Compléments de Programmation

Benoit Donnet

Année Académique 2024 - 2025

Agenda

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

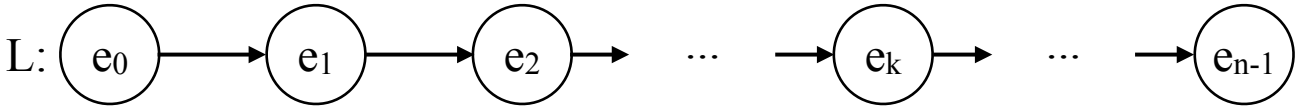
Vocabulaire

- 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
  - $L = (e_0, e_1, ..., e_i, ..., e_{n-1})$
  - e_i désigne l'élément de rang *i*

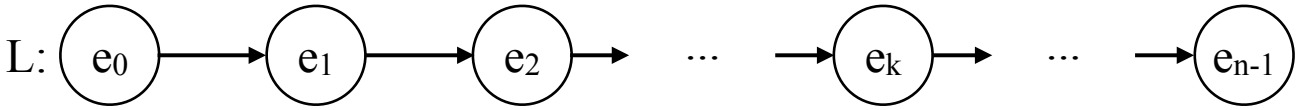
Opérations



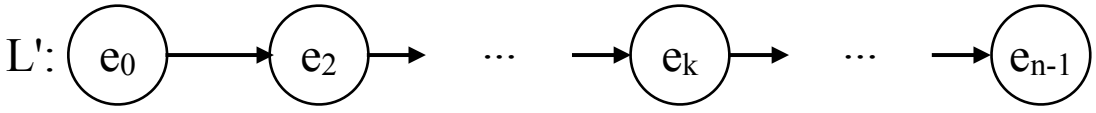
`length(L)` → n

`get(L, 2)` → e_2

Opérations (2)

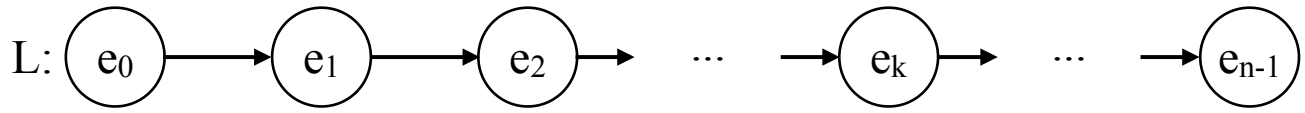


`remove(L, 1)` → L'

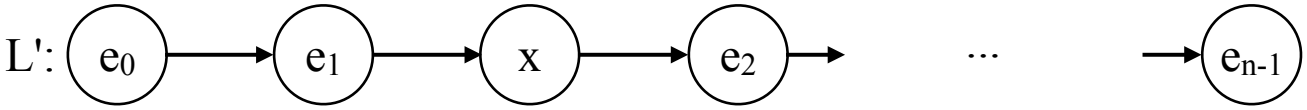


`length(L')` → n-1

Opérations (3)



`add_at(L, 2, x)` → L'



`get(L', 2)` → x
`length(L')` → n+1

Agenda

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

Spécification Abstraite

- Syntaxe

Type:
  List
Utilise:
  Boolean, Element, Integer
Opérations:
  empty_list: → List
  is_empty: List → Boolean
  length: List → Integer
  get: List × Integer → Element
  set: List × Integer × Element → List
  add_at: List × Integer × Element → List
  add_first: List × Element → List
  add_last: List × Element → List
  remove: List × Integer → List
  contains: List × Element → Boolean

Spécifications Abstraites (2)

- Sémantique
  - préconditions

Préconditions:
  $\forall L \in$ List, $\forall e \in$ Element, $\forall i \in$ Integer
  $\forall i, 0 \leq i \leq \texttt{length}(L)$, $\texttt{add\_at}(L, i, e)$
  $\forall i, 0 \leq i < \texttt{length}(L)$, $\texttt{remove}(L, i)$
  $\forall i, 0 \leq i < \texttt{length}(L)$, $\texttt{get}(L, i)$
  $\forall i, 0 \leq i < \texttt{length}(L)$, $\texttt{set}(L, i)$

Spécifications Abstraites (3)

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

- 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:

$\forall L \in$ List, $\forall e \in$ Element, $\forall i \in$ Integer

```
add_first(L, e) = add_at(L, 0, e)
add_last(L, e) = add_at(L, length(L), e)
```

Spécifications Abstraites (5)

Axiomes :

$\forall L \in$ List, $\forall e \in$ Element, $\forall i \in$ Integer

```
length(empty_list) = 0
length(remove(L, i)) = length(L) - 1
length(add_at(L, i, e)) = length(L) + 1
length(set(L, i, e)) = length(L)
```

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

Spécifications Abstraites (6)

Axiomes :

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

```
is_empty(empty_list) = True
is_empty(add_at(L, i, e)) = False
is_empty(set(L, i, e)) = False
```

$$\text{is\_empty}(\text{remove}(L, i)) = \begin{cases} \text{False} & \text{If Length}(L) > 1 \\ \text{True} & \text{If Length}(L) = 1 \end{cases}$$

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

Spécifications Abstraites (7)

Axiomes :

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

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

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

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

Spécifications Abstraites (8)

Axiomes :

$\forall$ L ∈ List, $\forall$ e, f ∈ Element, $\forall$ i, j ∈ Integer

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

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

Spécifications Abstraites (9)

Axiomes :

∀ L ∈ List, ∀ e, f ∈ Element, ∀ i, j ∈ Integer

contains(empty_list, e) = False

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

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

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

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

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);
```

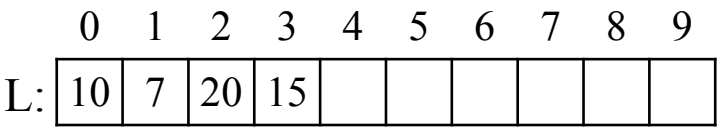
Interface (3)

```
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
```

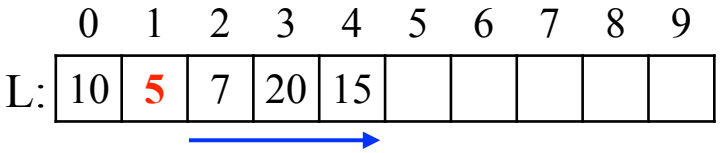

Implem. Tableau

- 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)

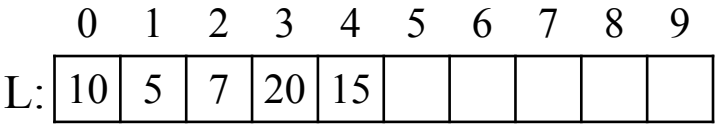



