $L = ()$
 - ✓ liste vide

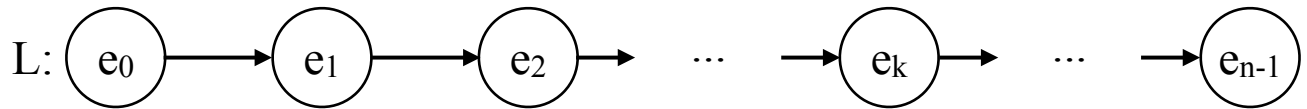
Vocabulaire

- Le nombre total d'éléments dans la liste est appelé **longueur de la liste**
 - la liste vide a une longueur de 0
 - exemples
 - ✓ $L = (4, 1, 7, 3, 1)$
 - longueur = 5
 - ✓ $L = ()$
 - longueur = 0
- Le premier élément de la liste est appelée **tête de liste**
 - exemple
 - ✓ $L = (4, 1, 7, 3, 1)$
 - tête de liste = 4

Vocabulaire (2)

- Le **rang** est la position de l'élément dans la liste
 - exemple
 - ✓ $L = (4, 1, 7, 3, 1)$
 - l'élément 4 est au rang 0
 - l'élément 1 est au rang 1
 - l'élément 7 est au rang 2
 - l'élément 3 est au rang 3
 - l'élément 1 est au rang 4
- De manière générale, pour une liste à n éléments
 - $L = (e_0, e_1, \dots, e_i, \dots, e_{n-1})$
 - e_i désigne l'élément de rang i

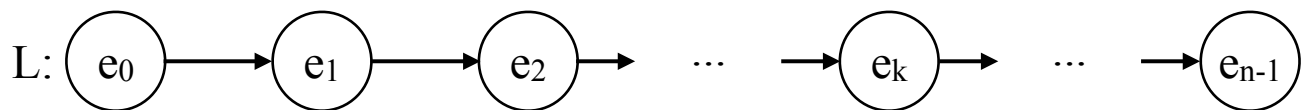
Opérations



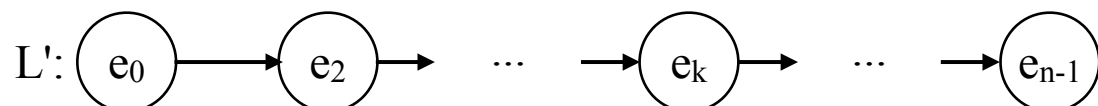
$\text{length}(L) \rightarrow n$

$\text{get}(L, 2) \rightarrow e_2$

Opérations (2)

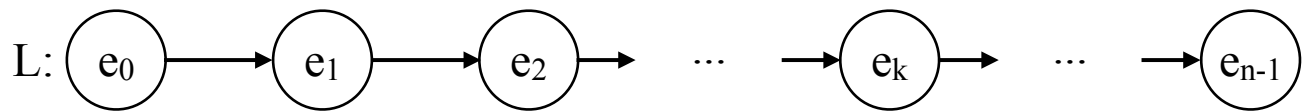


$\text{remove}(L, 1) \rightarrow L'$

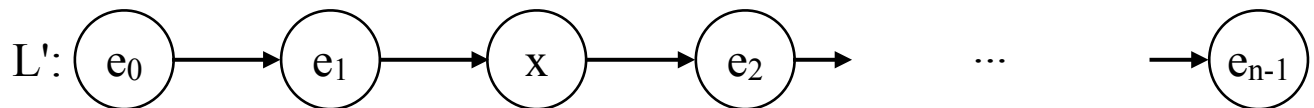


$\text{length}(L') \rightarrow n-1$

Opérations (3)



$\text{add_at}(L, 2, x) \rightarrow L'$



$\text{get}(L', 2) \rightarrow x$
 $\text{length}(L') \rightarrow n+1$

Agenda

- Chapitre 6: Listes
 - Principe
 - Spécification Abstraite
 - Implémentation
 - Algorithmique
 - Bestiaire

Spécification Abstraite

- Syntaxe

Type:

List

Utilise:

Boolean, Element, Integer

Opérations:

empty_list: \rightarrow List

is_empty: List \rightarrow Boolean

length: List \rightarrow Integer

get: List \times Integer \rightarrow Element

set: List \times Integer \times Element \rightarrow List

add_at: List \times Integer \times Element \rightarrow List

add_first: List \times Element \rightarrow List

add_last: List \times Element \rightarrow List

remove: List \times Integer \rightarrow List

contains: List \times Element \rightarrow Boolean

Spécifications Abstraites (2)

- Sémantique
 - préconditions

Préconditions:

$\forall L \in \text{List}, \forall e \in \text{Element}, \forall i \in \text{Integer}$

$\forall i, 0 \leq i \leq \text{length}(L), \text{add_at}(L, i, e)$

$\forall i, 0 \leq i < \text{length}(L), \text{remove}(L, i)$

$\forall i, 0 \leq i < \text{length}(L), \text{get}(L, i)$

$\forall i, 0 \leq i < \text{length}(L), \text{set}(L, i)$

Spécifications Abstraites (3)

		Opérations Internes					
		<code>empty_list</code>	<code>set(·)</code>	<code>add_first(·)</code>	<code>add_last(·)</code>	<code>add_at(·)</code>	<code>remove(·)</code>
Observateurs	<code>length(·)</code>						
	<code>is_empty(·)</code>						
	<code>get(·)</code>						
	<code>contains(·)</code>						

- Sémantique
 - axiomes
- Combien d'axiomes?
 - 4 Observateurs \times 6 Opérations Internes = 24 Axiomes
- Peut-on réduire?
 - exprimer `add_first()` et `add_last()` en fonction de `add_at()`

Spécifications Abstraites (4)

Axiomes:

$\forall L \in \text{List}, \forall e \in \text{Element}, \forall i \in \text{Integer}$
`add_first(L, e) = add_at(L, 0, e)`
`add_last(L, e) = add_at(L, length(L), e)`

Spécifications Abstraites (5)

Axiomes:

$\forall L \in \text{List}, \forall e \in \text{Element}, \forall i \in \text{Integer}$
 $\text{length}(\text{empty_list}) = 0$
 $\text{length}(\text{remove}(L, i)) = \text{length}(L) - 1$
 $\text{length}(\text{add_at}(L, i, e)) = \text{length}(L) + 1$
 $\text{length}(\text{set}(L, i, e)) = \text{length}(L)$

		Opérations Internes					
		<code>empty_list</code>	<code>set(·)</code>	<code>add_first(·)</code>	<code>add_last(·)</code>	<code>add_at(·)</code>	<code>remove(·)</code>
Observateurs	<code>length(·)</code>	✓	✓	✓	✓	✓	✓
	<code>is_empty(·)</code>						
	<code>get(·)</code>						
	<code>contains(·)</code>						

Spécifications Abstraites (6)

Axiomes:

$\forall L \in \text{List}, \forall e, f \in \text{Element}, \forall i, j \in \text{Integer}$
 $\text{is_empty}(\text{empty_list}) = \text{True}$
 $\text{is_empty}(\text{add_at}(L, i, e)) = \text{False}$
 $\text{is_empty}(\text{set}(L, i, e)) = \text{False}$
 $\text{is_empty}(\text{remove}(L, i)) = \begin{cases} \text{False} & \text{If } \text{Length}(L) > 1 \\ \text{True} & \text{If } \text{Length}(L) = 1 \end{cases}$

		Opérations Internes					
		<code>empty_list</code>	<code>set(·)</code>	<code>add_first(·)</code>	<code>add_last(·)</code>	<code>add_at(·)</code>	<code>remove(·)</code>
Observateurs	<code>length(·)</code>	✓	✓	✓	✓	✓	✓
	<code>is_empty(·)</code>	✓	✓	✓	✓	✓	✓
	<code>get(·)</code>						
	<code>contains(·)</code>						

Spécifications Abstraites (7)

Axiomes:

$\forall L \in \text{List}, \forall e \in \text{Element}, \forall i \in \text{Integer}$

$$\text{get}(\text{set}(L, i, e), j) = \begin{cases} \text{get}(L, i) & \text{If } i = j \\ \text{get}(L, j) & \text{Otherwise} \end{cases}$$

$$\text{get}(\text{remove}(L, i), j) = \begin{cases} \text{get}(L, j+1) & \text{If } 0 \leq i \leq \text{length}(L) \wedge i < j < \text{length}(L) - 1 \\ \text{get}(L, j) & \text{Otherwise} \end{cases}$$

		Opérations Internes					
		empty_list	set(·)	add_first(·)	add_last(·)	add_at(·)	remove(·)
Observateurs	length(·)	✓	✓	✓	✓	✓	✓
	is_empty(·)	✓	✓	✓	✓	✓	✓
	get(·)	∅	✓				✓
	contains(·)						

Spécifications Abstraites (8)

Axiomes:

$\forall L \in \text{List}, \forall e, f \in \text{Element}, \forall i, j \in \text{Integer}$

$$\text{get}(\text{add_at}(L, i, e), j) = \begin{cases} e & \text{If } i = j \\ \text{get}(L, j) & \text{If } i \neq j \\ \text{get}(L, j-1) & 0 \leq i \leq \text{length}(L) \wedge i < j \leq \text{length}(L) \end{cases}$$

		Opérations Internes					
		empty_list	set(·)	add_first(·)	add_last(·)	add_at(·)	remove(·)
Observateurs	length(·)	✓	✓	✓	✓	✓	✓
	is_empty(·)	✓	✓	✓	✓	✓	✓
	get(·)	∅	✓	✓	✓	✓	✓
	contains(·)						

