Programmation en C

Tableaux redimensionnables, Listes chaînées, Piles et Files

Régis Barbanchon

L1 Info-Math, Semestre 2

Vecteurs (ou tableaux redimensionnables)

On se donne le type `VectorOfInt` gérant un tableau d'entiers `elements[]` de capacité `capacity` et de longueur `length`.
Seuls les 1ers éléments d'indices `0..length-1` sont utilisés.

```
typedef struct VectorOfInt {
  int length, capacity;
  int * elements;
} VectorOfInt;
```

Vecteurs (ou tableaux redimensionnables)

On se donne le type `VectorOfInt` gérant un tableau d'entiers `elements[]` de capacité `capacity` et de longueur `length`.
Seuls les 1ers éléments d'indices `0..length-1` sont utilisés.

```
typedef struct VectorOfInt {
  int length, capacity;
  int * elements;
} VectorOfInt;
```

Le tableau est alloué dynamiquement par l'initialiseur `Init()` :

```
void VectorOfInt_Init (VectorOfInt * self, int capacity) {
  assert (capacity > 0);
  self->capacity= capacity;
  self->length= 0;
  self->elements= malloc (capacity * sizeof * self->elements);
  assert (self->elements != NULL);
}
```

Vecteurs (ou tableaux redimensionnables)

On se donne le type `VectorOfInt` gérant un tableau d'entiers `elements[]` de capacité `capacity` et de longueur `length`.
Seuls les 1ers éléments d'indices `0..length-1` sont utilisés.

```
typedef struct VectorOfInt {
  int length, capacity;
  int * elements;
} VectorOfInt;
```

Le tableau est alloué dynamiquement par l'initialiseur `Init()` :

```
void VectorOfInt_Init (VectorOfInt * self, int capacity) {
  assert (capacity > 0);
  self->capacity= capacity;
  self->length= 0;
  self->elements= malloc (capacity * sizeof * self->elements);
  assert (self->elements != NULL);
}
```

Il est détruit par le finaliseur `Clean()` :

```
void VectorOfInt_Clean (VectorOfInt * self) {
  free (self->elements);
}
```

Vecteurs : prédicats

On se donne par ailleurs les prédicats `IsEmpty()` et `IsFull()` :

```
bool VectorOfInt_IsEmpty (VectorOfInt const * self) {
  return self->length == 0;
}
```

```
bool VectorOfInt_IsFull (VectorOfInt const * self) {
  return self->length == self->capacity;
}
```

Vecteurs : prédicats

On se donne par ailleurs les prédicats `IsEmpty()` et `IsFull()` :

```
bool VectorOfInt_IsEmpty (VectorOfInt const * self) {
  return self->length == 0;
}
```

```
bool VectorOfInt_IsFull (VectorOfInt const * self) {
  return self->length == self->capacity;
}
```

Exemple de test pour le moment :

```
void Test_VectorOfInt_Init (void) {
  VectorOfInt vector;
  VectorOfInt_Init (& vector, 3);
  assert (   VectorOfInt_IsEmpty (& vector));
  assert ( ! VectorOfInt_IsFull  (& vector));
  VectorOfInt_Clean (& vector);
}
```

Vecteurs : prédicats

On se donne par ailleurs les prédicats `IsEmpty()` et `IsFull()` :

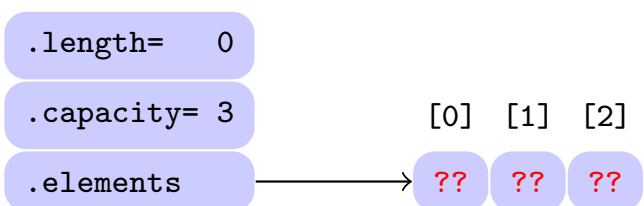
```
bool VectorOfInt_IsEmpty (VectorOfInt const * self) {
  return self->length == 0;
}
```

```
bool VectorOfInt_IsFull (VectorOfInt const * self) {
  return self->length == self->capacity;
}
```

Exemple de test pour le moment :

```
void Test_VectorOfInt_Init (void) {
  VectorOfInt vector;
  VectorOfInt_Init (& vector, 3);
  assert (   VectorOfInt_IsEmpty (& vector));
  assert ( ! VectorOfInt_IsFull  (& vector));
  VectorOfInt_Clean (& vector);
}
```

`vector` a la forme suivante après son initialisation :



Vecteurs : ajout simple à la fin

Écrivons `JustAddLast()` qui ajoute juste un élément à la fin, sans changer la capacité du tableau pourvu qu'il ne soit pas plein :

```
void VectorOfInt_JustAddLast (VectorOfInt * self, int element) {
  assert ( ! VectorOfInt_IsFull (self));
  self->elements [self->length]= element;
  self->length++;
}
```

Vecteurs : ajout simple à la fin

Écrivons `JustAddLast()` qui ajoute juste un élément à la fin, sans changer la capacité du tableau pourvu qu'il ne soit pas plein :

```
void VectorOfInt_JustAddLast (VectorOfInt * self, int element) {
  assert ( ! VectorOfInt_IsFull (self));
  self->elements [self->length]= element;
  self->length++;
}
```

Exemple d'une séquence de trois ajouts à la fin :

```
int main (void) {
  VectorOfInt vector;
  VectorOfInt_Init (& vector, 3);
  VectorOfInt_JustAddLast (& vector, 10);
  VectorOfInt_JustAddLast (& vector, 20);
  VectorOfInt_JustAddLast (& vector, 30);
  VectorOfInt_Clean (& vector);
  return 0;
}
```

Vecteurs : ajout simple à la fin

Écrivons `JustAddLast()` qui ajoute juste un élément à la fin, sans changer la capacité du tableau pourvu qu'il ne soit pas plein :

```
void VectorOfInt_JustAddLast (VectorOfInt * self, int element) {
  assert ( ! VectorOfInt_IsFull (self));
  self->elements [self->length]= element;
  self->length++;
}
```

Exemple d'une séquence de trois ajouts à la fin :

```
int main (void) {
  VectorOfInt vector;
  VectorOfInt_Init (& vector, 3);
  VectorOfInt_JustAddLast (& vector, 10);
  VectorOfInt_JustAddLast (& vector, 20);
  VectorOfInt_JustAddLast (& vector, 30);
  VectorOfInt_Clean (& vector);
  return 0;
}
```

On a l'évolution suivante pour l'état de `vector` :

`.length= ??`

`.capacity=??`

`.elements=??`

Vecteurs : ajout simple à la fin

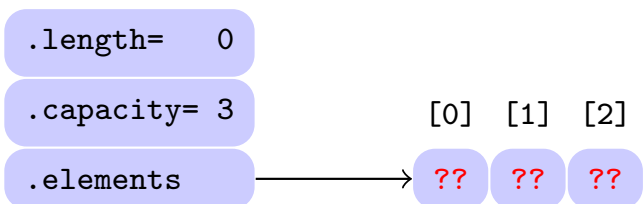
Écrivons `JustAddLast()` qui ajoute juste un élément à la fin, sans changer la capacité du tableau pourvu qu'il ne soit pas plein :

```
void VectorOfInt_JustAddLast (VectorOfInt * self, int element) {
  assert ( ! VectorOfInt_IsFull (self));
  self->elements [self->length]= element;
  self->length++;
}
```

Exemple d'une séquence de trois ajouts à la fin :

```
int main (void) {
  VectorOfInt vector;
  VectorOfInt_Init (& vector, 3);
  VectorOfInt_JustAddLast (& vector, 10);
  VectorOfInt_JustAddLast (& vector, 20);
  VectorOfInt_JustAddLast (& vector, 30);
  VectorOfInt_Clean (& vector);
  return 0;
}
```

On a l'évolution suivante pour l'état de `vector` :



Vecteurs : ajout simple à la fin

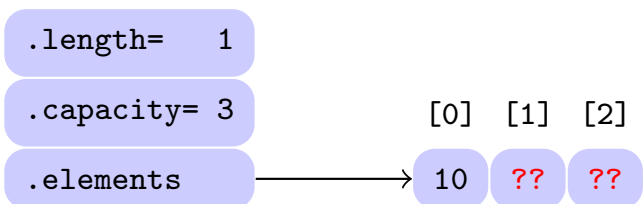
Écrivons `JustAddLast()` qui ajoute juste un élément à la fin, sans changer la capacité du tableau pourvu qu'il ne soit pas plein :

```
void VectorOfInt_JustAddLast (VectorOfInt * self, int element) {
  assert ( ! VectorOfInt_IsFull (self));
  self->elements [self->length]= element;
  self->length++;
}
```