```
L = add_at(L, 1, 5);
```

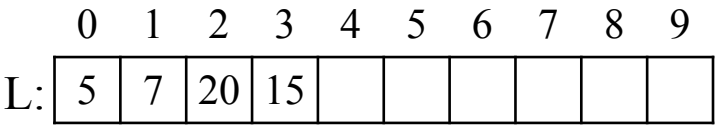


décalage vers la droite

Implem. Tableau (3)




```
L = remove(L, 0);
```



décalage vers la gauche

Implem. Tableau (4)

- Fichier `array_list.c`

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

#include "list.h"

#define INITIAL_SIZE 10
#define REALLOCATION_FACTOR 2

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

taille initiale du tableau

facteur multiplicateur

taille actuelle du tableau

nombre d'élément dans la liste

tableau générique

Implem. Tableau (5)

```c
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 = L0 ∧ 0 ≤ i ≤ length(L) ∧ e = e0
       ∧ ∀ j, 0 ≤ j ≤ i-1, e ∉ L->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`

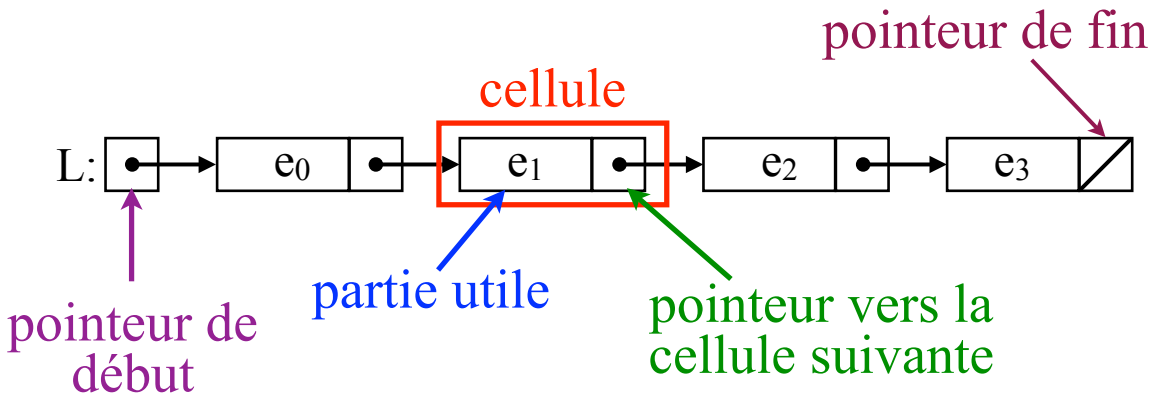
```
#include <assert.h>
#include "list.h"
#include "element.h"

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
```

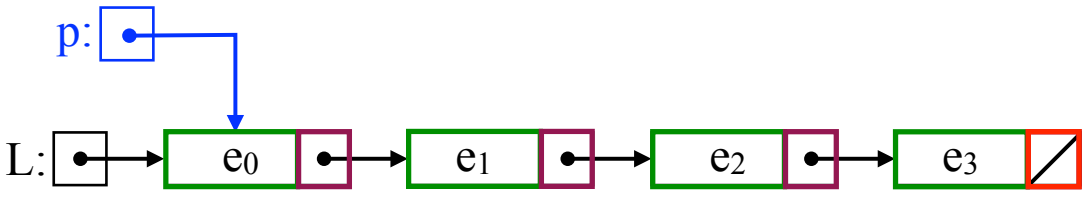
Implem. Pointeurs

- **Liste Chaînée**



Implem. Pointeurs (2)

- Parcours d'une liste chaînée



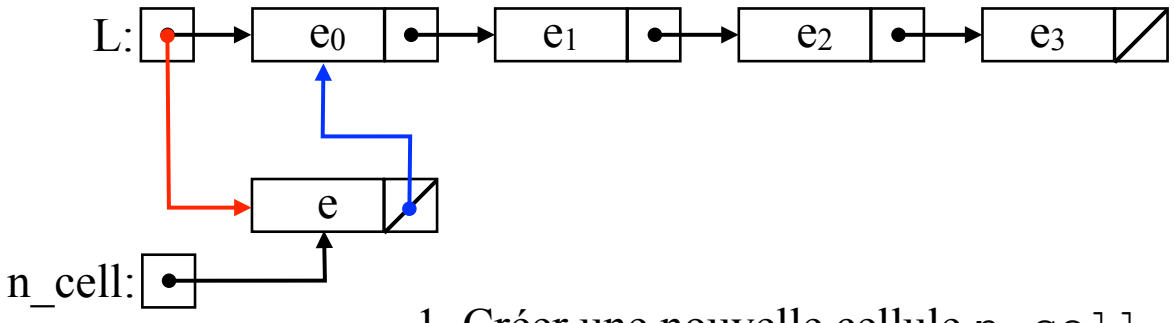
1. Utiliser un pointeur temporaire `p`
2. Effectuer le traitement sur la partie utile de la cellule courante
3. A l'aide de la partie liante de la cellule courante, passer à la cellule suivante
4. On s'arrête quand `p` vaut `NULL`

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)

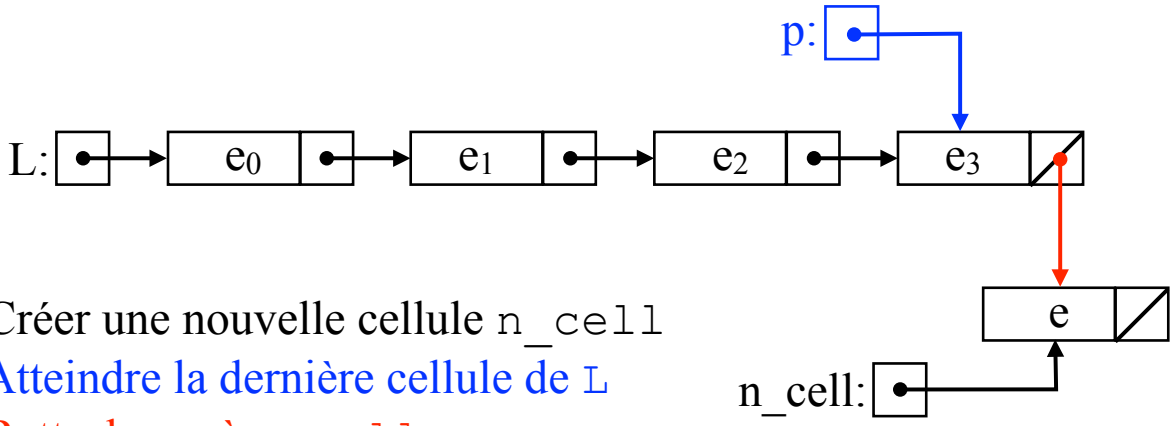
- Fonctionnement de `add_first(L, e)`



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

Implem. Pointeurs (3)