Spécifications Abstraites (9)

Axiomes :

$\forall L \in \text{List}, \forall e, f \in \text{Element}, \forall i, j \in \text{Integer}$

`contains(empty_list, e)` = False

`contains(set(L,i,e),f)` = $\begin{cases} \text{True} & \text{If } e = f \\ \text{contains}(L,f) & \text{Otherwise} \end{cases}$

`contains(add_at(L,i,e),f)` = $\begin{cases} \text{True} & \text{If } e = f \\ \text{contains}(L,f) & \text{Otherwise} \end{cases}$

`contains(L,e)` = $\begin{cases} \text{True} & \text{If } \text{get}(L,0) = e \\ \text{contains}(\text{remove}(L,0),e) & \text{Otherwise} \end{cases}$

		Opérations Internes					
		<code>empty_list</code>	<code>set(·)</code>	<code>add_first(·)</code>	<code>add_last(·)</code>	<code>add_at(·)</code>	<code>remove(·)</code>
Observateurs	<code>length(·)</code>	✓	✓	✓	✓	✓	✓
	<code>is_empty(·)</code>	✓	✓	✓	✓	✓	✓
	<code>get(·)</code>	∅	✓	✓	✓	✓	✓
	<code>contains(·)</code>	✓	✓	✓	✓	✓	✓

INFO0947 - ULiège - 2023/2024 - Benoit Donnet

21

Agenda

- Chapitre 6: Listes
 - Principe
 - Spécification Abstraite
 - Implémentation
 - ✓ Interface
 - ✓ Implémentation par Tableau
 - ✓ Implémentation par Pointeurs
 - ✓ Complexité
 - Algorithmique
 - Bestiaire

Interface

- Quelque soit l'implémentation, l'interface (i.e., le header) sera la même
- Le type opaque représentant une liste assure l'indépendance de l'interface par rapport à l'implémentation
- Le choix d'implémentation est laissé libre au programmeur
- L'utilisation "pratique" du TAD se fait via l'interface
 - c'est à la compilation qu'on choisit l'implémentation particulière

Interface (2)

- Fichier `list.h`

```
#ifndef __LIST__
#define __LIST__

#include "boolean.h"

typedef struct List_t List;

List *empty_list();

Boolean is_empty(List *L);

int length(List *L);

void *get(List *L, int i);

List *set(List *L, int i, void *e);
```

Interface (3)

```
List *add_at(List *L, int i, void *e);  
  
List *add_first(List *L, void *e);  
  
List *add_last(List *L, void *e);  
  
List *remove_list(List *L, int i);  
  
Boolean contains(List *L, void *e, Boolean (*equal_data)(void  
    *data1, void *data2));  
  
#endif
```

Implem. Tableau

- Représentation *contigüe* de la liste
- Les éléments sont rangés les uns à côté des autres dans un tableau
 - la case d'indice i fait référence à l'élément de rang i
 - ✓ permet un accès rapide à chaque élément
 - le rang est donc égal à l'indice
- La liste est donc un enregistrement contenant
 - un tableau
 - un entier indiquant le nombre d'éléments encodés dans le tableau
 - la taille du tableau
- Attention, on doit pouvoir augmenter (voire diminuer) à la volée la taille du tableau!

Implem. Tableau (2)

	0	1	2	3	4	5	6	7	8	9
L:	10	7	20	15						

`L = add_at(L, 1, 5);`

	0	1	2	3	4	5	6	7	8	9
L:	10	5	7	20	15					

→
décalage vers la droite

Implem. Tableau (3)

	0	1	2	3	4	5	6	7	8	9
L:	10	5	7	20	15					

`L = remove(L, 0);`

	0	1	2	3	4	5	6	7	8	9
L:	5	7	20	15						

←
décalage vers la gauche

Implem. Tableau (4)

- Fichier `array_list.c`

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

```
#include "list.h"
```

```
#define INITIAL_SIZE 10
```

taille initiale du tableau

```
#define REALLOCATION_FACTOR 2
```

facteur multiplicateur

```
struct list_t{
    int array_size;
    int length;
    void **array;
};
```

taille actuelle du tableau
nombre d'élément dans la liste
tableau générique

Implem. Tableau (5)

```
static List *realloc_array(List *L){
    assert(L!=NULL);

    L->array_size *= REALLOCATION_FACTOR;
    L->array = realloc(L->array, L->array_size*sizeof(void *));

    return L;
} //end realloc_array()

static List *shift_right(List *L, int begin){
    //déplace les éléments d'une position vers la droite
    //à faire à la maison, en suivant l'approche constructive
} //end shift_right()

static List *shift_left(List *l, int begin){
    //déplace les éléments d'une position vers la gauche
    //à faire à la maison, en suivant l'approche constructive
} //end shift_left()
```

Implem. Tableau (6)

```
List *empty_list(){
    List *L = malloc(sizeof(List));
    if(L==NULL)
        return NULL;

    L->array_size = INITIAL_SIZE;
    L->length = 0;

    L->array = malloc(INITIAL_SIZE * sizeof(void *));
    if(L->array==NULL){
        free(L);
        return NULL;
    }

    return L;
} //end empty_list()

Boolean is_empty(List *L){
    return L->length==0;
} //end is_empty()
```

Implem. Tableau (7)

```
int length(List *L){
    return L->length;
} //end length()

void *get(List *L, int i){
    assert(0<=i && i<L->length);

    return L->array[i];
} //end get()

List *set(List *L, int i, void *e){
    assert(0<=i && i<L->length);

    if(L->array[i]!=NULL)
        free(L->array[i]);

    L->array[i] = e;

    return L;
} //end set()
```


Implem. Tableau (8)

```
List *add_at(List *L, int i, void *e){
    assert(0<=i && i<L->array_size);

    if(L->array[i]!=NULL)
        L = shift_right(L, i);

    L->array[i] = e;
    L->length++;

    return L;
} //end add_at()

List *add_first(List *L, void *e){
    if(L->length!=0)
        L = shift_right(L, 1);

    L->array[0] = e;
    L->length++;

    return L;
} //end add_first()
```

Implem. Tableau (9)

```
List *add_last(List *L, void *e){
    if(L->length + 1 >= L->array_size)
        L = realloc_array(L);

    L->array[L->length] = e;
    L->length++;

    return L;
} //end add_last()

List *remove(List *L, int i){
    assert(0<=i && i<L->length);

    free(L->array[i]);

    L = shift_left(L, i+1);
    free(L->array[L->length-1]);
    L->length--;

    return L;
} //end remove()
```

Implem. Tableau (10)

```
Boolean contains(List *L, void *e,  
                Boolean (*equal_data)(void *data1, void *data2)){  
    int i, eqn;  
  
    {Inv:  $L = L_0 \wedge 0 \leq i \leq \text{length}(L) \wedge e = e_0$   
         $\wedge \forall j, 0 \leq j \leq i-1, e \notin L \rightarrow \text{array}[j]$ }  
    for(i=0; i<L->length; i++){  
        eqn = (*equal_data)(e, L->array[i]);  
        if(eqn)  
            return True;  
    }//end for - i  
  
    return False;  
}//end contains()
```

Implem. Tableau (11)

- Fichier element.h

```
#ifndef __ELEMENT__  
#define __ELEMENT__  
  
#include "boolean.h"  
  
typedef struct{  
    int e;  
}Element;  
  
Element *create_element(int e);  
  
Boolean equal_element(void *e1, void *e2);  
  
void print_element(Element *e);  
  
#endif
```

Implem. Tableau (12)

- Fichier `test_array_list.c`

```
#include <assert.h>
#include "list.h"
#include "element.h"

static void print(List *L){
    {Pré: L ≠ NULL}
    assert(L!=NULL);
    int i;
    Element *elem;

    {Inv: L = L0 ∧ 0 ≤ i ≤ length(L) ∧ ∀ j, 0 ≤ j ≤ i-1,
        get(L, j) affiché}
    for(i=0; i<length(L); i++){
        elem = (Element *)get(L, i);
        print_element(elem);
    }//end for - i
    {Post: L affiché}
} //end print()
```

Implem. Tableau (13)

```
int main(){
    List *L = empty_list();

    L = add_first(L, create_element(10));
    L = add_at(L, 1, create_element(7));
    L = add_at(L, 2, create_element(20));
    L = add_last(L, create_element(15));

    print(L);

    L = add_at(L, 1, create_element(5));

    print(L);

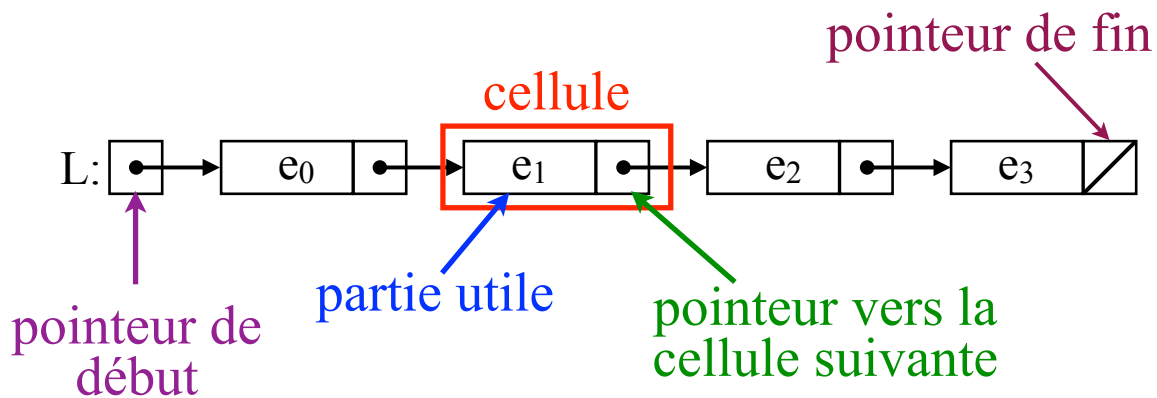
    L = remove(L, 0);

    print(L);

    return 1;
} //end program
```