Exemple d'une séquence de trois ajouts à la fin :

```
int main (void) {
  VectorOfInt vector;
  VectorOfInt_Init (& vector, 3);
  VectorOfInt_JustAddLast (& vector, 10); •
  VectorOfInt_JustAddLast (& vector, 20);
  VectorOfInt_JustAddLast (& vector, 30);
  VectorOfInt_Clean (& vector);
  return 0;
}
```

On a l'évolution suivante pour l'état de `vector` :



Vecteurs : ajout simple à la fin

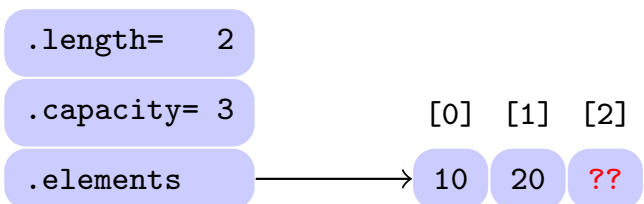
Écrivons `JustAddLast()` qui ajoute juste un élément à la fin, sans changer la capacité du tableau pourvu qu'il ne soit pas plein :

```
void VectorOfInt_JustAddLast (VectorOfInt * self, int element) {
  assert ( ! VectorOfInt_IsFull (self));
  self->elements [self->length]= element;
  self->length++;
}
```

Exemple d'une séquence de trois ajouts à la fin :

```
int main (void) {
  VectorOfInt vector;
  VectorOfInt_Init (& vector, 3);
  VectorOfInt_JustAddLast (& vector, 10);
  VectorOfInt_JustAddLast (& vector, 20); •
  VectorOfInt_JustAddLast (& vector, 30);
  VectorOfInt_Clean (& vector);
  return 0;
}
```

On a l'évolution suivante pour l'état de `vector` :



Vecteurs : ajout simple à la fin

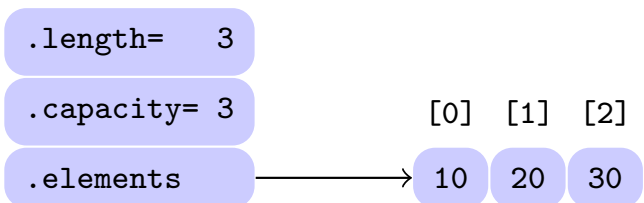
Écrivons `JustAddLast()` qui ajoute juste un élément à la fin, sans changer la capacité du tableau pourvu qu'il ne soit pas plein :

```
void VectorOfInt_JustAddLast (VectorOfInt * self, int element) {
  assert ( ! VectorOfInt_IsFull (self));
  self->elements [self->length]= element;
  self->length++;
}
```

Exemple d'une séquence de trois ajouts à la fin :

```
int main (void) {
  VectorOfInt vector;
  VectorOfInt_Init (& vector, 3);
  VectorOfInt_JustAddLast (& vector, 10);
  VectorOfInt_JustAddLast (& vector, 20);
  VectorOfInt_JustAddLast (& vector, 30); •
  VectorOfInt_Clean (& vector);
  return 0;
}
```

On a l'évolution suivante pour l'état de `vector` :



Vecteurs : ajout simple à la fin

Écrivons `JustAddLast()` qui ajoute juste un élément à la fin, sans changer la capacité du tableau pourvu qu'il ne soit pas plein :

```
void VectorOfInt_JustAddLast (VectorOfInt * self, int element) {
  assert ( ! VectorOfInt_IsFull (self));
  self->elements [self->length]= element;
  self->length++;
}
```

Exemple d'une séquence de trois ajouts à la fin :

```
int main (void) {
  VectorOfInt vector;
  VectorOfInt_Init (& vector, 3);
  VectorOfInt_JustAddLast (& vector, 10);
  VectorOfInt_JustAddLast (& vector, 20);
  VectorOfInt_JustAddLast (& vector, 30);
  VectorOfInt_Clean (& vector);
  return 0;
}
```

•

On a l'évolution suivante pour l'état de `vector` :

`.length=  3`

`.capacity= 3`

`.elements=??`

Vecteurs : test de l'ajout simple à la fin

Pour tester `JustAddLast()`, `EqualsArray()` peut être pratique :

```
bool VectorOfInt_EqualsArray (VectorOfInt const * self,
                              int const array[], int array_length) {
  if (self->length != array_length) return false;
  for (int k= 0; k < self->length; k++)
    if (self->elements [k] != array [k]) return false;
  return true;
}
```

Vecteurs : test de l'ajout simple à la fin

Pour tester `JustAddLast()`, `EqualsArray()` peut être pratique :

```
bool VectorOfInt_EqualsArray (VectorOfInt const * self,
                              int const array[], int array_length) {
  if (self->length != array_length) return false;
  for (int k= 0; k < self->length; k++)
    if (self->elements [k] != array [k]) return false;
  return true;
}
```

On peut alors écrire le test suivant pour `JustAddLast()` :

```
void Test_VectorOfInt_JustAddLast (void) {
  VectorOfInt vector;
  VectorOfInt_Init (& vector, 3);
  assert (VectorOfInt_EqualsArray (& vector, NULL, 0));
  assert ( ! VectorOfInt_IsFull (& vector));

  VectorOfInt_JustAddLast (& vector, 10);
  assert (VectorOfInt_EqualsArray (& vector, (int[]) { 10 }, 1));
  assert ( ! VectorOfInt_IsFull (& vector));

  VectorOfInt_JustAddLast (& vector, 20);
  assert (VectorOfInt_EqualsArray (& vector, (int[]) { 10, 20 }, 2));
  assert ( ! VectorOfInt_IsFull (& vector));

  VectorOfInt_JustAddLast (& vector, 30);
  assert (VectorOfInt_EqualsArray (& vector, (int[]) { 10, 20, 30 }, 3));
  assert (   VectorOfInt_IsFull (& vector));
  VectorOfInt_Clean (& vector);
}
```

Vecteurs : dépassement de capacité

Si on dépasse la capacité avec `JustAddLast()`, par exemple...

```
int main (void) {
  Test_VectorOfInt();
  VectorOfInt vector;
  VectorOfInt_Init (& vector, 3);
  VectorOfInt_JustAddLast (& vector, 10);
  VectorOfInt_JustAddLast (& vector, 20);
  VectorOfInt_JustAddLast (& vector, 30);
  VectorOfInt_JustAddLast (& vector, 40); // boom!!! capacity exceeded
  VectorOfInt_Clean (& vector);
  return 0;
}
```

la pré-condition exprimée par `assert()` va faire avorter le programme (sortie par `abort()` et diagnostic de bug sur `stderr`) :

```
void VectorOfInt_JustAddLast(VectorOfInt *, int):
Assertion '! VectorOfInt_IsFull (self)' failed.
Aborted (core dumped)
```

Si la capacité est atteinte lors d'un ajout, on veut ré-allouer un tableau plus grand. Pour cela, on peut utiliser la fonction `realloc()` déclarée dans `<stdlib.h>`

Vecteurs : la fonction `realloc()`

Voici la doc de `realloc()` dans la section 3 du manuel :

```
$ man 3 realloc
```

```
MALLOC(3)                  Linux Programmer's Manual                 MALLOC(3)

NAME
       malloc, free, calloc, realloc - allocate and free dynamic memory

SYNOPSIS
       #include <stdlib.h>
       void *realloc(void *ptr, size_t size);

DESCRIPTION
       The  realloc() function changes the size of the memory block pointed to
       by ptr to size bytes.  The contents will be unchanged in the range from
       the start of the region up to the minimum of the old and new sizes.  If
       the new size is larger than the old size, the added memory will not  be
       initialized.   If  ptr  is  NULL,  then  the call is equivalent to mal-
       loc(size), for all values of size; if size is equal to zero, and ptr is
       not  NULL,  then  the  call  is equivalent to free(ptr).  Unless ptr is
       NULL, it must have been returned by an earlier call to  malloc(),  cal-
       loc()  or  realloc().  If the area pointed to was moved, a free(ptr) is
       done.

RETURN VALUE
       The realloc() function returns a pointer to the newly allocated memory,
       which  is  suitably  aligned for any built-in type and may be different
       from ptr, or NULL if the request fails.  If size was equal to 0, either
       NULL  or  a  pointer  suitable  to be passed to free() is returned.  If
       realloc() fails, the original block is left untouched; it is not  freed
       or moved.
```