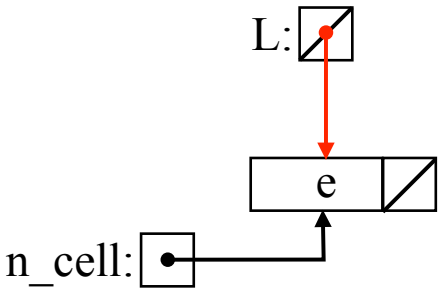
- Fonctionnement de `add_last(L, e)`
  - cas général



1. Créer une nouvelle cellule `n_cell`
2. Atteindre la dernière cellule de `L`
3. Rattacher `p` à `n_cell`

Implem. Pointeurs (4)

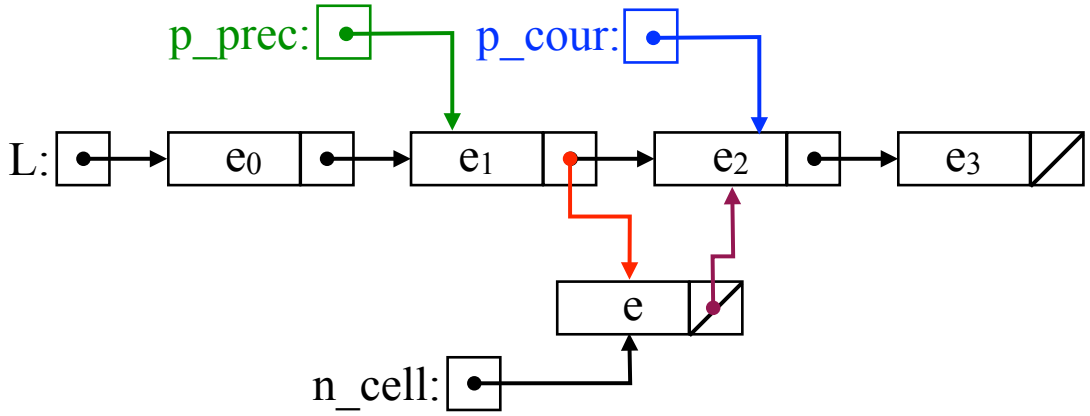
- Fonctionnement de `add_last(L, e)`
  - cas liste vide



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



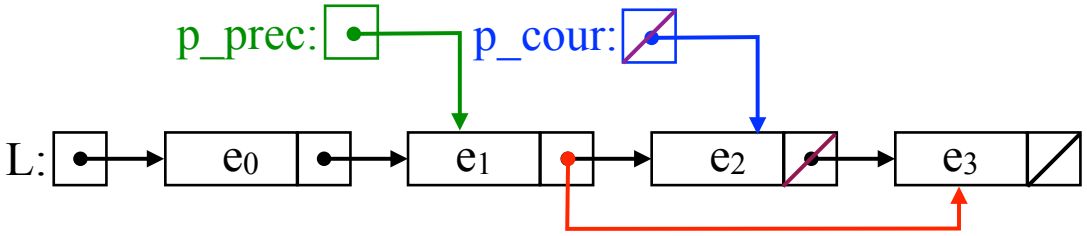
1. Créer une nouvelle cellule `n_cell`
2. Arriver au bon endroit avec `p_cour` et `p_prec`
3. Rattacher `n_cell` à `p_cour`
4. Rattacher `p_prec` à `n_cell`

Implem. Pointeur (6)

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

Implem. Pointeurs (7)

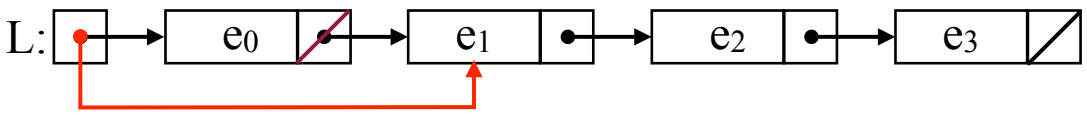
- Fonctionnement de `remove()`
  - cas général: `remove(L, 2)`



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`

```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)

```c
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(length(L))*
`get`	*O(1)*	*O(length(L))*
`set`	*O(1)*	*O(length(L))*
`add_at`	*O(length(L))*	*O(length(L))*
`add_first`	*O(length(L))*	*O(1)*
`add_last`	*O(length(L))*	*O(length(L))*
`remove`	*O(length(L))*	*O(length(L))*
`contains`	*O(length(L))*	*O(length(L))*

Complexité (2)

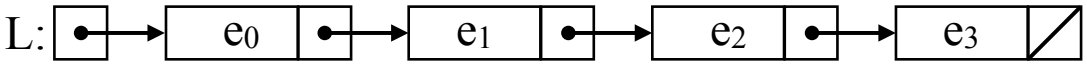
- Le tableau précédent donne la complexité temporelle
- Quid de la complexité *spatiale*?
  - l'implémentation par tableau implique un certain gaspillage d'espace mémoire
  - l'implémentation par pointeur n'utilise que ce qui est nécessaire
    - ✓ mais coût lié à la partie liante
- En pratique, on préférera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

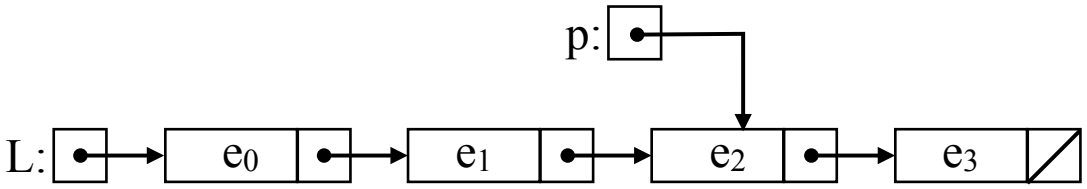
- La liste (e_0, e_1, e_2, e_3) est représentée comme suit:



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

Notations (2)

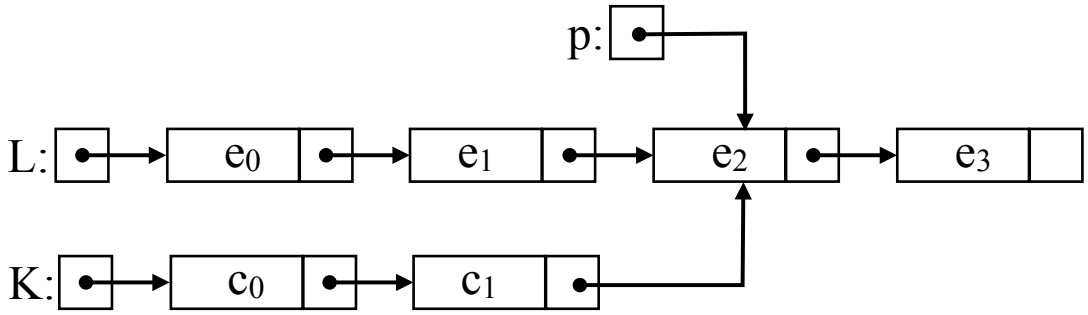
- Notation pour $L^+ = (e_0, e_1, e_2, e_3)$
  - $p^+(e_1, e_2)$ est une sous-liste de L^+
    - ✓ (e_2, e_4) n'en est pas une
- Soient une liste L^+ et une sous-liste p^+ de L^+:



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

Notations (3)

- Exemple d'ambiguité



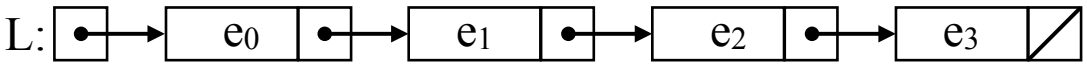
- p^- est ambiguë car on ne sait pas si c'est
  - $p^-_L = (e_0, e_1)$
  - $p^-_K = (c_0, c_1)$

Notations (4)

- Remarque
  - si $L^+ = (e_0, e_1, e_2, e_3)$
    - ✓ $L^- = ()$
  - si L = `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*

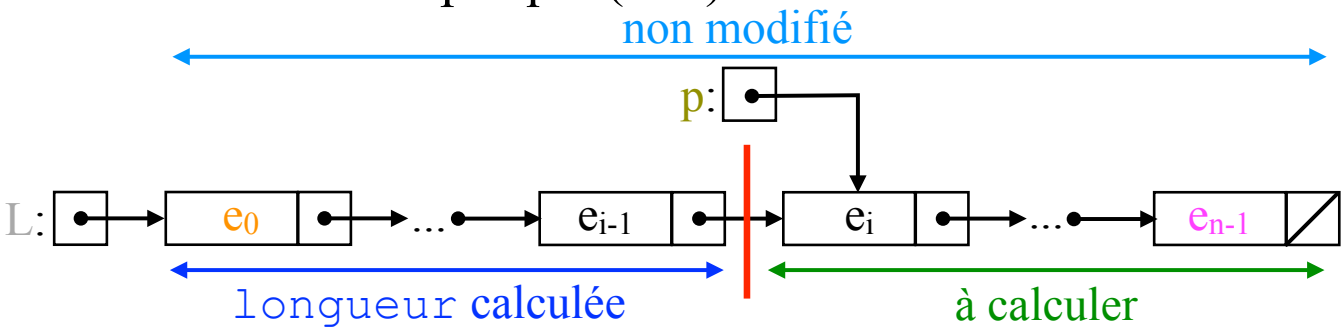
length() (2)

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

```
/*
 * @pre: ∅
 * @post: L = L₀ ∧ length = long(L₀)
 */
int length(List *L);
```

length() (3)

- Invariant Graphique (GLI)



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

- Invariant Formel (FLI)

  $L = L_0$

  ∧

  longueur = long(p^-)

length() (4)

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

Par définition, $p^- = ()$

$\Rightarrow long(p^-) = 0$

length() (5)

- Instructions

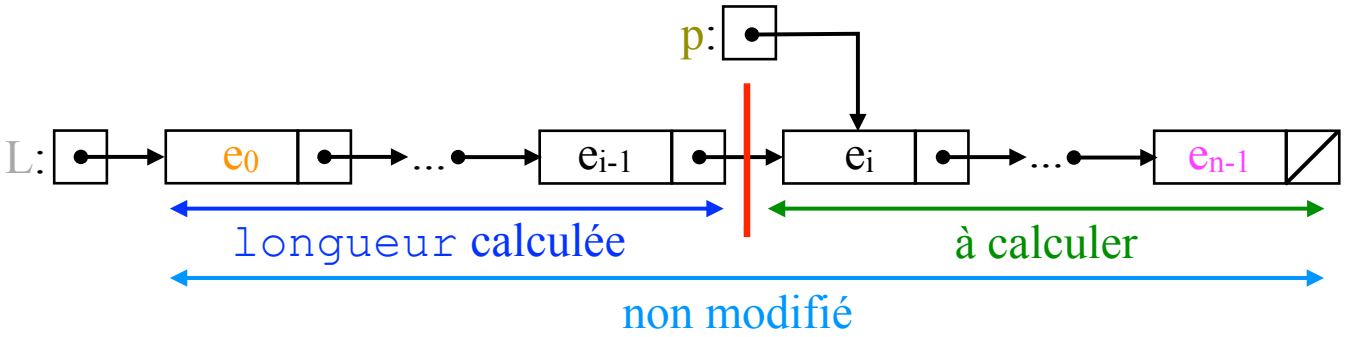
```
{Pré ≡ ∅}
List *p = L;
{L = L0 ∧ p = L}
int longueur = 0;
{Inv ≡ L = L0 ∧ longueur = long(p-)}
```


length() (6)