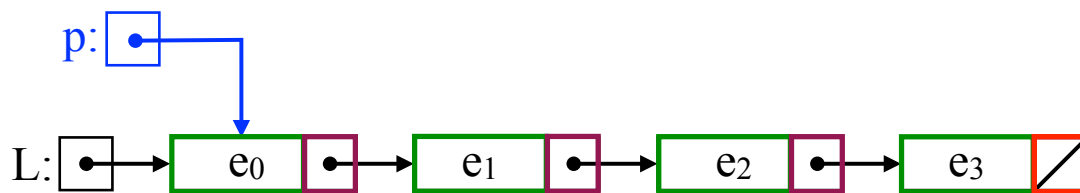
Implem. Pointeurs

- Liste Chaînée



Implem. Pointeurs (2)

- Parcours d'une liste chaînée



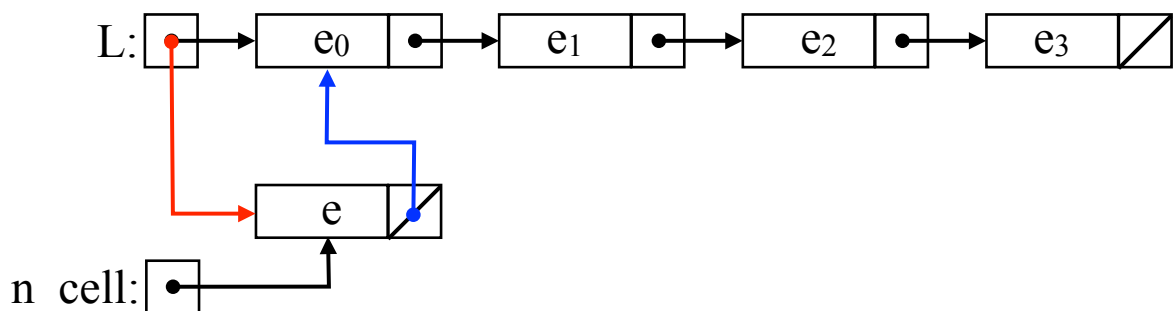
1. Utiliser un pointeur temporaire p
2. Effectuer le traitement sur la partie utile de la cellule courante
3. A l'aide de la partie liante de la cellule courante, passer à la cellule suivante
4. On s'arrête quand p vaut NULL

Implem. Pointeurs (3)

- Même principe pour `get()`, `set()`
 - il suffit de s'arrêter sur la bonne cellule
- Même principe pour `contains()`
 - on s'arrête quand on a trouvé la bonne cellule
 - ✓ l'élément se trouve dans la liste
 - on s'arrête quand on a atteint la fin de la liste
 - ✓ l'élément ne se trouve pas dans la liste

Implem. Pointeurs (4)

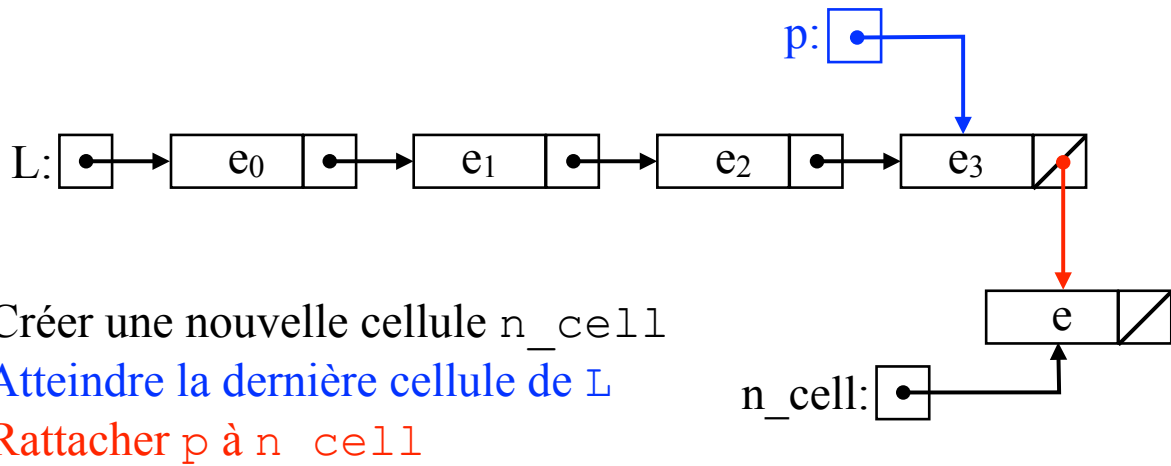
- Fonctionnement de `add_first(L, e)`



1. Créer une nouvelle cellule `n_cell`
2. Rattacher `n_cell` à la 1^{ère} cellule de `L`
3. Rattacher `L` à `n_cell`

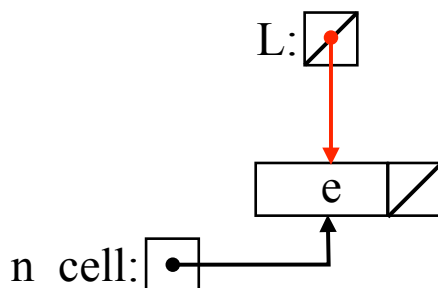
Implem. Pointeurs (3)

- Fonctionnement de `add_last(L, e)`
 - cas général



Implem. Pointeurs (4)

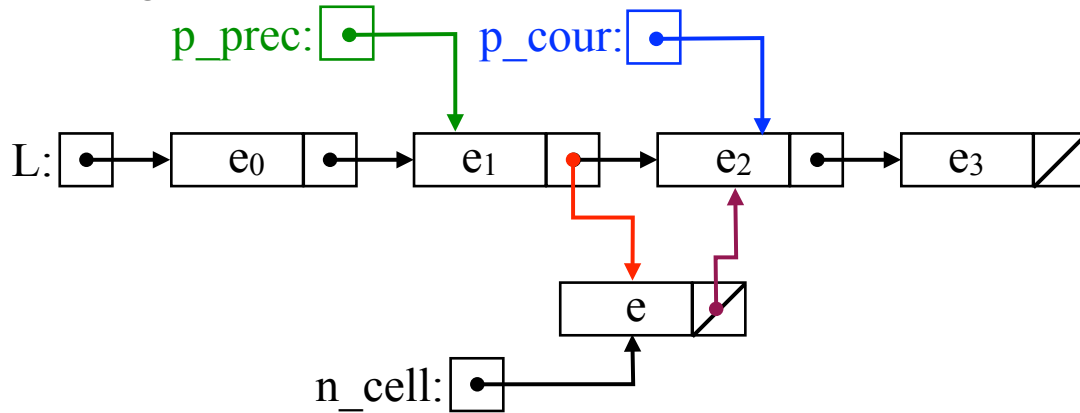
- Fonctionnement de `add_last(L, e)`
 - cas liste vide



1. Créer une nouvelle cellule `n_cell`
2. Rattacher `L` à `n_cell`

Implem. Pointeurs (5)

- Fonctionnement de `add_at(L, 2, e)`
 - cas général



1. Créer une nouvelle cellule `n_cell`
2. Arriver au bon endroit avec `p_cour` et `p_prec`
3. Rattacher `n_cell` à `p_cour`
4. Rattacher `p_prec` à `n_cell`

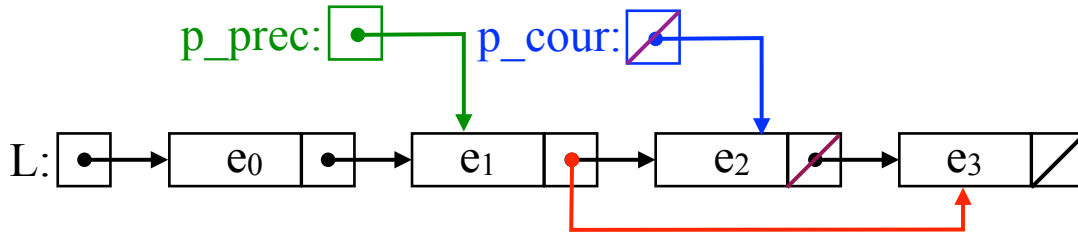
Implem. Pointeur (6)

- Fonctionnement de `add_at(L, 0, e)`
 - cas particulier
 - équivalent à `add_first(L, e)`

Implem. Pointeurs (7)

- Fonctionnement de `remove()`

- cas général: `remove(L, 2)`



1. Arriver au bon endroit avec `p_cour` et `p_prec`
2. Isoler `p_cour`
 - a. relier `p_prec` à `p_cour->next`
 - b. isoler `p_cour`
3. Détruire `p_cour`

Implem. Pointeurs (8)

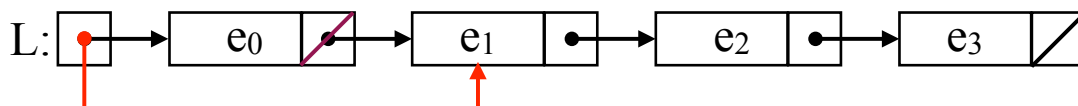
- Fonctionnement de `remove()`

- cas particulier: `remove(L, 0)`

- Technique +/- identique au cas général

- `p_prec` n'est pas utilisable
- `p_prec` doit être remplacé par `L`

- Dans le code, il faudra donc distinguer le cas général du cas particulier



Implem. Pointeurs (11)