Vecteurs : ajout à la fin avec realloc

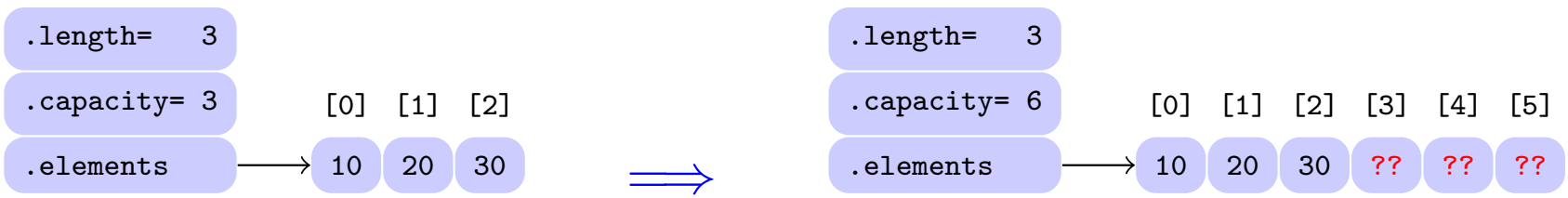
On écrit d'abord la fonction `DoubleCapacity()`
qui double la capacité du tableau via `realloc()` :

```
void VectorOfInt_DoubleCapacity (VectorOfInt * self) {
  int new_capacity= self->capacity * 2;
  int * new_elements= realloc (self->elements, new_capacity);
  assert (new_elements != NULL);
  self->capacity= new_capacity;
  self->elements= new_elements;
}
```

Vecteurs : ajout à la fin avec realloc

On écrit d'abord la fonction `DoubleCapacity()`
qui double la capacité du tableau via `realloc()` :

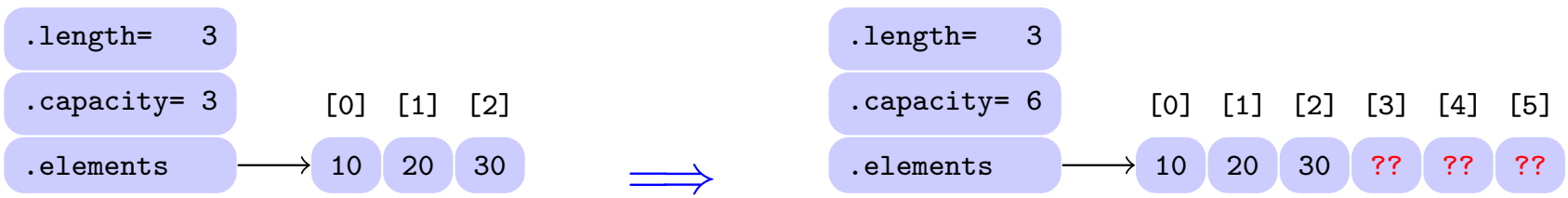
```
void VectorOfInt_DoubleCapacity (VectorOfInt * self) {
  int new_capacity= self->capacity * 2;
  int * new_elements= realloc (self->elements, new_capacity);
  assert (new_elements != NULL);
  self->capacity= new_capacity;
  self->elements= new_elements;
}
```



Vecteurs : ajout à la fin avec realloc

On écrit d'abord la fonction `DoubleCapacity()`
qui double la capacité du tableau via `realloc()` :

```
void VectorOfInt_DoubleCapacity (VectorOfInt * self) {
  int new_capacity= self->capacity * 2;
  int * new_elements= realloc (self->elements, new_capacity);
  assert (new_elements != NULL);
  self->capacity= new_capacity;
  self->elements= new_elements;
}
```



puis on écrit `AddLast()` qui ajoute un élément à la fin du tableau,
en doublant préalablement sa capacité à chaque débordement :

```
void VectorOfInt_AddLast (VectorOfInt * self, int element) {
  if (VectorOfInt_IsFull (self)) VectorOfInt_DoubleCapacity (self);
  VectorOfInt_JustAddLast (self, element);
}
```

Vecteurs : test de l'ajout à la fin avec realloc

On peut alors écrire le test suivant pour `AddLast()` :

```
void Test_VectorOfInt_AddLast (void) {
  VectorOfInt vector;
  VectorOfInt_Init (& vector, 3);

  VectorOfInt_AddLast (& vector, 10);
  VectorOfInt_AddLast (& vector, 20);
  VectorOfInt_AddLast (& vector, 30);
  assert (VectorOfInt_EqualsArray (& vector, (int[]) { 10, 20, 30 }, 3));
  assert (vector.capacity == 3);

  VectorOfInt_AddLast (& vector, 40);
  assert (VectorOfInt_EqualsArray (& vector, (int[]) { 10, 20, 30, 40 }, 4));
  assert (vector.capacity == 6);
  VectorOfInt_Clean (& vector);
}
```

Vecteurs : test de l'ajout à la fin avec realloc

On peut alors écrire le test suivant pour `AddLast()` :

```
void Test_VectorOfInt_AddLast (void) {
  VectorOfInt vector;                    •
  VectorOfInt_Init (& vector, 3);

  VectorOfInt_AddLast (& vector, 10);
  VectorOfInt_AddLast (& vector, 20);
  VectorOfInt_AddLast (& vector, 30);
  assert (VectorOfInt_EqualsArray (& vector, (int[]) { 10, 20, 30 }, 3));
  assert (vector.capacity == 3);

  VectorOfInt_AddLast (& vector, 40);
  assert (VectorOfInt_EqualsArray (& vector, (int[]) { 10, 20, 30, 40 }, 4));
  assert (vector.capacity == 6);
  VectorOfInt_Clean (& vector);
}
```

On a l'évolution suivante pour l'état de `vector` :

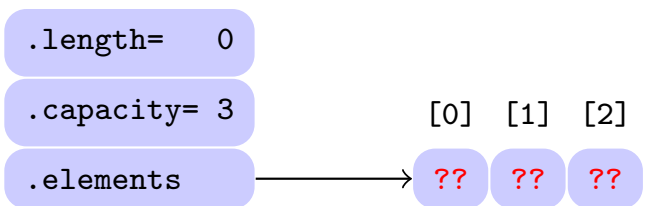
```
.length=  ??
.capacity=??
.elements=??
```

Vecteurs : test de l'ajout à la fin avec realloc

On peut alors écrire le test suivant pour `AddLast()` :

```
void Test_VectorOfInt_AddLast (void) {
  VectorOfInt vector;
  VectorOfInt_Init (& vector, 3);     •

  VectorOfInt_AddLast (& vector, 10);
  VectorOfInt_AddLast (& vector, 20);
  VectorOfInt_AddLast (& vector, 30);
  assert (VectorOfInt_EqualsArray (& vector, (int[]) { 10, 20, 30 }, 3));
  assert (vector.capacity == 3);

  VectorOfInt_AddLast (& vector, 40);
  assert (VectorOfInt_EqualsArray (& vector, (int[]) { 10, 20, 30, 40 }, 4));
  assert (vector.capacity == 6);
  VectorOfInt_Clean (& vector);
}
```

On a l'évolution suivante pour l'état de `vector` :

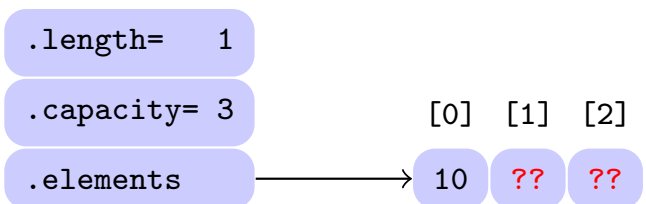


Vecteurs : test de l'ajout à la fin avec realloc

On peut alors écrire le test suivant pour `AddLast()` :

```
void Test_VectorOfInt_AddLast (void) {
  VectorOfInt vector;
  VectorOfInt_Init (& vector, 3);

  VectorOfInt_AddLast (& vector, 10); •
  VectorOfInt_AddLast (& vector, 20);
  VectorOfInt_AddLast (& vector, 30);
  assert (VectorOfInt_EqualsArray (& vector, (int[]) { 10, 20, 30 }, 3));
  assert (vector.capacity == 3);

  VectorOfInt_AddLast (& vector, 40);
  assert (VectorOfInt_EqualsArray (& vector, (int[]) { 10, 20, 30, 40 }, 4));
  assert (vector.capacity == 6);
  VectorOfInt_Clean (& vector);
}
```

On a l'évolution suivante pour l'état de `vector` :