- Critère d'Arrêt (¬B)
  - $p^+ = ()$
  - `p == NULL`
- {Inv ∧ ¬B} `END` {Post}
  - rien à faire, hormis retourner la valeur de longueur
  - {Inv ∧ ¬B} ⇒ {Post}
    - ✓ {L = L_0 ∧ longueur = long(p^-) ∧ p^+ = ()}
    - ✓ si $p^+ = ()$, alors p^- indique qu'on a parcouru tout L
      - $p^- = L^+$
    - ✓ ⇒ $longueur = long(L_0)$

length() (7)

- {Inv ∧ B} ITER {Inv}



Il faut établir:

incrémenter longueur

```
long(p) = long(p-) + 1
```

avancer p à la cellule suivante

```
p = p1->next;
```


length() (8)

- Instructions

```
{Inv ≡ L = L0 ∧ longueur = long(p-)}
while(p!=NULL){
  {Inv ∧ B ≡ L = L0 ∧ longueur = long(p-) ∧ p+ ≠ ()}
  longueur++;
  {L = L0 ∧ longueur = long(p-)+1 ∧ p+ ≠ ()}
  p = p->next;
  {Inv ≡ L = L0 ∧ longueur = long(p-)}
}//fin while - p
```

length() (9)

- Code complet

```
int length(List *L){
  {Pré ≡ ∅}
  List *p = L;
  int longueur = 0;

  {Inv ≡ L = L₀ ∧ longueur = long(p-)}
  while(p!=NULL){
    longueur++;
    p = p->next;
  }//fin while - p

  return longueur;
  {Post ≡ L = L₀ ∧ length = long(L₀)}
}//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`
  - Caractérisation des Inputs
    - ✓ `L` est de type `List *`
      - ‣ ne peut être vide
    - ✓ `i` est de type `int`
      - ‣ $0 \leq i <$ `length(L)`

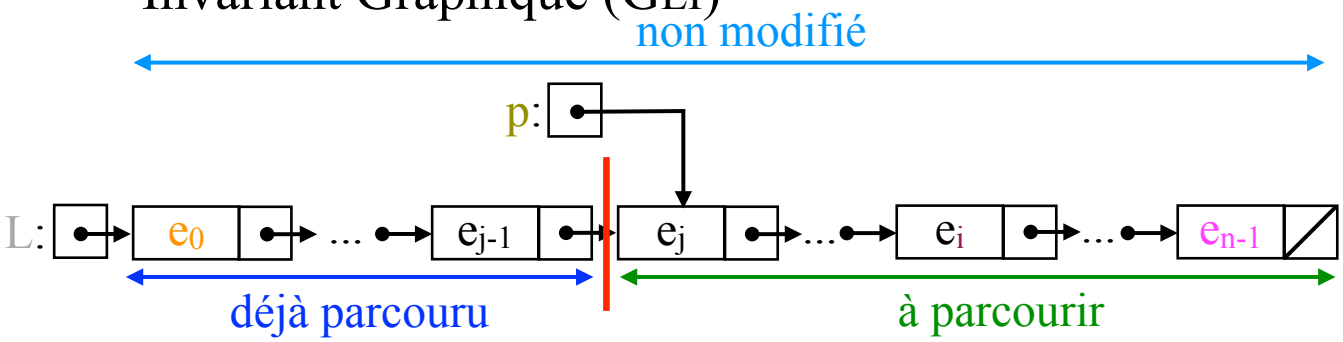
get() (2)

- Spécification

```
/*
 * @pre: L ≠ () ∧ 0 ≤ i < length(L)
 * @post: L = L₀ ∧ i = i₀ ∧ get = eᵢ
 */
void *get(List *L, int i);
```


get() (3)

- Invariant Graphique (GLI)



- Invariant Formel (FLI)

$L = L_0 \wedge i = i_0$

$\wedge$

$0 \leq long(p^-) \leq i$

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

get() (4)

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

Par définition, $p^- = ()$

$\Rightarrow long(p^-) = 0$

déclarer un indice *j* à initialiser à 0

get() (5)

- Instructions

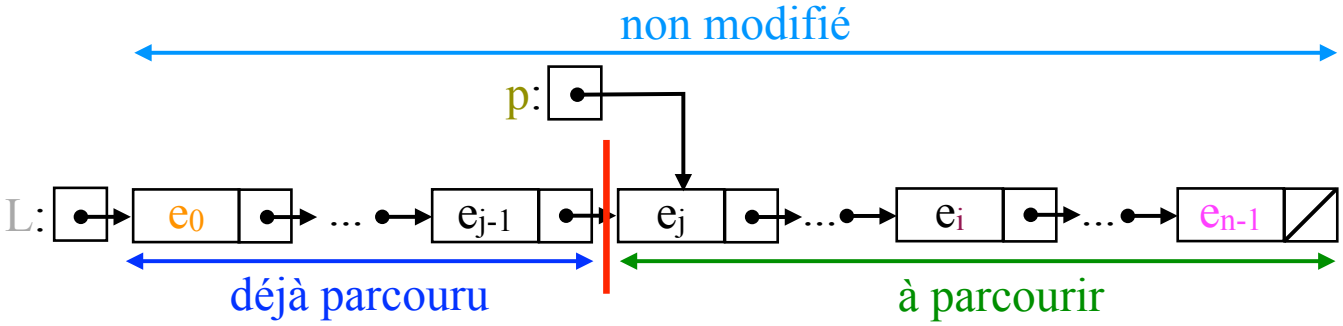
```
{Pré ≡ L initialisée ∧ 0 ≤ i < length(L)}
List *p = L;
{L = L0 ∧ i = i0 ∧ p = L}
int j = 0;
{Inv ≡ L = L0 ∧ i = i0 ∧ 0 ≤ long(p-) ≤ i ∧ j = long(p-)}
```

get() (6)

- Critère d'Arrêt (¬B)
  - `j == i`
- {Inv ∧ ¬B} `END` {Post}
  - rien à faire, hormis retourner la valeur au rang i
  - {Inv ∧ ¬B} ⇒ {Post}
    - ✓ à démontrer en exercice, à la maison

get() (7)

- {Inv ∧ B} `ITER` {Inv}



Il faut établir:

avancer `p` à la cellule suivante

```
p = p->next;
```

incrémenter l'indice j

```
j = j1 + 1
```

get() (8)

- Instructions

```
{Inv ≡ L = L₀ ∧ i = i₀ ∧ 0 ≤ long(p⁻) ≤ i ∧  j = long(p⁻)}
while(j!=i){
  {Inv ∧ B ≡ L = L₀ ∧ i = i₀ ∧ 0 ≤ long(p⁻) ≤ i ∧ j ≠ i}
  p = p->next;
  {L = L₀ ∧ i = i₀ ∧ 0 ≤ long(p⁻) < i ∧ j ≠ i}
  j++;
  {Inv ≡ L = L₀ ∧ i = i₀ ∧ 0 ≤ long(p⁻) ≤ i ∧ j = long(p⁻)}
}//fin while
```


get() (9)

- Code complet