- Fichier `linked_list.c`

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

#include "list.h"

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

typedef struct list_t cell;
```

partie utile
partie liante

Implem. Pointeurs (12)

```
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;

    return n_cell;
} //end create_cell()

List *empty_list(){
    return NULL;
} //end empty_list()

Boolean is_empty(List *l){
    return l==NULL;
} //end is_empty()
```

Implem. Pointeurs (13)

- L'implémentation des autres fonctionnalités est loin d'être triviale
- Nécessite l'application (rigoureuse) de l'approche constructive

Complexité

- Analyse de complexité

Fonction	Tableau	Pointeur
empty_list	$O(1)$	$O(1)$
is_empty	$O(1)$	$O(1)$
length	$O(1)$	$O(\text{length}(L))$
get	$O(1)$	$O(\text{length}(L))$
set	$O(1)$	$O(\text{length}(L))$
add_at	$O(\text{length}(L))$	$O(\text{length}(L))$
add_first	$O(\text{length}(L))$	$O(1)$
add_last	$O(\text{length}(L))$	$O(\text{length}(L))$
remove	$O(\text{length}(L))$	$O(\text{length}(L))$
contains	$O(\text{length}(L))$	$O(\text{length}(L))$

Complexité (2)

- Le tableau précédent donne la complexité temporelle
- Quid de la complexité *spatiale*?
 - l'implémentation par tableau implique un certain gaspillage d'espace mémoire
 - l'implémentation par pointeur n'utilise que ce qui est nécessaire
 - ✓ mais coût lié à la partie liante
- En pratique, on préférera toujours une implémentation par pointeur

Agenda

- Chapitre 6: Listes
 - Principe
 - Spécification Abstraite
 - Implémentation
 - Algorithmique
 - ✓ Notations
 - ✓ `length()`
 - ✓ `get()`
 - ✓ `set()`
 - ✓ `add_first()`
 - ✓ `add_last()`
 - ✓ `add_at()`
 - ✓ `remove()`
 - ✓ `contains()`
 - Bestiaire

Notations

- La liste (e_0, e_1, e_2, e_3) est représentée comme suit:

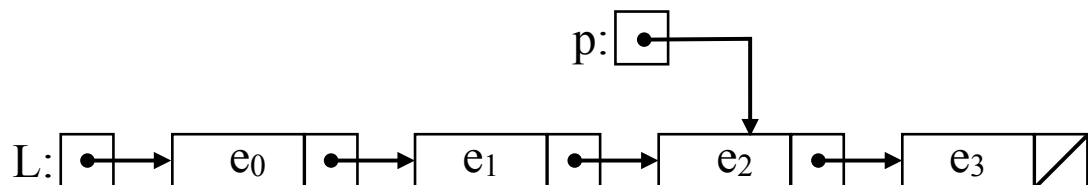


- Notation

- $L^+ = (e_0, e_1, e_2, e_3)$

Notations (2)

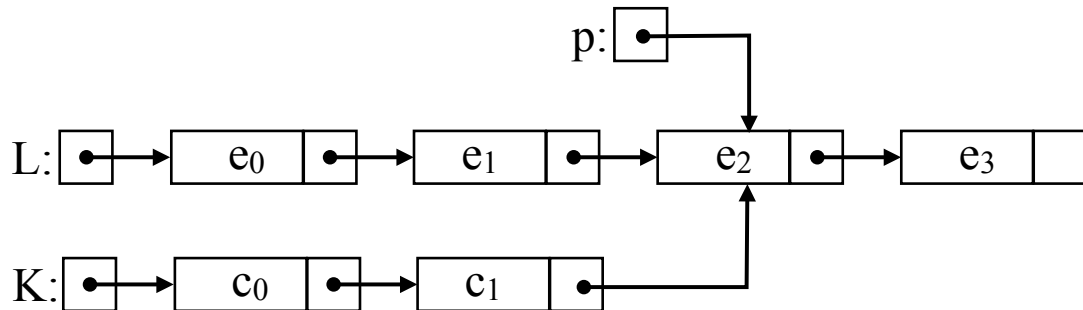
- Notation pour $L^+ = (e_0, e_1, e_2, e_3)$
 - $p^+(e_1, e_2)$ est une sous-liste de L^+
 - ✓ (e_2, e_4) n'en est pas une
- Soient une liste L^+ et une sous-liste p^+ de L^+ :



- on peut définir les notations suivantes
 - $p^+ = (e_2, e_3)$
 - $p^-L = (e_0, e_1)$
 - ✓ si pas d'ambiguïté, on peut noter p^-
 - par convention, $L^+ = p^- || p^+$

Notations (3)

- Exemple d'ambiguïté



- p^- est ambiguë car on ne sait pas si c'est
 - $p^-_L = (e_0, e_1)$
 - $p^-_K = (c_0, c_1)$

Notations (4)

- Remarque
 - si $L^+ = (e_0, e_1, e_2, e_3)$
 - ✓ $L^- = ()$
 - si $L = \text{empty_list}()$
 - ✓ $L^+ = ()$

```
length()
```

- Soit la liste suivante:



- Elle contient 4 cellules
 - sa longueur est de 4
- Notation
 - $long(L^+) = 4$
- De manière générale, $long(L)$ donne le nombre de cellules de L

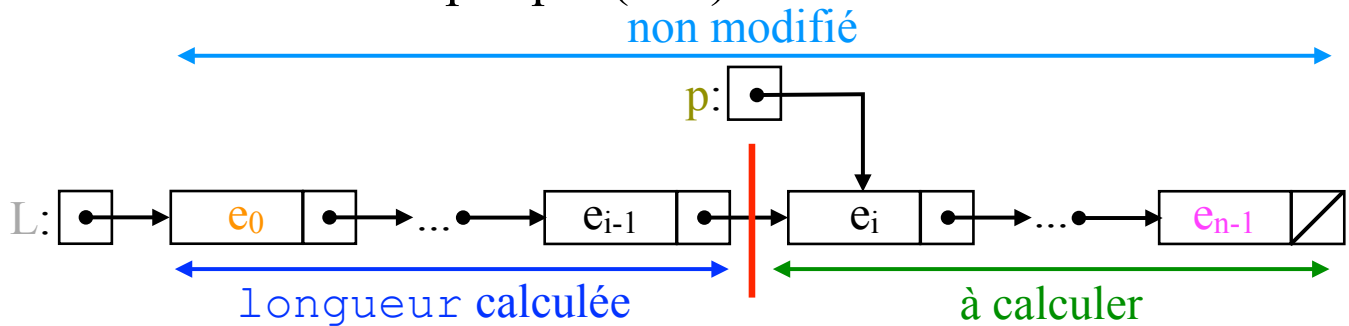
length () (2)

- Définition du problème
 - Input
 - ✓ L , tête de liste
 - Output
 - ✓ $long(L)$
 - Objet Utilisé
 - ✓ L est de type `List` *
 - peut être vide
 - dans ce cas, la $long(L) = 0$
- Spécification

```
/*
 * @pre:  $\emptyset$ 
 * @post:  $L = L_0 \wedge \text{length} = \text{long}(L_0)$ 
 */
int length(List *L);
```

length () (3)

- Invariant Graphique (GLI)



- Invariant Formel (FLI)

$$L = L_0$$

\wedge

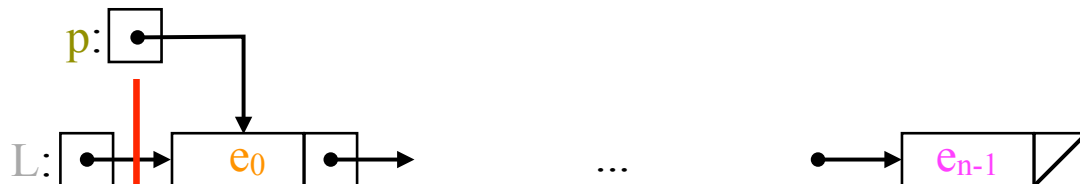
$$\text{longueur} = \text{long}(p^-)$$

Légende:
 Règle 1
 Règle 2
 Règle 3
 Règle 4
 Règle 5
 Règle 6

length () (4)

- $\{\text{Pré}\} \text{ INIT } \{\text{Inv}\}$

- situation à établir



Par définition, $p^- = ()$
 $\Rightarrow \text{long}(p^-) = 0$

length () (5)

- Instructions

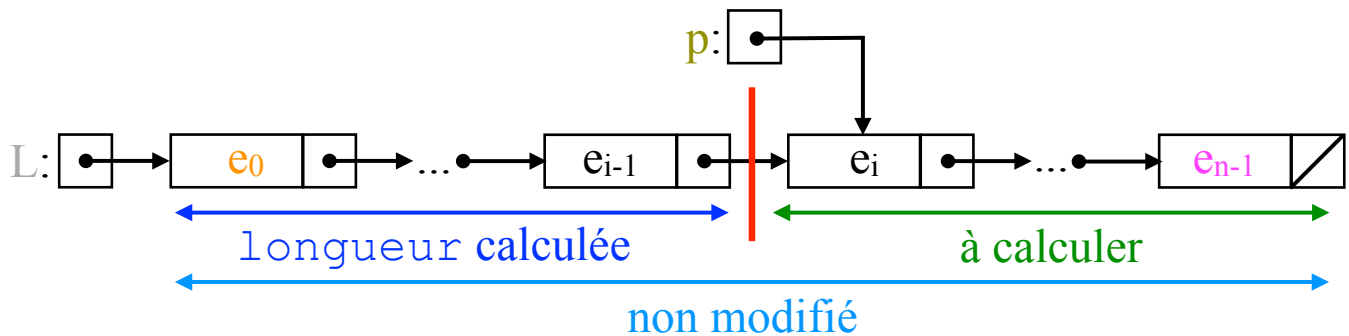
```
{Pré  $\equiv \emptyset$ }  
List *p = L;  
{L = L0  $\wedge$  p = L}  
int longueur = 0;  
{Inv  $\equiv$  L = L0  $\wedge$  longueur = long(p-)}
```

length () (6)

- Critère d'Arrêt ($\neg B$)
 - $p^+ = ()$
 - `p == NULL`
- {Inv $\wedge \neg B$ } END {Post}
 - rien à faire, hormis retourner la valeur de longueur
 - {Inv $\wedge \neg B$ } \Rightarrow {Post}
 - ✓ {L = L₀ \wedge longueur = long(p⁻) \wedge p⁺ = ()}
 - ✓ si p⁺ = (), alors p⁻ indique qu'on a parcouru tout L
 - p⁻ = L⁺
 - ✓ \Rightarrow longueur = long(L₀)

length () (7)

- $\{Inv \wedge B\} \text{ ITER } \{Inv\}$



Il faut établir:

incrémenter longueur

$$\text{long}(p) = \text{long}(p^-) + 1$$

avancer p à la cellule suivante

$$p = p_1 \rightarrow \text{next};$$

length () (8)

- Instructions