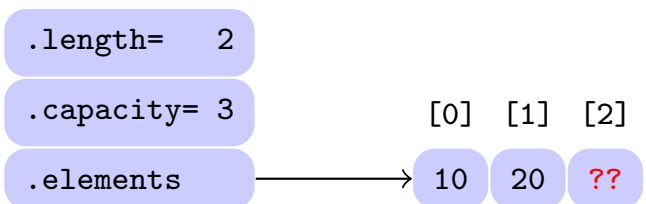


Vecteurs : test de l'ajout à la fin avec realloc

On peut alors écrire le test suivant pour `AddLast()` :

```
void Test_VectorOfInt_AddLast (void) {
  VectorOfInt vector;
  VectorOfInt_Init (& vector, 3);

  VectorOfInt_AddLast (& vector, 10);
  VectorOfInt_AddLast (& vector, 20); •
  VectorOfInt_AddLast (& vector, 30);
  assert (VectorOfInt_EqualsArray (& vector, (int[]) { 10, 20, 30 }, 3));
  assert (vector.capacity == 3);

  VectorOfInt_AddLast (& vector, 40);
  assert (VectorOfInt_EqualsArray (& vector, (int[]) { 10, 20, 30, 40 }, 4));
  assert (vector.capacity == 6);
  VectorOfInt_Clean (& vector);
}
```

On a l'évolution suivante pour l'état de `vector` :

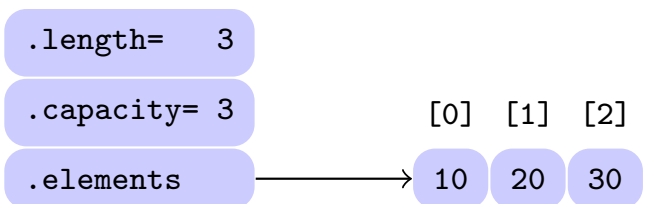


Vecteurs : test de l'ajout à la fin avec realloc

On peut alors écrire le test suivant pour `AddLast()` :

```
void Test_VectorOfInt_AddLast (void) {
  VectorOfInt vector;
  VectorOfInt_Init (& vector, 3);

  VectorOfInt_AddLast (& vector, 10);
  VectorOfInt_AddLast (& vector, 20);
  VectorOfInt_AddLast (& vector, 30); •
  assert (VectorOfInt_EqualsArray (& vector, (int[]) { 10, 20, 30 }, 3));
  assert (vector.capacity == 3);

  VectorOfInt_AddLast (& vector, 40);
  assert (VectorOfInt_EqualsArray (& vector, (int[]) { 10, 20, 30, 40 }, 4));
  assert (vector.capacity == 6);
  VectorOfInt_Clean (& vector);
}
```

On a l'évolution suivante pour l'état de `vector` :

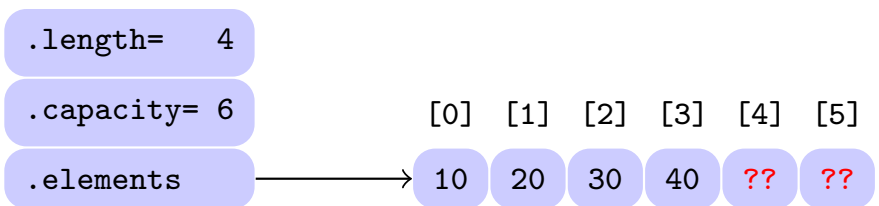


Vecteurs : test de l'ajout à la fin avec realloc

On peut alors écrire le test suivant pour `AddLast()` :

```
void Test_VectorOfInt_AddLast (void) {
  VectorOfInt vector;
  VectorOfInt_Init (& vector, 3);

  VectorOfInt_AddLast (& vector, 10);
  VectorOfInt_AddLast (& vector, 20);
  VectorOfInt_AddLast (& vector, 30);
  assert (VectorOfInt_EqualsArray (& vector, (int[]) { 10, 20, 30 }, 3));
  assert (vector.capacity == 3);

  VectorOfInt_AddLast (& vector, 40); •
  assert (VectorOfInt_EqualsArray (& vector, (int[]) { 10, 20, 30, 40 }, 4));
  assert (vector.capacity == 6);
  VectorOfInt_Clean (& vector);
}
```

On a l'évolution suivante pour l'état de `vector` :



Vecteurs : test de l'ajout à la fin avec realloc

On peut alors écrire le test suivant pour `AddLast()` :

```
void Test_VectorOfInt_AddLast (void) {
  VectorOfInt vector;
  VectorOfInt_Init (& vector, 3);

  VectorOfInt_AddLast (& vector, 10);
  VectorOfInt_AddLast (& vector, 20);
  VectorOfInt_AddLast (& vector, 30);
  assert (VectorOfInt_EqualsArray (& vector, (int[]) { 10, 20, 30 }, 3));
  assert (vector.capacity == 3);

  VectorOfInt_AddLast (& vector, 40);
  assert (VectorOfInt_EqualsArray (& vector, (int[]) { 10, 20, 30, 40 }, 4));
  assert (vector.capacity == 6);
  VectorOfInt_Clean (& vector);      •
}
```

On a l'évolution suivante pour l'état de `vector` :

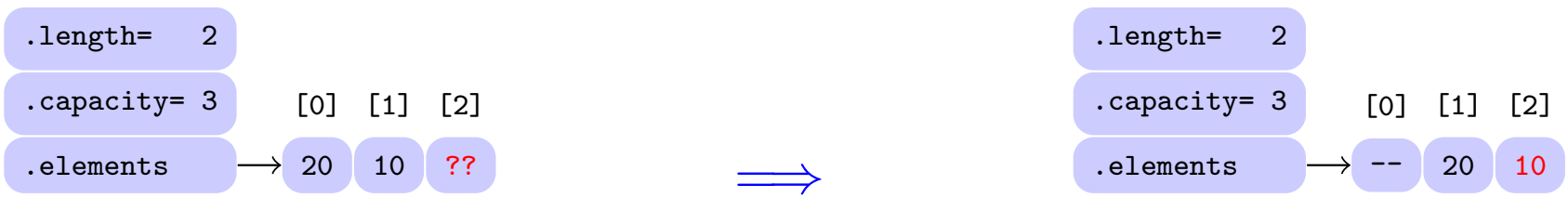
```
.length=   4
.capacity= 6
.elements=??
```

Vecteurs : ajout simple au début

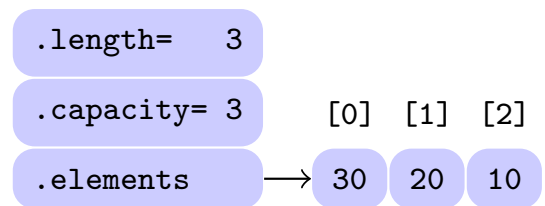
Écrivons `JustAddFirst()` qui ajoute juste un élément au début, sans changer la capacité du tableau pourvu qu'il ne soit pas plein.

```
void VectorOfInt_JustAddFirst (VectorOfInt * self, int element)  {
  assert ( ! VectorOfInt_IsFull (self));
  for (int k= self->length; k > 0; k--) {
    self->elements [k]= self->elements [k-1];
  }
  self->elements [0]= element;
  self->length++;
}
```

Par exemple, si le vecteur contient 20 et 10 avant l'ajout de 30, ces 2 éléments sont d'abord décalés d'une case vers sur la droite :



puis 30 est inséré au début :



Vecteurs : test de l'ajout simple au début

On peut alors écrire le test suivant pour `JustAddFirst()` :

```
void Test_VectorOfInt_JustAddFirst (void) {
  VectorOfInt vector;
  VectorOfInt_Init (& vector, 3);
  assert (VectorOfInt_EqualsArray (& vector, NULL, 0));

  VectorOfInt_JustAddFirst (& vector, 10);
  assert (VectorOfInt_EqualsArray (& vector, (int[]) { 10 }, 1));

  VectorOfInt_JustAddFirst (& vector, 20);
  assert (VectorOfInt_EqualsArray (& vector, (int[]) { 20, 10 }, 2));

  VectorOfInt_JustAddFirst (& vector, 30);
  assert (VectorOfInt_EqualsArray (& vector, (int[]) { 30, 20, 10 }, 3));

  VectorOfInt_Clean (& vector);
}
```

Vecteurs : test de l'ajout simple au début

On peut alors écrire le test suivant pour `JustAddFirst()` :

```
void Test_VectorOfInt_JustAddFirst (void) {
  VectorOfInt vector;                    •
  VectorOfInt_Init (& vector, 3);
  assert (VectorOfInt_EqualsArray (& vector, NULL, 0));

  VectorOfInt_JustAddFirst (& vector, 10);
  assert (VectorOfInt_EqualsArray (& vector, (int[]) { 10 }, 1));

  VectorOfInt_JustAddFirst (& vector, 20);
  assert (VectorOfInt_EqualsArray (& vector, (int[]) { 20, 10 }, 2));

  VectorOfInt_JustAddFirst (& vector, 30);
  assert (VectorOfInt_EqualsArray (& vector, (int[]) { 30, 20, 10 }, 3));

  VectorOfInt_Clean (& vector);
}
```

On a l'évolution suivante pour l'état de `vector` :