```
void *get(List *L, int i){
  {Pré ≡ L initialisée ∧ 0 ≤ i < length(L)}
  List *p = L;
  int j = 0;

  {Inv ≡ L = L₀ ∧ i = i₀ ∧ 0 ≤ long(p⁻) ≤ i}
  while(j!=i){
    p = p->next;
    j++;
  }//fin while

  return p->data;
  {Post ≡ L = L₀ ∧ i = i₀ ∧ get = eᵢ}
}//fin get()
```

`set()`

- Grosso modo le même principe que `get()`
- Deux différences
  - expression de la postcondition
  - {Inv ∧ ¬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
- Caractérisation des Inputs
  - `L` de type `List *`
    - ✓ peut être éventuellement vide
  - `e` de type `void *`
    - ✓ ne peut être `NULL`

add_first() (2)

- Analyse
  - SP_1: créer une nouvelle cellule
  - SP_2: rattacher la cellule à L, si possible
- Enchaînement des SPs
  - $SP_1 \rightarrow SP_2$
- 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()
```

SP_1

SP_2

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
  - Caractérisation des Inputs
    - ✓ `L` est de type `List *`
      - peut être éventuellement vide
    - ✓ `e` est de type `void *`
      - ne peut être `NULL`

add_last() (2)

- Analyse du problème
  - SP_1: créer une nouvelle cellule
    - ✓ identique au SP1 de `add_first()`
  - SP_2: `L` est vide
  - SP_3: `L` n'est pas vide
- SP_3 peut être subdivisé en 2 autres SPs
  - $SP_{3.a}$: parcourir `L` jusqu'à la dernière cellule
  - $SP_{3.b}$: rattacher la nouvelle cellule, si possible, à `L`
- Enchaînement des SPs
  - $SP_1 \rightarrow SP_2$
    $\rightarrow (SP_{3.a} \rightarrow SP_{3.b})$

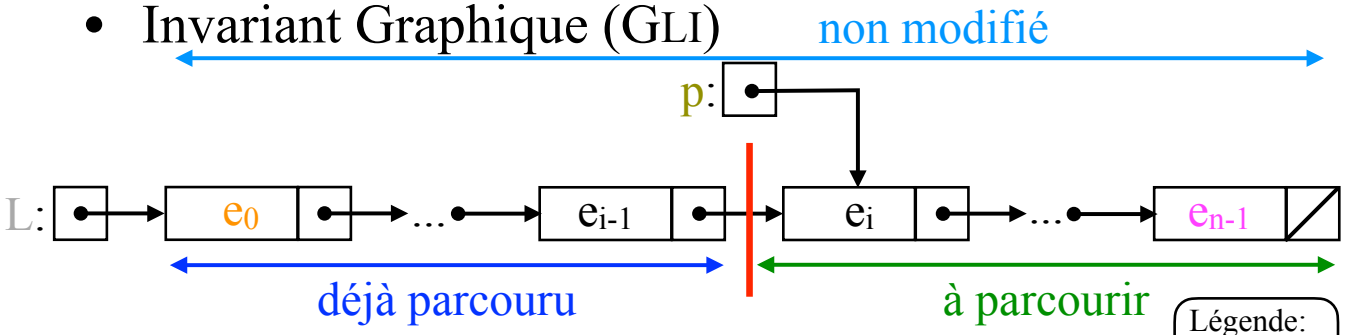
add_last() (3)

- 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)



- Invariant Formel (FLI)

$L = L_0 \wedge e = e_0$

$\wedge$

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

$\wedge$

$long(p^+) \geq 1$

add_last() (5)

- Code pour SP_2

```
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
  - Caractérisation des Inputs
    - ✓ `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
  - SP_1: le rang vaut 0
    - ✓ problème identique à `add_first()`
  - SP_2: le rang est plus grand ou égal à 1
- SP_2 peut être divisé en 3 SPs
  - $SP_{2.a}$: se positionner au rang `i`
  - $SP_{2.b}$: créer une nouvelle cellule
    - ✓ identique au SP_1 de `add_first()`
  - $SP_{2.c}$: rattacher la cellule à la liste
- Enchaînement des SPs
  - SP_1

    $SP_{2.a} \rightarrow SP_{2.b} \rightarrow SP_{2.c}$

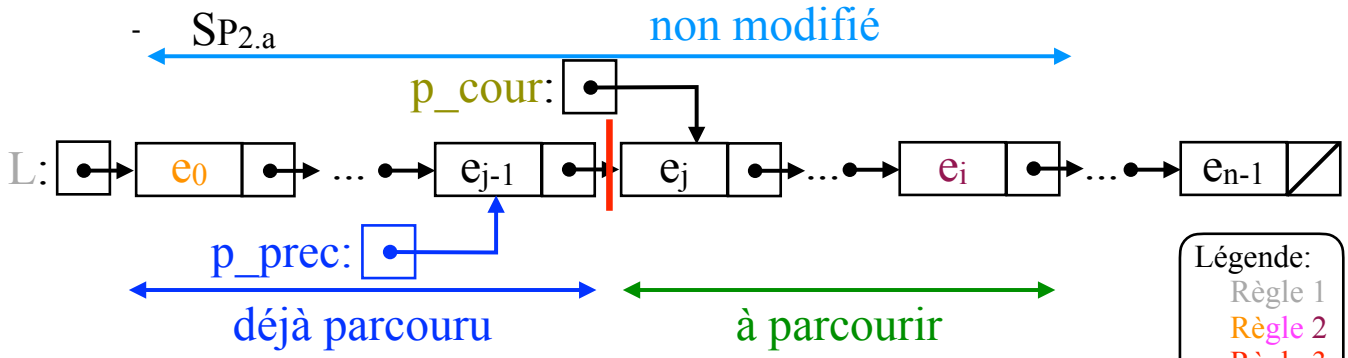
add_at() (3)

- 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)
  - $SP_{2.a}$



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

- Invariant Formel (FLI)

$e = e_0 \wedge L = L_0$

$\wedge$

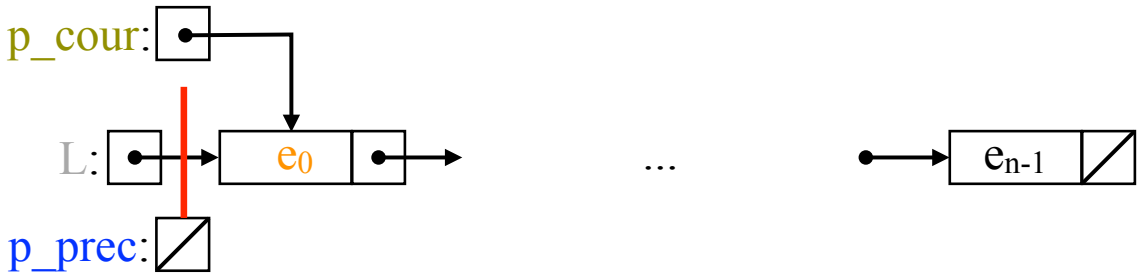
$0 \leq j \leq i$

$\wedge$

$p\_prec^- = (e_0, ..., e_{j-2}) \wedge p\_cour^- = p\_prec^- \,||\, (e_{j-1})$

add_at() (5)

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



Il faut établir

`p_cour` sur la 1ère cellule

`p_prec` à `p_cour`$^-$

`j` = 0

add_at() (6)

- Instructions