```
{Inv  $\equiv$  L = L0  $\wedge$  longueur = long(p-)}  
while(p!=NULL){  
  {Inv  $\wedge$  B  $\equiv$  L = L0  $\wedge$  longueur = long(p-)  $\wedge$  p+  $\neq$  ()}  
  longueur++;  
  {L = L0  $\wedge$  longueur = long(p-)+1  $\wedge$  p+  $\neq$  ()}  
  p = p->next;  
  {Inv  $\equiv$  L = L0  $\wedge$  longueur = long(p-)}  
}  
//fin while - p
```

length () (9)

- Code complet

```
int length(List *L){
    {Pré  $\equiv \emptyset$ }
    List *p = L;
    int longueur = 0;

    {Inv  $\equiv L = L_0 \wedge \text{longueur} = \text{long}(p_-)$ }
    while(p!=NULL){
        longueur++;
        p = p->next;
    }//fin while - p

    return longueur;
    {Post  $\equiv L = L_0 \wedge \text{length} = \text{long}(L_0)$ }
} //fin length()
```

get ()

- Définition du problème
 - Input
 - ✓ L , tête de liste
 - ✓ i , rang de l'élément à retourner
 - Output
 - ✓ e_i , l'élément de rang i dans L
 - Objets Utilisés
 - ✓ L est de type `List *`
 - ne peut être vide
 - ✓ i est de type `int`
 - $0 \leq i < \text{length}(L)$

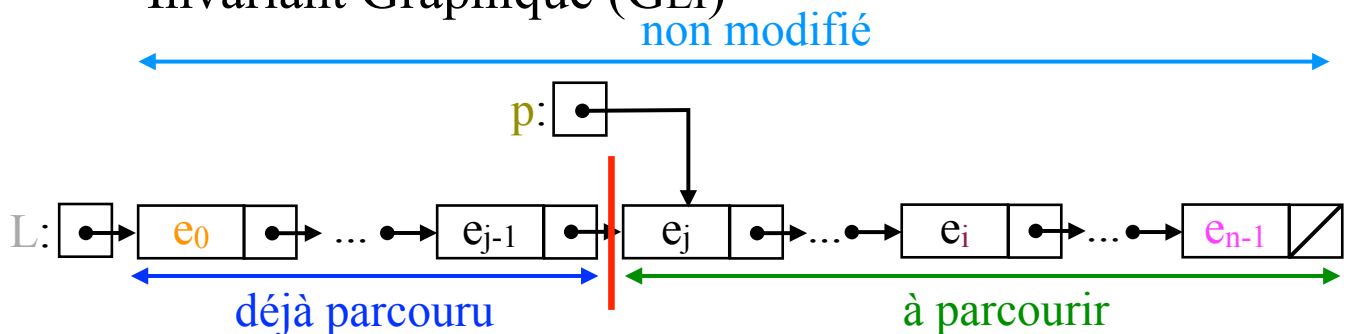
get () (2)

- Spécification

```
/*  
 * @pre:  $L \neq () \wedge 0 \leq i < \text{length}(L)$   
 * @post:  $L = L_0 \wedge i = i_0 \wedge \text{get} = e_i$   
 */  
void *get(List *L, int i);
```

get () (3)

- Invariant Graphique (GLI)



- Invariant Formel (FLI)

$L = L_0 \wedge i = i_0$

\wedge

$0 \leq \text{long}(p) \leq i$

Légende:

Règle 1

Règle 2

Règle 3

Règle 4

Règle 5

Règle 6

get () (4)

- {Pré} INIT {Inv}
 - situation à établir



Par définition, $p^- = ()$

$\Rightarrow \text{long}(p^-) = 0$

déclarer un indice j à initialiser à 0

get () (5)

- Instructions

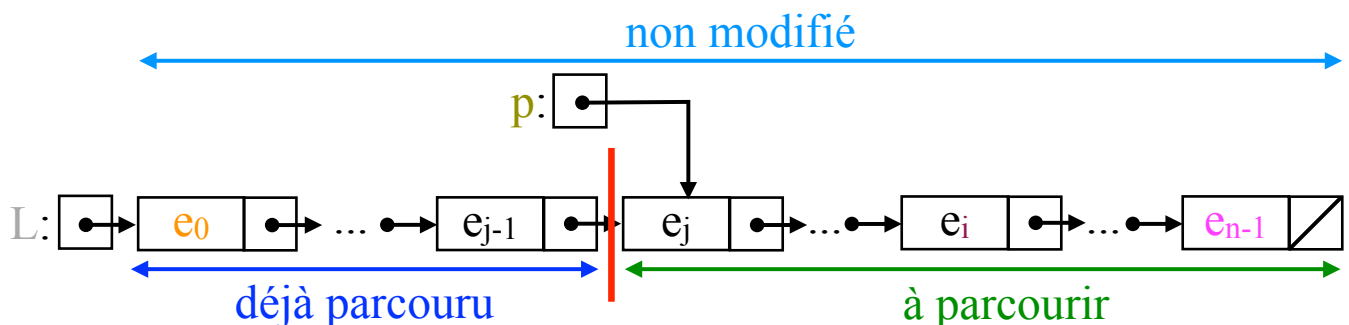
```
{Pré  $\equiv$  L initialisée  $\wedge 0 \leq i < \text{length}(L)$ }  
List *p = L;  
{L = L0  $\wedge$  i = i0  $\wedge$  p = L}  
int j = 0;  
{Inv  $\equiv$  L = L0  $\wedge$  i = i0  $\wedge 0 \leq \text{long}(p^-) \leq i$ }
```

get () (6)

- Critère d'Arrêt ($\neg B$)
 - $j == i$
- $\{Inv \wedge \neg B\}$ END $\{Post\}$
 - rien à faire, hormis retourner la valeur au rang i
 - $\{Inv \wedge \neg B\} \Rightarrow \{Post\}$
 - ✓ à démontrer en exercice, à la maison

get () (7)

- $\{Inv \wedge B\}$ ITER $\{Inv\}$



Il faut établir:

avancer p à la cellule suivante

$p = p \rightarrow next;$

incrémenter l'indice j

$j = j_1 + 1$

get () (8)

- Instructions

```
{Inv  $\equiv$  L = L0  $\wedge$  i = i0  $\wedge$  0  $\leq$  long(p-)  $\leq$  i}
while(j!=i){
  {Inv  $\wedge$  B  $\equiv$  L = L0  $\wedge$  i = i0  $\wedge$  0  $\leq$  long(p-)  $\leq$  i  $\wedge$  j  $\neq$  i}
  p = p->next;
  {L = L0  $\wedge$  i = i0  $\wedge$  0  $\leq$  long(p-) < i  $\wedge$  j  $\neq$  i}
  j++;
  {Inv  $\equiv$  L = L0  $\wedge$  i = i0  $\wedge$  0  $\leq$  long(p-)  $\leq$  i}
} //fin while
```

get () (9)

- Code complet

```
void *get(List *L, int i){
  {Pré  $\equiv$  L initialisée  $\wedge$  0  $\leq$  i < length(L)}
  List *p = L;
  int j = 0;

  {Inv  $\equiv$  L = L0  $\wedge$  i = i0  $\wedge$  0  $\leq$  long(p-)  $\leq$  i}
  while(j!=i){
    p = p->next;
    j++;
  } //fin while

  return p->data;
  {Post  $\equiv$  L = L0  $\wedge$  i = i0  $\wedge$  get = ei}
} //fin get()
```

set ()

- Grosso modo le même principe que get ()
- Deux différences
 - expression de la postcondition
 - $\{Inv \wedge \neg B\}$ END $\{Post\}$
 - ✓ il y a des instructions à exécuter
 - ✓ ne pas oublier de libérer la mémoire occupée par l'ancien e_i
- A faire à la maison, à titre d'exercice

add_first ()

- Définition du problème
 - Input
 - ✓ L , tête de liste
 - ✓ e , la donnée à ajouter
 - Output
 - ✓ L avec e en début de liste, si possible
- Objets Utilisés
 - L de type `List` *
 - ✓ peut être éventuellement vide
 - e de type `void` *
 - ✓ ne peut être `NULL`

add_first() (2)

- Analyse
 - SP₁: créer une nouvelle cellule
 - SP₂: rattacher la cellule à L, si possible
- Enchaînement des SPs
 - SP₁ → SP₂
- Spécification