`.length= ??`

`.capacity=??`

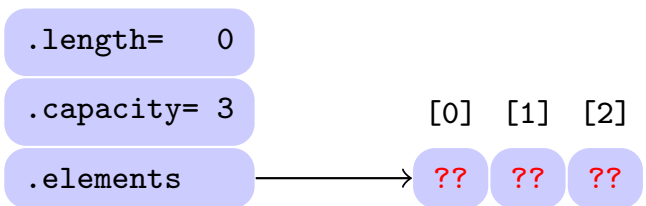
`.elements=??`

Vecteurs : test de l'ajout simple au début

On peut alors écrire le test suivant pour `JustAddFirst()` :

```
void Test_VectorOfInt_JustAddFirst (void) {
  VectorOfInt vector;
  VectorOfInt_Init (& vector, 3);
  assert (VectorOfInt_EqualsArray (& vector, NULL, 0));

  VectorOfInt_JustAddFirst (& vector, 10);
  assert (VectorOfInt_EqualsArray (& vector, (int[]) { 10 }, 1));

  VectorOfInt_JustAddFirst (& vector, 20);
  assert (VectorOfInt_EqualsArray (& vector, (int[]) { 20, 10 }, 2));

  VectorOfInt_JustAddFirst (& vector, 30);
  assert (VectorOfInt_EqualsArray (& vector, (int[]) { 30, 20, 10 }, 3));

  VectorOfInt_Clean (& vector);
}
```

On a l'évolution suivante pour l'état de `vector` :

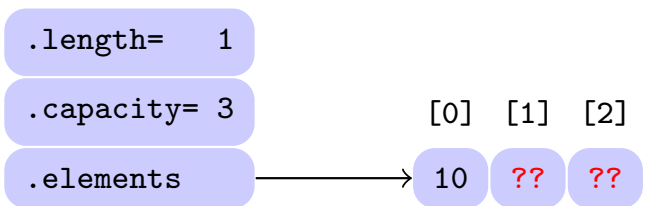


Vecteurs : test de l'ajout simple au début

On peut alors écrire le test suivant pour `JustAddFirst()` :

```
void Test_VectorOfInt_JustAddFirst (void) {
  VectorOfInt vector;
  VectorOfInt_Init (& vector, 3);
  assert (VectorOfInt_EqualsArray (& vector, NULL, 0));

  VectorOfInt_JustAddFirst (& vector, 10); •
  assert (VectorOfInt_EqualsArray (& vector, (int[]) { 10 }, 1));

  VectorOfInt_JustAddFirst (& vector, 20);
  assert (VectorOfInt_EqualsArray (& vector, (int[]) { 20, 10 }, 2));

  VectorOfInt_JustAddFirst (& vector, 30);
  assert (VectorOfInt_EqualsArray (& vector, (int[]) { 30, 20, 10 }, 3));

  VectorOfInt_Clean (& vector);
}
```

On a l'évolution suivante pour l'état de `vector` :

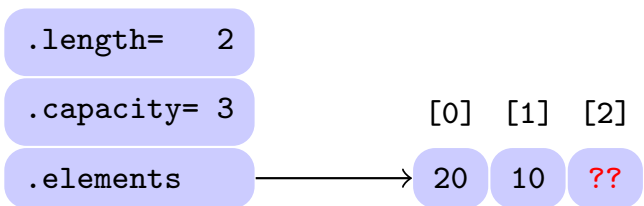


Vecteurs : test de l'ajout simple au début

On peut alors écrire le test suivant pour `JustAddFirst()` :

```
void Test_VectorOfInt_JustAddFirst (void) {
  VectorOfInt vector;
  VectorOfInt_Init (& vector, 3);
  assert (VectorOfInt_EqualsArray (& vector, NULL, 0));

  VectorOfInt_JustAddFirst (& vector, 10);
  assert (VectorOfInt_EqualsArray (& vector, (int[]) { 10 }, 1));

  VectorOfInt_JustAddFirst (& vector, 20); •
  assert (VectorOfInt_EqualsArray (& vector, (int[]) { 20, 10 }, 2));

  VectorOfInt_JustAddFirst (& vector, 30);
  assert (VectorOfInt_EqualsArray (& vector, (int[]) { 30, 20, 10 }, 3));

  VectorOfInt_Clean (& vector);
}
```

On a l'évolution suivante pour l'état de `vector` :

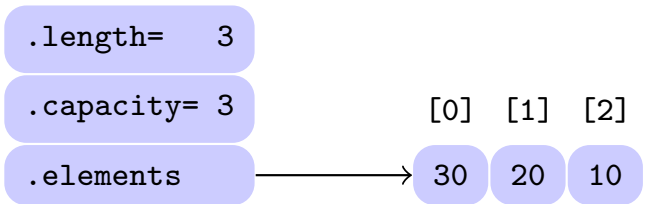


Vecteurs : test de l'ajout simple au début

On peut alors écrire le test suivant pour `JustAddFirst()` :

```
void Test_VectorOfInt_JustAddFirst (void) {
  VectorOfInt vector;
  VectorOfInt_Init (& vector, 3);
  assert (VectorOfInt_EqualsArray (& vector, NULL, 0));

  VectorOfInt_JustAddFirst (& vector, 10);
  assert (VectorOfInt_EqualsArray (& vector, (int[]) { 10 }, 1));

  VectorOfInt_JustAddFirst (& vector, 20);
  assert (VectorOfInt_EqualsArray (& vector, (int[]) { 20, 10 }, 2));

  VectorOfInt_JustAddFirst (& vector, 30); •
  assert (VectorOfInt_EqualsArray (& vector, (int[]) { 30, 20, 10 }, 3));

  VectorOfInt_Clean (& vector);
}
```

On a l'évolution suivante pour l'état de `vector` :



Vecteurs : test de l'ajout simple au début

On peut alors écrire le test suivant pour `JustAddFirst()` :

```
void Test_VectorOfInt_JustAddFirst (void) {
  VectorOfInt vector;
  VectorOfInt_Init (& vector, 3);
  assert (VectorOfInt_EqualsArray (& vector, NULL, 0));

  VectorOfInt_JustAddFirst (& vector, 10);
  assert (VectorOfInt_EqualsArray (& vector, (int[]) { 10 }, 1));

  VectorOfInt_JustAddFirst (& vector, 20);
  assert (VectorOfInt_EqualsArray (& vector, (int[]) { 20, 10 }, 2));

  VectorOfInt_JustAddFirst (& vector, 30);
  assert (VectorOfInt_EqualsArray (& vector, (int[]) { 30, 20, 10 }, 3));

  VectorOfInt_Clean (& vector);                    •
}
```

On a l'évolution suivante pour l'état de `vector` :

```
.length=   3
.capacity= 3
.elements=??
```

Vecteurs : ajout au début avec realloc

Comme pour `AddLast()`, on écrit la variante `AddFirst()` de `JustAddFirst()` qui double la capacité en cas de débordement :

```
void VectorOfInt_AddFirst (VectorOfInt * self, int element) {
  if (VectorOfInt_IsFull (self)) VectorOfInt_DoubleCapacity (self);
  VectorOfInt_JustAddFirst (self, element);
}
```

Vecteurs : ajout au début avec realloc

Comme pour `AddLast()`, on écrit la variante `AddFirst()` de `JustAddFirst()` qui double la capacité en cas de débordement :

```
void VectorOfInt_AddFirst (VectorOfInt * self, int element) {
  if (VectorOfInt_IsFull (self)) VectorOfInt_DoubleCapacity (self);
  VectorOfInt_JustAddFirst (self, element);
}
```

Le test de `AddFirst()` est analogue à celui de `AddLast()` :

```
void Test_VectorOfInt_AddFirst (void) {
  VectorOfInt vector;
  VectorOfInt_Init (& vector, 3);

  VectorOfInt_AddFirst (& vector, 10);
  VectorOfInt_AddFirst (& vector, 20);
  VectorOfInt_AddFirst (& vector, 30);
  assert (VectorOfInt_EqualsArray (& vector, (int[]) { 30, 20, 10 }, 3));
  assert (vector.capacity == 3);

  VectorOfInt_AddFirst (& vector, 40);
  assert (VectorOfInt_EqualsArray (& vector, (int[]) { 40, 30, 20, 10 }, 4));
  assert (vector.capacity == 6);
  VectorOfInt_Clean (& vector);
}
```

Vecteurs : suppression à la fin

De même que l'ajout à la fin `AddLast()` est simple,
la suppression à la fin `RemoveLast()` est simple :

```
int VectorOfInt_RemoveLast (VectorOfInt * self) {
  assert ( ! VectorOfInt_IsEmpty (self));
  self->length--;
  return self->elements [self->length];
}
```