```
{Pré ≡ e ≠ NULL ∧ 0 ≤ i < length(L)}
int j = 0;
{e = e0 ∧ L = L0 ∧ 0 ≤ i < length(L) ∧ j = 0}
{e = e0 ∧ L = L0 ∧ 0 ≤ j ≤ i}
List *p_cour = L;
{e = e0 ∧ L = L0 ∧ 0 ≤ j ≤ i ∧ p_cour- = ()}
List *p_prec = NULL;
{e = e0 ∧ L = L0 ∧ 0 ≤ j ≤ i ∧ p_cour- = () ∧ p_prec = ()}
{Inv ≡ e = e0 ∧ L = L0 ∧ 0 ≤ j ≤ i  ∧ p_prec-=(e0, ..., ej-2)
      ∧ p_cour- = p_prec- || (ej-1)}
```


add_at() (7)

- Critère d'Arrêt (¬B)
  - j == i
- {Inv ∧ 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 ≡ e = e0 ∧ L = L0 ∧ 0 ≤ j ≤ i  ∧ p_prec-=(e0, ..., ej-2)
       ∧ p_cour- = p_prec- || (ej-1)}
while(j<i){
   {Inv ∧ B ≡ e = e0 ∧ L = L0 ∧ 0 ≤ j < i  ∧
    p_prec-=(e0, ..., ej-2) ∧ p_cour- = p_prec- || (ej-1)}
   p_prec = p_cour;
   {e = e0 ∧ L = L0 ∧ 0 ≤ j < i  ∧ p_prec-=(e0, ..., ej-1) ∧
    p_cour- = p_prec-}
   p_cour = p_cour->next;
   {e = e0 ∧ L = L0 ∧ 0 ≤ j < i  ∧ p_prec-=(e0, ..., ej-1) ∧
    p_cour- = p_prec- || (ej)}
   j++;
   {Inv ≡ e = e0 ∧ L = L0 ∧ 0 ≤ j ≤ i  ∧
        p_prec-=(e0, ..., ej-2) ∧ p_cour- = p_prec- || (ej-1)}
}//fin while - j
```


add_at() (9)

- $SP_{2.c}$

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

  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

  List *n_cell = create_cell(e);
  if(n_cell == NULL) return L;

  n_cell->next = p_cour;
  p_prec->next = n_cell;

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

SP_1

$SP_{2.a}$

$SP_{2.b}$

$SP_{2.c}$

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

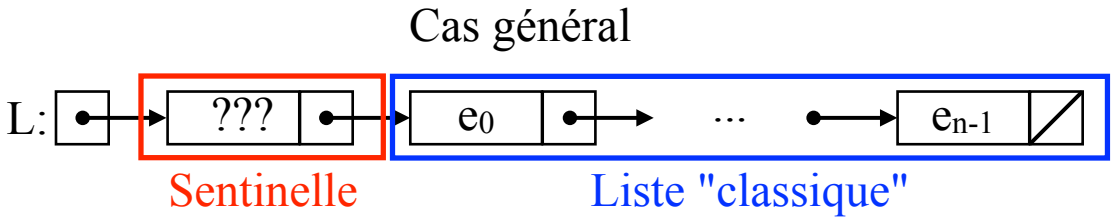
- Ajout d'une cellule en tête de liste
  - `L->next` pointe sur la 1ère cellule de la liste
  - `L->data` ne contient rien
- **Sentinelle**
- Toutes les fonctions prennent en argument un pointeur vers la sentinelle
  - et non plus la tête de liste

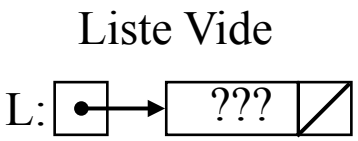
Cellule En-Tête (2)

- 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->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
  - $L^+ = s \,||\, (e_0, e_1, \ldots, e_{n-1})$

Cellule En-Tête (3)

- Implémentation





Cellule En-Tête (4)

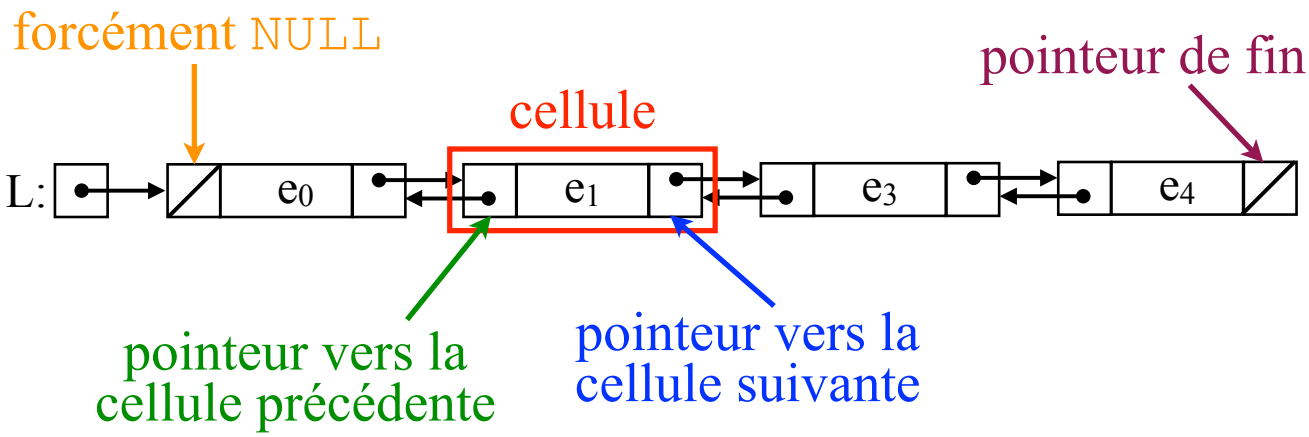
- Structure de données
  - identique à une liste chaînée classique
  - ce sont les processus de création et de manipulation qui diffèrent