```
/*  
 * @pre: e ≠ NULL  
 * @post: e = e0 ∧ add_first = [(e) || L0] ∨ L0  
 */  
List *add_first(List *L, void *e);
```

add_first() (3)

- Code complet

```
List *add_first(List *L, void *e){  
    {Pré ≡ e ≠ NULL}  
    List *n_cell = create_cell(e);  
    if(n_cell==NULL)  
        {e = e0 ∧ L = L0 ∧ n_cell = ()}  
        return L;  
        {e = e0 ∧ L = n_cell || L0}  
        {Post ≡ e = e0 ∧ add_first = L0}  
  
    {e = e0 ∧ n_cell = (e) ∧ L = L0}  
    n_cell->next = L;  
    {e = e0 ∧ n_cell = (e) || L0}  
  
    return n_cell;  
    {Post ≡ e = e0 ∧ add_first = ((e) || L0)}  
} //fin add_first()
```

SP₁

SP₂

add_last()

- Définition du problème
 - Input
 - ✓ `L`, tête de liste
 - ✓ `e`, la donnée à ajouter
 - Output
 - ✓ `L` avec `e` en fin de liste, si possible
 - Objets Utilisés
 - ✓ `L` est de type `List` *
 - peut être éventuellement vide
 - ✓ `e` est de type `void` *
 - ne peut être `NULL`

add_last() (2)

- Analyse du problème
 - `SP1`: créer une nouvelle cellule
 - ✓ identique au `SP1` de `add_first()`
 - `SP2`: `L` est vide
 - `SP3`: `L` n'est pas vide
- `SP3` peut être subdivisé en 2 autres SPs
 - `SP3.a`: parcourir `L` jusqu'à la dernière cellule
 - `SP3.b`: rattacher la nouvelle cellule, si possible, à `L`
- Enchaînement des SPs
 - `SP1` → `SP2`
→ (`SP3.a` → `SP3.b`)

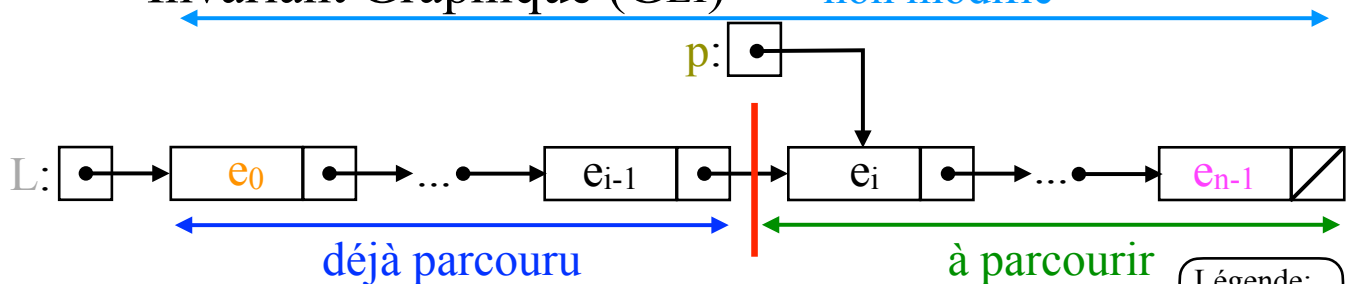
add_last() (3)

- Spécification

```
/*  
 * @pre: e ≠ NULL  
 * @post: e = e0 ∧ add_last = [ (L0 || (e)) ∨ L0 ]  
 */  
List *add_last(List *L, void *e);
```

add_last() (4)

- Le SP_{3,a} nécessite un parcours de liste
- Invariant Graphique (GLI) non modifié



- Invariant Formel (FLI)

$$L = L_0 \wedge e = e_0$$

\wedge

$$0 \leq \text{long}(p^-) \leq \text{long}(L) - 1$$

\wedge

$$\text{long}(p^+) \geq 1$$

Légende:
Règle 1
Règle 2
Règle 3
Règle 4
Règle 5
Règle 6

add_last() (5)

- Code pour SP₂

```
List *add_last(List *L, void *e){
    {Pré ≡ e ≠ NULL}
    //SP1
    List *n_cell = create_cell(e);
    if(n_cell==NULL)
        return L;
    {e ≠ NULL ∧ n_cell = (e) ∧ e = e0}
    {n_cell = (e) ∧ e = e0}
    if(is_empty(L))
        {n_cell = (e) ∧ L = () ∧ e = e0}
        L = n_cell;
        {L = (e) ∧ e = e0}
        {L = L0 || (e) ∧ e = e0}
    else
        //traitement SP3
        return L;

    {Post ≡ e = e0 ∧ add_last = [(L0 || (e)) ∨ L0]}
} //fin add_last()
```

add_last() (6)

```
List *add_last(List *L, void *e){
    {Pré ≡ e ≠ NULL}
    //traitement SP1
    if(is_empty(L))
        //traitement SP2
    else{
        {n_cell = (e) ∧ e = e0}
        List *p = L;
        {Inv ≡ e = e0 ∧ 0 ≤ long(p-) ≤ long(L)-1 ∧ long(p+) ≥ 1 ∧ L = L0}
        while(p->next != NULL)
            {Inv ∧ B ≡ e = e0 ∧ 0 ≤ long(p-) ≤ long(L)-1 ∧ long(p+) ≥ 2
             ∧ L = L0}
            p = p->next;
            {Inv ≡ e = e0 ∧ 0 ≤ long(p-) ≤ long(L)-1 ∧ long(p+) ≥ 1 ∧ L
             = L0}
            {e = e0 ∧ n_cell = (e) ∧ L = L0}
            p->next = n_cell;
            {e = e0 ∧ L = L0 || (e)}
        return L;

        {Post ≡ e = e0 ∧ add_last = [(L0 || (e)) ∨ L0]}
    } //fin add_last()
```

add_last() (7)

- Code complet

```
List *add_last(List *L, void *e){
    {Pré ≡ e ≠ NULL}
    List *n_cell = create_cell(e);
    if(n_cell==NULL)
        return L;
    if(is_empty(L))
        L = n_cell;
    else{
        List *p = L;
        {Inv ≡ e = e0 ∧ 0 ≤ long(p-) ≤ long(L)-1 ∧ long(p+) ≥ 1 ∧ L = L0}
        while(p->next != NULL)
            p = p->next;
        p->next = n_cell;
    }
    return L;

    {Post ≡ e = e0 ∧ add_last = [(L0 || (e)) ∨ L0]}
} //fin add_last()
```

add_at()

- Définition du problème

- Input
 - ✓ L, tête de liste
 - ✓ i, rang où ajouter la donnée
 - ✓ e, la donnée à ajouter
- Output
 - ✓ L avec e ajouté au rang i, si possible
- Objets Utilisés
 - ✓ L est de type List *
 - peut être éventuellement vide
 - ✦ mais dans ce cas, i==0
 - ✓ i est de type int
 - ✓ e est de type void *
 - ne peut être NULL

add_at () (2)

- Analyse du problème
 - SP₁: le rang vaut 0
 - ✓ problème identique à add_first()
 - SP₂: le rang est plus grand ou égal à 1
- SP₂ peut être divisé en 3 SPs
 - SP_{2.a}: se positionner au rang i
 - SP_{2.b}: créer une nouvelle cellule
 - ✓ identique au SP₁ de add_first()
 - SP_{2.c}: rattacher la cellule à la liste
- Enchaînement des SPs
 - SP₁
SP_{2.a} → SP_{2.b} → SP_{2.c}

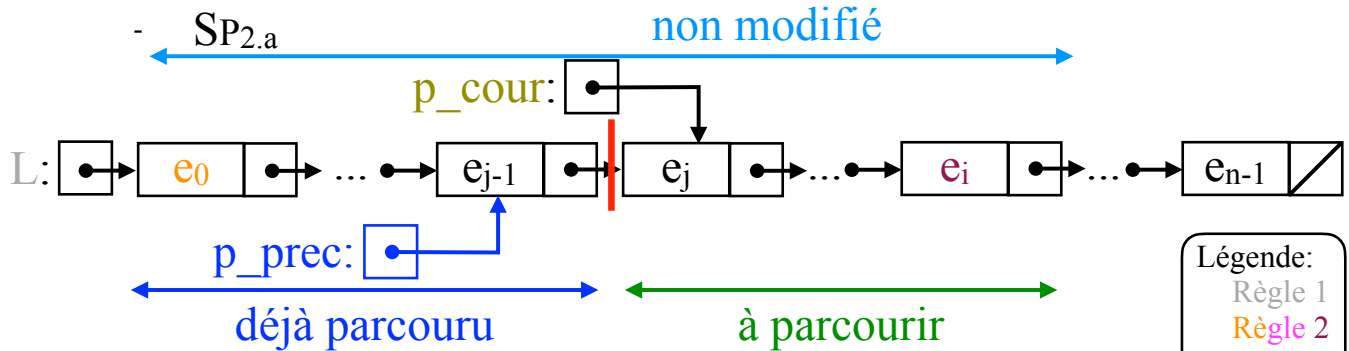
add_at () (3)

- Spécification