Voici un test possible pour `RemoveLast()` :

```
void Test_VectorOfInt_RemoveLast (void) {
  VectorOfInt vector;
  VectorOfInt_Init (& vector, 3);
  VectorOfInt_AddLast (& vector, 10);
  VectorOfInt_AddLast (& vector, 20);
  VectorOfInt_AddLast (& vector, 30);
  assert (VectorOfInt_EqualsArray (& vector, (int[]) { 10, 20, 30 }, 3));

  assert (VectorOfInt_RemoveLast (& vector) == 30);
  assert (VectorOfInt_EqualsArray (& vector, (int[]) { 10, 20 }, 2));
  assert (VectorOfInt_RemoveLast (& vector) == 20);
  assert (VectorOfInt_EqualsArray (& vector, (int[]) { 10 }, 1));
  assert (VectorOfInt_RemoveLast (& vector) == 10);
  assert (VectorOfInt_IsEmpty (& vector));
  VectorOfInt_Clean (& vector);
}
```

Vecteurs : suppression au début

La suppression au début `RemoveFirst()` nécessite un décalage vers la gauche, à l'inverse de celui de `AddFirst()` vers la droite :

```
int VectorOfInt_RemoveFirst (VectorOfInt * self) {
  assert ( ! VectorOfInt_IsEmpty (self));
  int first_element= self->elements [0];
  for (int k= 1; k < self->length; k++)
    self->elements [k-1]= self->elements [k];
  self->length--;
  return first_element;
}
```

Le test de `RemoveFirst()`, analogue de celui de `RemoveLast()` :

```
void Test_VectorOfInt_RemoveFirst (void) {
  VectorOfInt vector;
  VectorOfInt_Init (& vector, 3);
  VectorOfInt_AddLast (& vector, 10);
  VectorOfInt_AddLast (& vector, 20);
  VectorOfInt_AddLast (& vector, 30);
  assert (VectorOfInt_EqualsArray (& vector, (int[]) { 10, 20, 30 }, 3));

  assert (VectorOfInt_RemoveFirst (& vector) == 10);
  assert (VectorOfInt_EqualsArray (& vector, (int[]) { 20, 30 }, 2));
  assert (VectorOfInt_RemoveFirst (& vector) == 20);
  assert (VectorOfInt_EqualsArray (& vector, (int[]) { 30 }, 1));
  assert (VectorOfInt_RemoveFirst (& vector) == 30);
  assert (VectorOfInt_IsEmpty (& vector));
  VectorOfInt_Clean (& vector);
}
```

Vecteurs : la fonction memmove

La fonction `memmove()` déclarée dans `<string.h>`
permet de copier un bloc de mémoire vers un autre.
Les 2 blocs peuvent se chevaucher.

```
$ man 3 memmove
```

```
MEMMOVE(3)                  Linux Programmer's Manual                 MEMMOVE(3)

NAME
       memmove - copy memory area

SYNOPSIS
       #include <string.h>
       void *memmove(void *dest, const void *src, size_t n);

DESCRIPTION
       The  memmove()  function  copies n bytes from memory area src to memory
       area dest.  The memory areas may overlap: copying takes place as though
       the  bytes in src are first copied into a temporary array that does not
       overlap src or dest, and the bytes are then copied from  the  temporary
       array to dest.

RETURN VALUE
       The memmove() function returns a pointer to dest.
```

Vecteurs : décalage des éléménts par memmove

On peut l'utiliser pour décaler les éléments du vecteur, soit vers la droite lors de l'ajout au début :

```
void VectorOfInt_JustAddFirst (VectorOfInt * self, int element)  {
  assert ( ! VectorOfInt_IsFull (self));

  size_t nb_bytes= self->length * sizeof * self->elements;
  memmove (self->elements + 1, self->elements, nb_bytes);

  self->elements [0]= element;
  self->length++;
}
```

soit vers la gauche lors de la suppression au début :

```
int VectorOfInt_RemoveFirst (VectorOfInt * self) {
  assert ( ! VectorOfInt_IsEmpty (self));
  int first_element= self->elements [0];

  int nb_bytes= (self->length - 1) * sizeof * self->elements;
  memmove (self->elements, self->elements + 1, nb_bytes);

  self->length--;
  return first_element;
}
```

Les tests pour `JustAddFirst()` et `RemoveFirst()` passant toujours après ces refactorings, on est rassuré sur leurs validités.

Vecteurs : rassemblement des tests en batterie

Pour conclure sur les tableaux redimensionnables,
voici la batterie de tests pour le type `VectorOfInt` :

```
void Test_VectorOfInt (void) {
  Test_VectorOfInt_Init();
  Test_VectorOfInt_JustAddLast();
  Test_VectorOfInt_AddLast();
  Test_VectorOfInt_JustAddFirst();
  Test_VectorOfInt_AddFirst();
  Test_VectorOfInt_RemoveLast();
  Test_VectorOfInt_RemoveFirst();
}
```

et le lancement de la batterie dans le programme principal :

```
int main (void) {
  Test_VectorOfInt();
  return 0;
}
```

Listes chaînées

Dans une liste chaînée, les valeurs ne sont pas stockées
dans des cases contiguës comme dans un tableau,
mais dans des maillons séparés, chaînés entre eux par pointeurs.

On perd l'accès direct aux données par indexation,
mais on peut insérer ou supprimer une donnée devant d'autres
sans avoir à décaler en mémoire les données suivantes.
La capacité devient également illimitée.

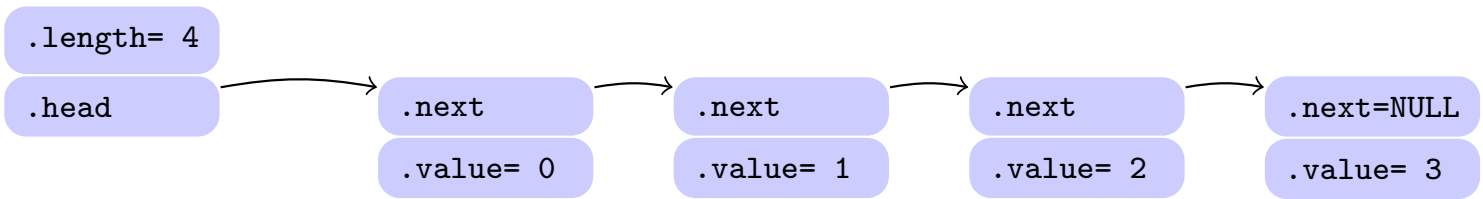
Voici 4 variantes de chaînages pour la liste `(0, 1, 2, 3)` :

Listes chaînées

Dans une liste chaînée, les valeurs ne sont pas stockées
dans des cases contiguës comme dans un tableau,
mais dans des maillons séparés, chaînés entre eux par pointeurs.

On perd l'accès direct aux données par indexation,
mais on peut insérer ou supprimer une donnée devant d'autres
sans avoir à décaler en mémoire les données suivantes.
La capacité devient également illimitée.

Voici 4 variantes de chaînages pour la liste `(0, 1, 2, 3)` :



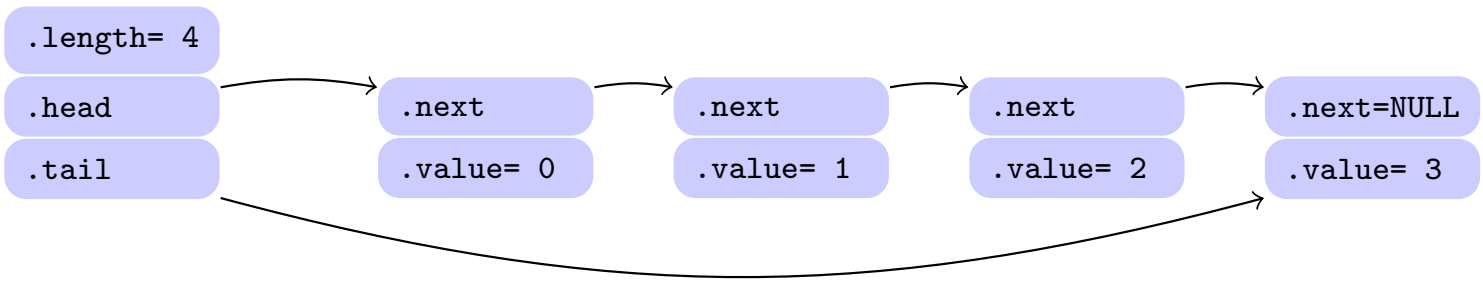
Variante 1 : liste simplement chaînée

Listes chaînées

Dans une liste chaînée, les valeurs ne sont pas stockées
dans des cases contiguës comme dans un tableau,
mais dans des maillons séparés, chaînés entre eux par pointeurs.