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)

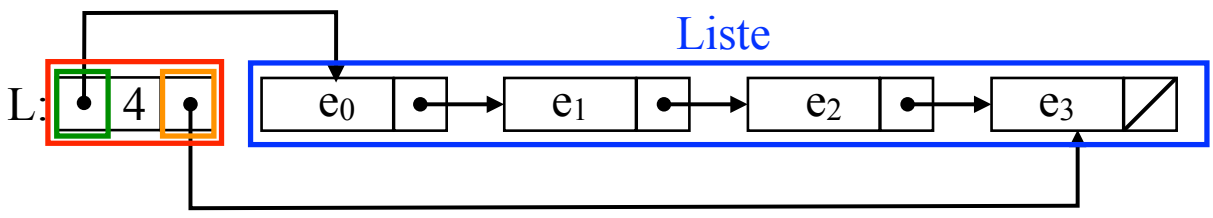
- Le parcours d'une liste doublement chaînée n'est pas différent d'une liste chaînée
  - si on avance (i.e., gauche à droite)
    - ✓ les algorithmes présentés précédemment sont identiques
  - si on recule (i.e., droite à gauche)
    - ✓ on applique le même principe qu'une liste chaînée mais sur le champ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

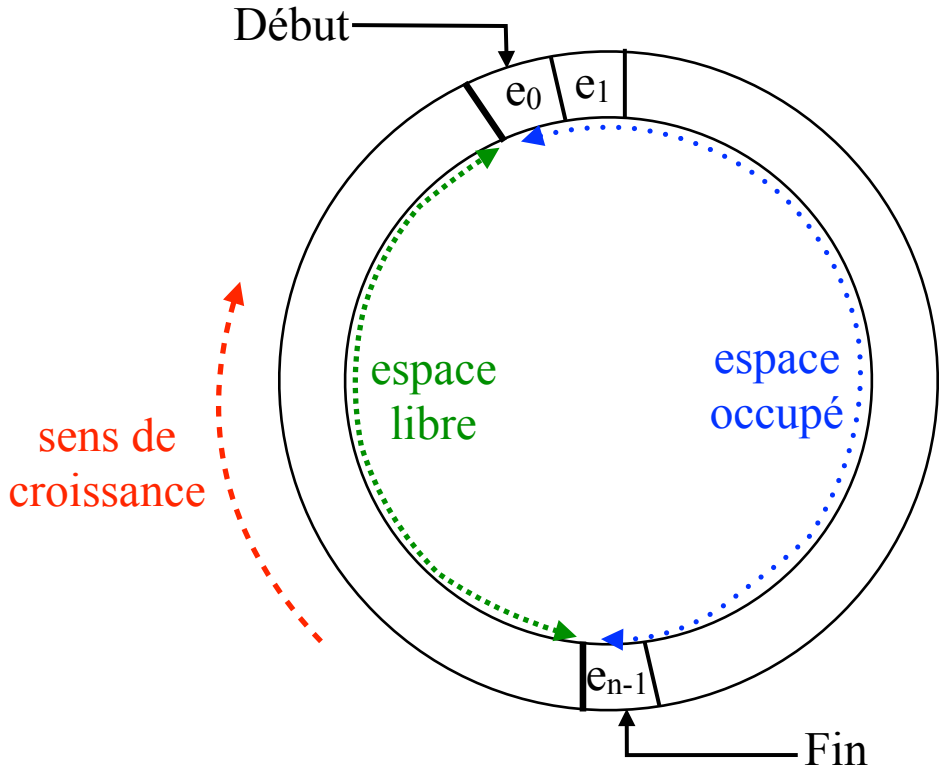

Pointeur Début/Fin (4)

- 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

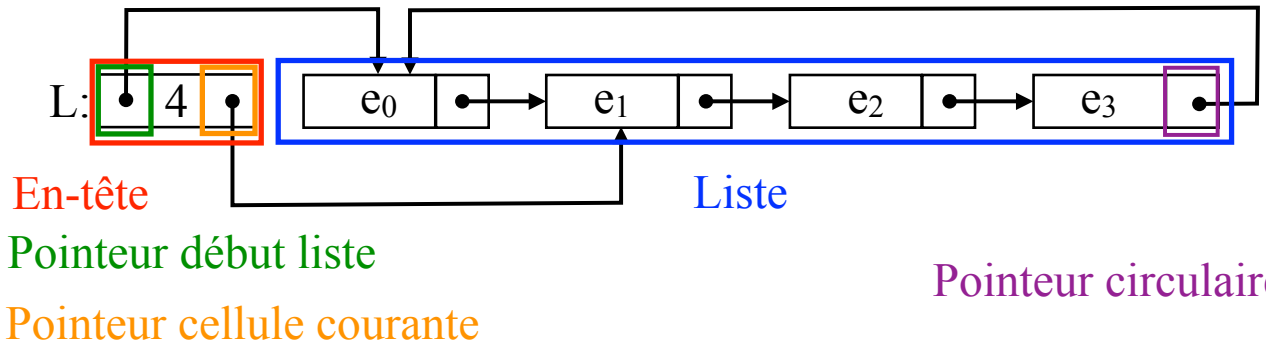
- Dans certaines circonstances, il peut être utile de développer une structure de données
  - qui permette de stocker dynamiquement de l'information
  - sans pour autant accroître "indéfiniment" l'espace mémoire utilisé
- On stocke les données de manière ordonnée
  - on commence au début jusqu'à arriver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

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 1ère cellule