```
/*  
 * @pre: e ≠ NULL ∧ 0 ≤ i ≤ length(L)  
 * @post: e = e0 ∧ add_at = [ L0 ∨  
 *          ((e0, ..., ei-1) || e || (ei, ..., en-1),  
 *          n ≥ 1) ]  
 */  
List *add_at(List *L, int i, void *e);
```

add_at () (4)

- Invariant Graphique (GLI)



- Invariant Formel (FLI)

$$e = e_0 \wedge L = L_0$$

\wedge

$$0 \leq j \leq i$$

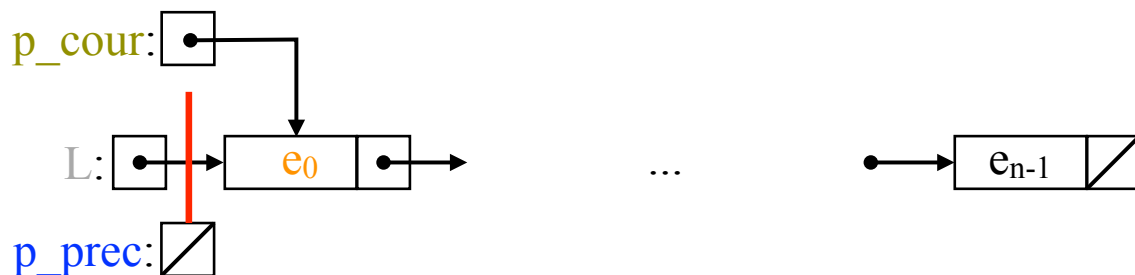
\wedge

$$p_prec = (e_0, \dots, e_{j-2}) \wedge p_cour = p_prec \parallel (e_{j-1})$$

add_at () (5)

- $\{\text{Pré}\} \text{ INIT } \{\text{Inv}\}$

- situation à établir



Il faut établir

p_cour sur la 1^{ère} cellule

p_prec à p_cour

$j = 0$

add_at () (6)

- Instructions

```
{Pré  $\equiv e \neq \text{NULL} \wedge 0 \leq i < \text{length}(L)$ }  
int j = 0;  
{e = e0  $\wedge$  L = L0  $\wedge$  0  $\leq$  i < length(L)  $\wedge$  j = 0}  
{e = e0  $\wedge$  L = L0  $\wedge$  0  $\leq$  j  $\leq$  i}  
List *p_cour = L;  
{e = e0  $\wedge$  L = L0  $\wedge$  0  $\leq$  j  $\leq$  i  $\wedge$  p_cour- = ()}  
List *p_prec = NULL;  
{e = e0  $\wedge$  L = L0  $\wedge$  0  $\leq$  j  $\leq$  i  $\wedge$  p_cour- = ()  $\wedge$  p_prec = ()}  
{Inv  $\equiv e = e_0 \wedge L = L_0 \wedge 0 \leq j \leq i \wedge$  p_prec-=(e0, ..., ej-2)  
   $\wedge$  p_cour- = p_prec- || (ej-1)}
```

add_at () (7)

- Critère d'Arrêt ($\neg B$)
 - j == i
- {Inv \wedge B} ITER {Inv}
 - établir
 - ✓ p_prec doit pointer sur la même cellule que p_cour
 - ✓ p_cour doit avancer d'une cellule
 - ✓ incrémenter j

add_at () (8)

- Instructions

```
{Inv  $\equiv$  e = e0  $\wedge$  L = L0  $\wedge$  0  $\leq$  j  $\leq$  i  $\wedge$  p_prec-=(e0, ..., ej-2)  
     $\wedge$  p_cour- = p_prec- || (ej-1)}  
while(j<i){  
    {Inv  $\wedge$  B  $\equiv$  e = e0  $\wedge$  L = L0  $\wedge$  0  $\leq$  j < i  $\wedge$   
    p_prec-=(e0, ..., ej-2)  $\wedge$  p_cour- = p_prec- || (ej-1)}  
    p_prec = p_cour;  
    {e = e0  $\wedge$  L = L0  $\wedge$  0  $\leq$  j < i  $\wedge$  p_prec-=(e0, ..., ej-1)  $\wedge$   
    p_cour- = p_prec-}  
    p_cour = p_cour->next;  
    {e = e0  $\wedge$  L = L0  $\wedge$  0  $\leq$  j < i  $\wedge$  p_prec-=(e0, ..., ej-1)  $\wedge$   
    p_cour- = p_prec- || (ej)}  
    j++;  
    {Inv  $\equiv$  e = e0  $\wedge$  L = L0  $\wedge$  0  $\leq$  j  $\leq$  i  $\wedge$   
    p_prec-=(e0, ..., ej-2)  $\wedge$  p_cour- = p_prec- || (ej-1)}  
} //fin while - j
```

add_at () (9)

- SP_{2.c}

```
{e = e0  $\wedge$  L = L0  $\wedge$  n_cell = (e)  $\wedge$  p_prec-=(e0, ..., ei-2)  $\wedge$   
p_cour- = p_prec- || (ei-1)}  
n_cell->next = p_cour;  
{e = e0  $\wedge$  n_cell = (e) || p_cour+  $\wedge$  p_prec-=(e0, ..., ei-2)  $\wedge$   
p_cour- = p_prec- || (ei-1)}  
p_prec->next = n_cell;  
{e = e0  $\wedge$  p_prec- || (ei-1) || (e) || p_cour+}  
{e = e0  $\wedge$  L = (e0, ..., ei-1) || (e) || (ei, ..., en-1)}
```


add_at () (10)

```
List *add_at(List *L, int i, void *e){
```

```
    if(i==0) return add_first(L, e);
```

SP₁

```
    int j=0;
```

```
    List *p_cour = L;
```

```
    List *p_prec = NULL;
```

```
    while(j<i){
```

```
        p_prec = p_cour;
```

```
        p_cour = p_cour->next;
```

```
        j++;
```

```
    } //fin while - j
```

SP_{2.a}

```
    List *n_cell = create_cell(e);
```

```
    if(n_cell == NULL) return L;
```

SP_{2.b}

```
    n_cell->next = p_cour;
```

```
    p_prec->next = n_cell;
```

SP_{2.c}

```
    return L;
```

```
} //end add_at()
```

remove ()

- A faire à la maison, à titre d'exercice
 - approche constructive complète exigée

contains ()

- A faire à la maison, à titre d'exercice
 - approche constructive complète exigée

Agenda

- Chapitre 6: Listes
 - Principe
 - Spécification Abstraite
 - Implémentation
 - Algorithmique
 - Bestiaire
 - ✓ Liste avec Cellule d'En-Tête
 - ✓ Liste Doublement Chaînée
 - ✓ Liste Pointeur Début/Fin
 - ✓ Liste Circulaire

Cellule En-Tête

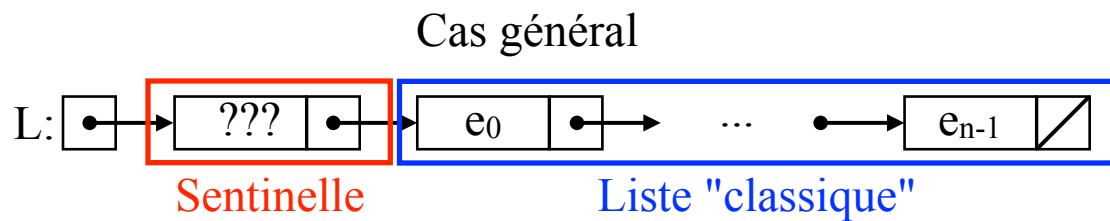
- Ajout d'une cellule en tête de liste
 - $L \rightarrow \text{next}$ pointe sur la 1^{ère} cellule de la liste
 - $L \rightarrow \text{data}$ ne contient rien
- **Sentinelle**
- Toutes les fonctions prennent en argument un pointeur vers la sentinelle
 - et non plus la tête de liste

Cellule En-Tête (2)

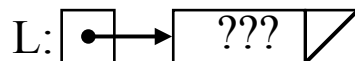
- **Avantage**
 - simplification de la gestion des cas limites
 - ✓ cas où la liste est vide
 - la liste contient toujours au moins une cellule
 - ✓ cas où le premier élément de la liste est modifié
 - $L \rightarrow \text{next}$ est modifié
 - ✓ cas où le premier élément de la liste est supprimé
 - pas besoin de retourner un pointeur vers la tête de liste
- **Notation**
 - $L^+ = s \parallel (e_0, e_1, \dots, e_{n-1})$

Cellule En-Tête (3)

- Implémentation



Liste Vide



Cellule En-Tête (4)

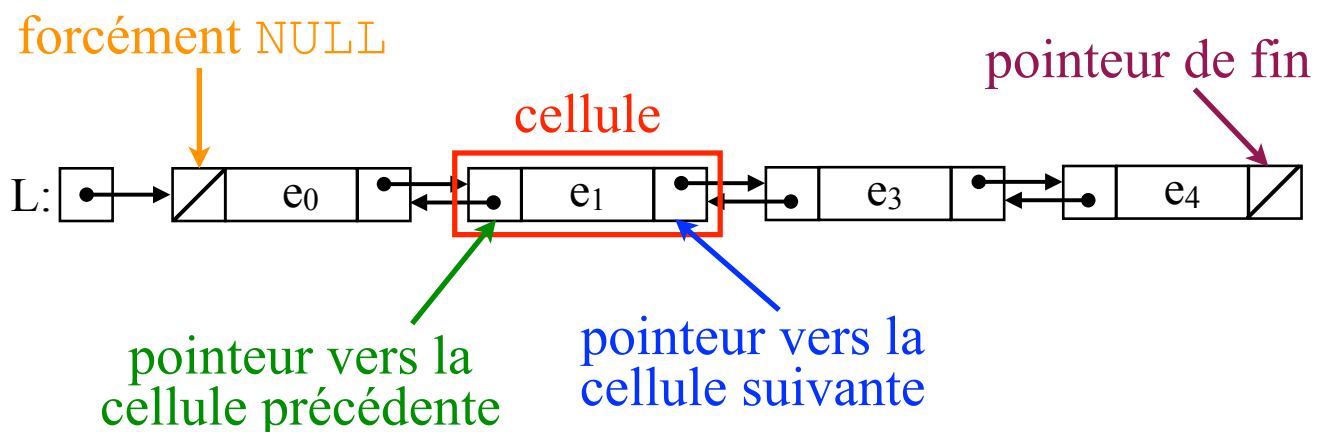
- Structure de données
 - identique à une liste chaînée classique
 - ce sont les processus de création et de manipulation qui diffèrent

Doublement Chaînée

- Une liste simplement chaînée ne permet pas de revenir en arrière
 - il est parfois intéressant de pouvoir bouger librement dans la liste
- Solution:
 - **liste doublement chaînée**
- Une liste doublement chaînée contient deux parties liantes
 - un pointeur vers la cellule suivante
 - un pointeur vers la cellule précédente
- L'accès est toujours séquentiel mais on peut avancer/reculer dans la liste

Doublement Chaînée (2)

- Implémentation



Doublement Chaînée (3)

- Implémentation de la structure de données
 - vision opaque