On perd l'accès direct aux données par indexation,
mais on peut insérer ou supprimer une donnée devant d'autres
sans avoir à décaler en mémoire les données suivantes.
La capacité devient également illimitée.

Voici 4 variantes de chaînages pour la liste `(0, 1, 2, 3)` :



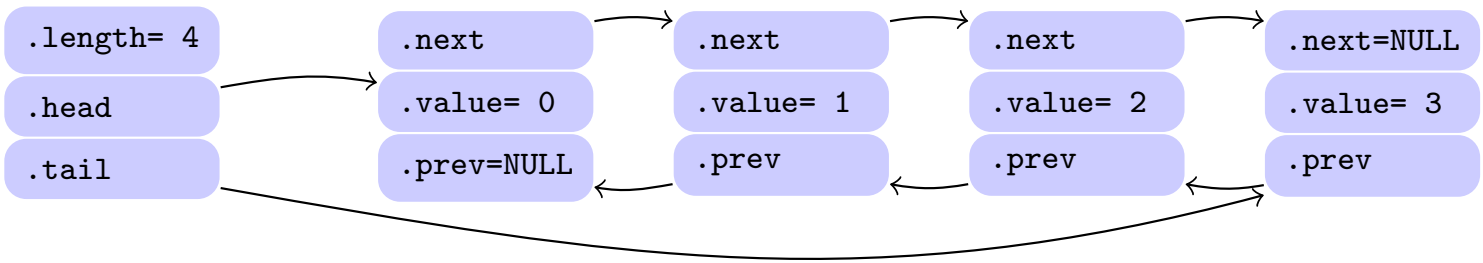
Variante 2 : liste simplement chaînée avec accès à la fin

Listes chaînées

Dans une liste chaînée, les valeurs ne sont pas stockées
dans des cases contiguës comme dans un tableau,
mais dans des maillons séparés, chaînés entre eux par pointeurs.

On perd l'accès direct aux données par indexation,
mais on peut insérer ou supprimer une donnée devant d'autres
sans avoir à décaler en mémoire les données suivantes.
La capacité devient également illimitée.

Voici 4 variantes de chaînages pour la liste `(0, 1, 2, 3)` :



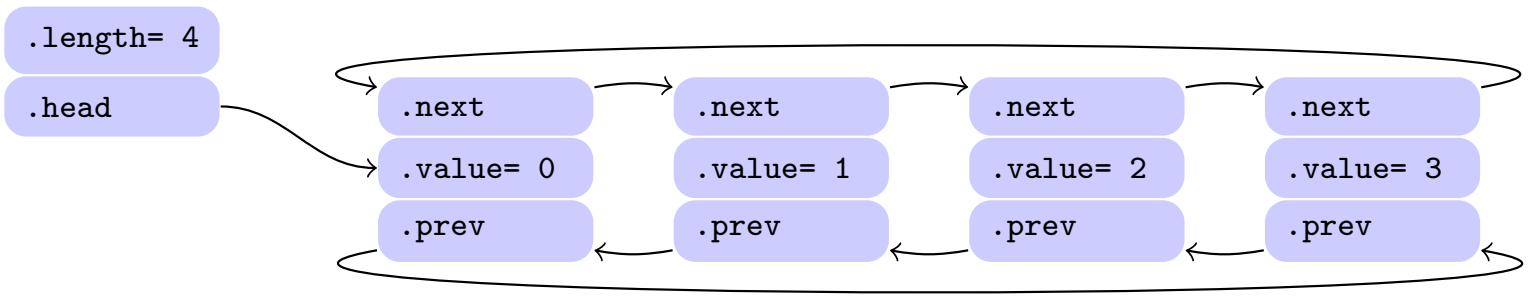
Variante 3 : liste doublement chaînée

Listes chaînées

Dans une liste chaînée, les valeurs ne sont pas stockées
dans des cases contiguës comme dans un tableau,
mais dans des maillons séparés, chaînés entre eux par pointeurs.

On perd l'accès direct aux données par indexation,
mais on peut insérer ou supprimer une donnée devant d'autres
sans avoir à décaler en mémoire les données suivantes.
La capacité devient également illimitée.

Voici 4 variantes de chaînages pour la liste `(0, 1, 2, 3)` :



Variante 4 : liste doublement chaînée circulaire

Variante 1 : Liste simplement chaînée

Dans cette variante, on a seulement accès `head` au début :

```
typedef struct Link {
  struct Link * next;
  int value;
} Link;
```

```
typedef struct List {
  int length;
  Link * head;
} List;
```

Création de `(1, 2, 3, 4)` par 4 insertions au début :

```
int main (void) {
  List list;
  List_Init (& list);
  List_AddFirst (& list, 3);
  List_AddFirst (& list, 2);
  List_AddFirst (& list, 1);
  List_AddFirst (& list, 0);
  List_Empty (& list);
  return 0;
}
```

Variante 1 : Liste simplement chaînée

Dans cette variante, on a seulement accès `head` au début :

```
typedef struct Link {
  struct Link * next;
  int value;
} Link;
```

```
typedef struct List {
  int length;
  Link * head;
} List;
```

Création de `(1, 2, 3, 4)` par 4 insertions au début :

```
int main (void) {
  List list;
  List_Init (& list);
  List_AddFirst (& list, 3);
  List_AddFirst (& list, 2);
  List_AddFirst (& list, 1);
  List_AddFirst (& list, 0);
  List_Empty (& list);
  return 0;
}
```

On a l'évolution suivante pour l'état de `list` :

`.length=??`

`.head= ??`

Variante 1 : Liste simplement chaînée

Dans cette variante, on a seulement accès `head` au début :

```
typedef struct Link {
  struct Link * next;
  int value;
} Link;
```

```
typedef struct List {
  int length;
  Link * head;
} List;
```

Création de `(1, 2, 3, 4)` par 4 insertions au début :

```
int main (void) {
  List list;
  List_Init (& list);
  List_AddFirst (& list, 3);
  List_AddFirst (& list, 2);
  List_AddFirst (& list, 1);
  List_AddFirst (& list, 0);
  List_Empty (& list);
  return 0;
}
```

On a l'évolution suivante pour l'état de `list` :

`.length= 0`

`.head=NULL`

Variante 1 : Liste simplement chaînée

Dans cette variante, on a seulement accès `head` au début :

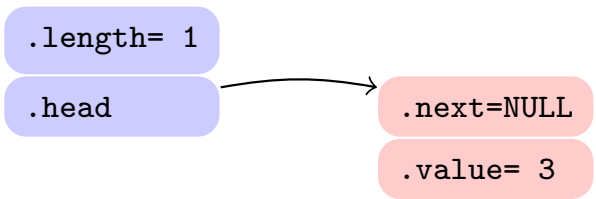
```
typedef struct Link {
  struct Link * next;
  int value;
} Link;
```

```
typedef struct List {
  int length;
  Link * head;
} List;
```

Création de `(1, 2, 3, 4)` par 4 insertions au début :

```
int main (void) {
  List list;
  List_Init (& list);
  List_AddFirst (& list, 3);
  List_AddFirst (& list, 2);
  List_AddFirst (& list, 1);
  List_AddFirst (& list, 0);
  List_Empty (& list);
  return 0;
}
```

On a l'évolution suivante pour l'état de `list` :



Variante 1 : Liste simplement chaînée

Dans cette variante, on a seulement accès `head` au début :

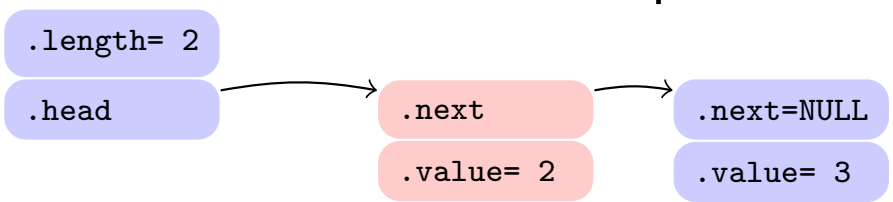
```
typedef struct Link {
  struct Link * next;
  int value;
} Link;
```

```
typedef struct List {
  int length;
  Link * head;
} List;
```

Création de `(1, 2, 3, 4)` par 4 insertions au début :

```
int main (void) {
  List list;
  List_Init (& list);
  List_AddFirst (& list, 3);
  List_AddFirst (& list, 2);
  List_AddFirst (& list, 1);
  List_AddFirst (& list, 0);
  List_Empty (& list);
  return 0;
}
```

On a l'évolution suivante pour l'état de `list` :