```
typedef struct dl_cell DLList;
```

- Implémentation de la structure de données
 - vision concrète

```
struct dl_cell{  
    struct dl_cell *prev;  
    void *data;  
    struct dl_cell *next;  
};
```

Doublement Chaînée (4)

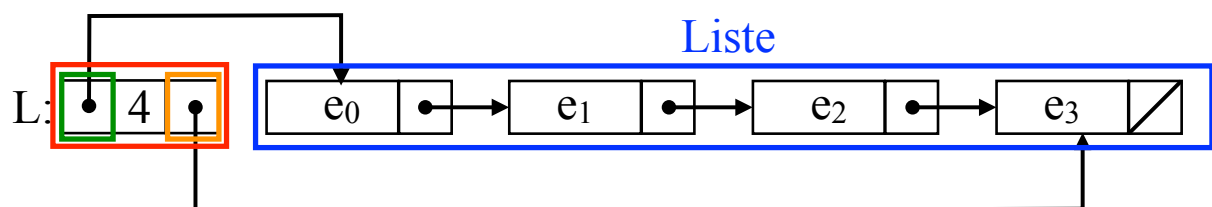
- Le parcours d'une liste doublement chaînée n'est pas différent d'une liste chaînée
 - si on avance (i.e., gauche à droite)
 - ✓ les algorithmes présentés précédemment sont identiques
 - si on recule (i.e., droite à gauche)
 - ✓ on applique le même principe qu'une liste chaînée mais sur le champ `prev`
- A faire à titre d'exercices

Ptr Début/Fin

- Une liste simplement (ou doublement) chaînée ne permet pas d'accéder directement à la fin de la liste
 - nécessité de parcourir toute la liste
 - peu efficace
- Solution?
 - liste chaînée (simplement ou doublement) avec pointeur de début et de fin
- Idée
 - disposer d'une structure d'en-tête contenant
 - ✓ un pointeur de début
 - ✓ un pointeur de fin
 - ✓ des informations additionnelles
 - e.g., taille de la liste

Pointeur Début/Fin (2)

- Implémentation



En-tête

Pointeur début liste

Pointeur fin liste

Pointeur Début/Fin (3)

- Implémentation de la structure de données
 - vision opaque

```
typedef struct header_cell HeaderList;
```

- Implémentation de la structure de données
 - vision concrète
 - se base sur une liste chaînée
 - ✓ simple ou double

```
struct header_cell{  
    List *head;  
    List *tail;  
    unsigned int length;  
};
```

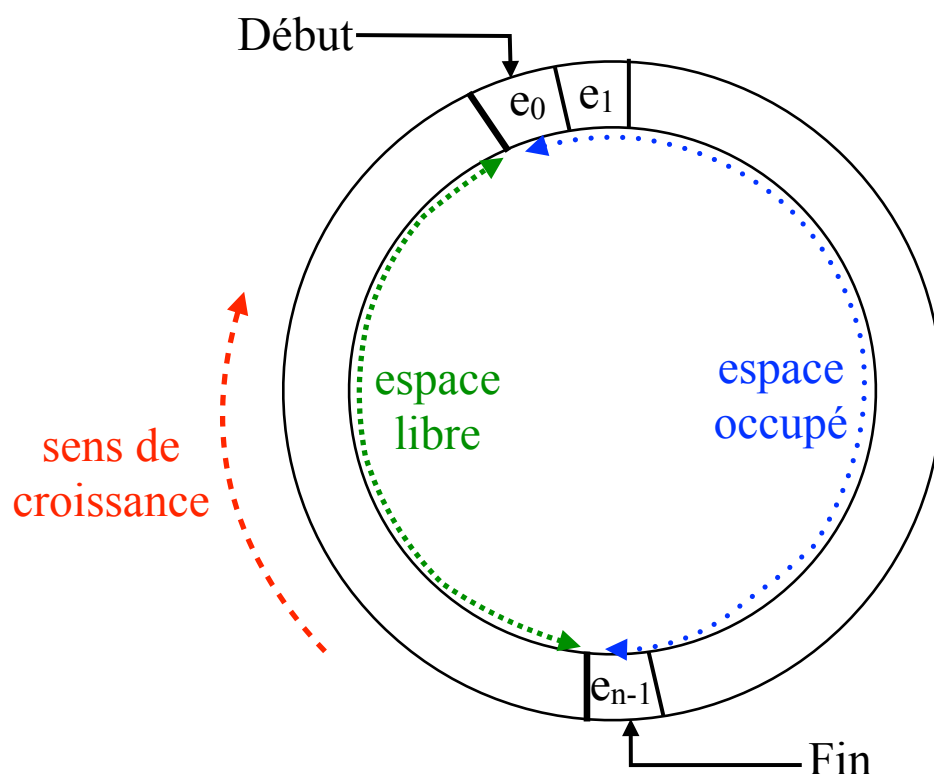
Pointeur Début/Fin (4)

- Le parcours de cette structure est identique à une liste
 - simplement chaînée
 - doublement chaînée
- Ajout
 - en début de structure
 - ✓ identique à `add_first()`
 - en milieu de structure
 - ✓ identique à `add_at()`
 - en fin de structure
 - ✓ facilité par le pointeur de fin
 - dans les 3 cas, il faut mettre à jour le champ `length`

Liste Circulaire

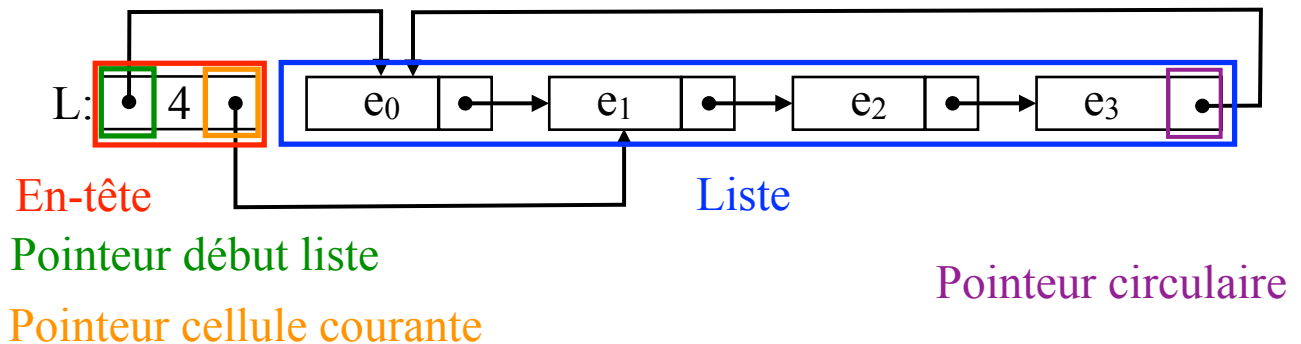
- Dans certaines circonstances, il peut être utile de développer une structure de données
 - qui permette de stocker dynamiquement de l'information
 - sans pour autant accroître "indéfiniment" l'espace mémoire utilisé
- On stocke les données de manière ordonnée
 - on commence au début jusqu'à arriver au maximum
 - une fois le maximum atteint, on recommence au début
 - ✓ on écrase donc la 1^{ère} donnée insérée
- On a donc une structure **circulaire**

Liste Circulaire (2)



Liste Circulaire (3)

- Implémentation



Liste Circulaire (4)

- Implémentation de la structure de données
 - vision opaque

```
typedef struct circular_cell CircularList;
```

- Implémentation de la structure de données
 - vision concrète
 - identique à une liste avec pointeur début/fin

```
struct circular_cell{  
    List *head;  
    List *current;  
    unsigned int length;  
};
```

Liste Circulaire (5)

- L'ajout d'un élément doit se faire avec attention
 - tant qu'on n'a pas atteint le maximum et que la liste n'a jamais été remplie complètement
 - ✓ il faut créer une nouvelle cellule
 - ✓ mettre à jour le pointeur circulaire
 - ✓ mettre à jour le pointeur de fin
 - tant qu'on n'a pas encore atteint le maximum et que la liste a déjà été remplie au moins une fois
 - ✓ pas de création de cellule
 - ✓ mettre à jour le pointeur de fin
 - si on a atteint le maximum
 - ✓ il faut faire "demi-tour"
 - ✓ stocker la donnée dans la 1^{ère} cellule