Variante 1 : Liste simplement chaînée

Dans cette variante, on a seulement accès `head` au début :

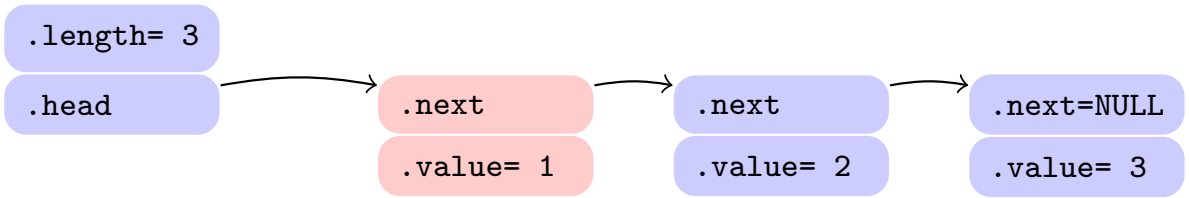
```
typedef struct Link {
  struct Link * next;
  int value;
} Link;
```

```
typedef struct List {
  int length;
  Link * head;
} List;
```

Création de `(1, 2, 3, 4)` par 4 insertions au début :

```
int main (void) {
  List list;
  List_Init (& list);
  List_AddFirst (& list, 3);
  List_AddFirst (& list, 2);
  List_AddFirst (& list, 1);
  List_AddFirst (& list, 0);
  List_Empty (& list);
  return 0;
}
```

On a l'évolution suivante pour l'état de `list` :



Variante 1 : Liste simplement chaînée

Dans cette variante, on a seulement accès `head` au début :

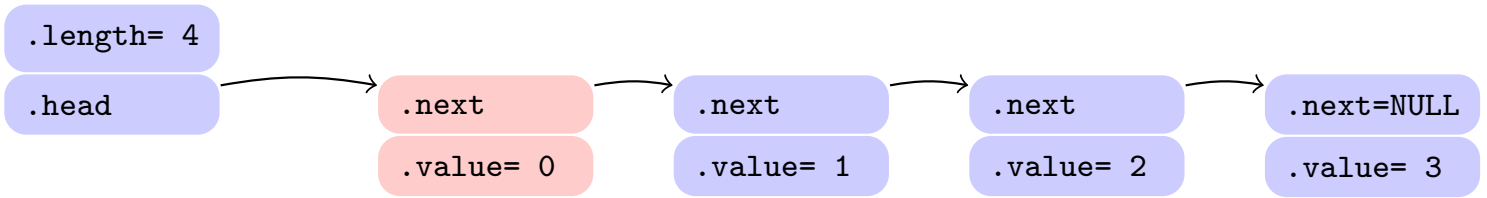
```
typedef struct Link {
  struct Link * next;
  int value;
} Link;
```

```
typedef struct List {
  int length;
  Link * head;
} List;
```

Création de `(1, 2, 3, 4)` par 4 insertions au début :

```
int main (void) {
  List list;
  List_Init (& list);
  List_AddFirst (& list, 3);
  List_AddFirst (& list, 2);
  List_AddFirst (& list, 1);
  List_AddFirst (& list, 0);
  List_Empty (& list);
  return 0;
}
```

On a l'évolution suivante pour l'état de `list` :



Variante 1 : Liste simplement chaînée

Dans cette variante, on a seulement accès `head` au début :

```
typedef struct Link {
  struct Link * next;
  int value;
} Link;
```

```
typedef struct List {
  int length;
  Link * head;
} List;
```

Création de `(1, 2, 3, 4)` par 4 insertions au début :

```
int main (void) {
  List list;
  List_Init (& list);
  List_AddFirst (& list, 3);
  List_AddFirst (& list, 2);
  List_AddFirst (& list, 1);
  List_AddFirst (& list, 0);
  List_Empty (& list);
  return 0;
}
```

On a l'évolution suivante pour l'état de `list` :

`.length= 0`

`.head=NULL`

Variante 2 : Liste simplement chaînée avec accès à la fin

Dans cette variante, la liste a aussi un accès `tail` à la fin :

```
typedef struct Link {
  struct Link * next;
  int value;
} Link;
```

```
typedef struct List {
  int length;
  Link * head, * tail;
} List;
```

Création de `(1, 2, 3, 4)` par 4 insertions à la fin :

```
int main (void) {
  List list;
  List_Init (& list);
  List_AddLast (& list, 0);
  List_AddLast (& list, 1);
  List_AddLast (& list, 2);
  List_AddLast (& list, 3);
  List_Empty (& list);
  return 0;
}
```

Variante 2 : Liste simplement chaînée avec accès à la fin

Dans cette variante, la liste a aussi un accès `tail` à la fin :

```
typedef struct Link {
  struct Link * next;
  int value;
} Link;
```

```
typedef struct List {
  int length;
  Link * head, * tail;
} List;
```

Création de `(1, 2, 3, 4)` par 4 insertions à la fin :

```
int main (void) {
  List list;
  List_Init (& list);
  List_AddLast (& list, 0);
  List_AddLast (& list, 1);
  List_AddLast (& list, 2);
  List_AddLast (& list, 3);
  List_Empty (& list);
  return 0;
}
```

On a l'évolution suivante pour l'état de `list` :

`.length=??`

`.head= ??`

`.tail= ??`

Variante 2 : Liste simplement chaînée avec accès à la fin

Dans cette variante, la liste a aussi un accès `tail` à la fin :

```
typedef struct Link {
  struct Link * next;
  int value;
} Link;
```

```
typedef struct List {
  int length;
  Link * head, * tail;
} List;
```

Création de `(1, 2, 3, 4)` par 4 insertions à la fin :

```
int main (void) {
  List list;
  List_Init (& list);
  List_AddLast (& list, 0);
  List_AddLast (& list, 1);
  List_AddLast (& list, 2);
  List_AddLast (& list, 3);
  List_Empty (& list);
  return 0;
}
```

On a l'évolution suivante pour l'état de `list` :

`.length= 0`

`.head=NULL`

`.tail=NULL`

Variante 2 : Liste simplement chaînée avec accès à la fin

Dans cette variante, la liste a aussi un accès `tail` à la fin :

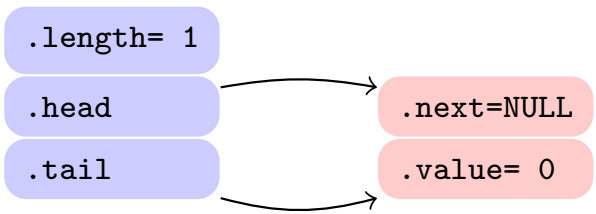
```
typedef struct Link {
  struct Link * next;
  int value;
} Link;
```

```
typedef struct List {
  int length;
  Link * head, * tail;
} List;
```

Création de `(1, 2, 3, 4)` par 4 insertions à la fin :

```
int main (void) {
  List list;
  List_Init (& list);
  List_AddLast (& list, 0);
  List_AddLast (& list, 1);
  List_AddLast (& list, 2);
  List_AddLast (& list, 3);
  List_Empty (& list);
  return 0;
}
```

On a l'évolution suivante pour l'état de `list` :



Variante 2 : Liste simplement chaînée avec accès à la fin

Dans cette variante, la liste a aussi un accès `tail` à la fin :

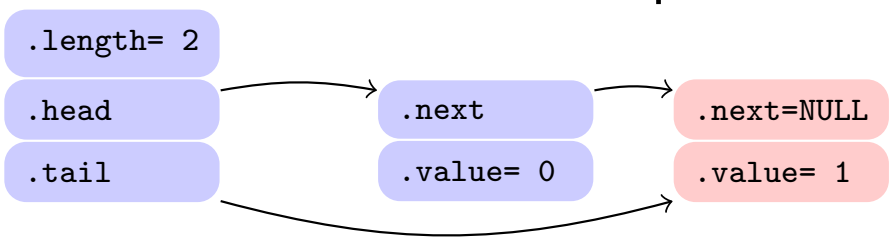
```
typedef struct Link {
  struct Link * next;
  int value;
} Link;
```

```
typedef struct List {
  int length;
  Link * head, * tail;
} List;
```

Création de `(1, 2, 3, 4)` par 4 insertions à la fin :

```
int main (void) {
  List list;
  List_Init (& list);
  List_AddLast (& list, 0);
  List_AddLast (& list, 1);
  List_AddLast (& list, 2);
  List_AddLast (& list, 3);
  List_Empty (& list);
  return 0;
}
```

On a l'évolution suivante pour l'état de `list` :



Variante 2 : Liste simplement chaînée avec accès à la fin

Dans cette variante, la liste a aussi un accès `tail` à la fin :

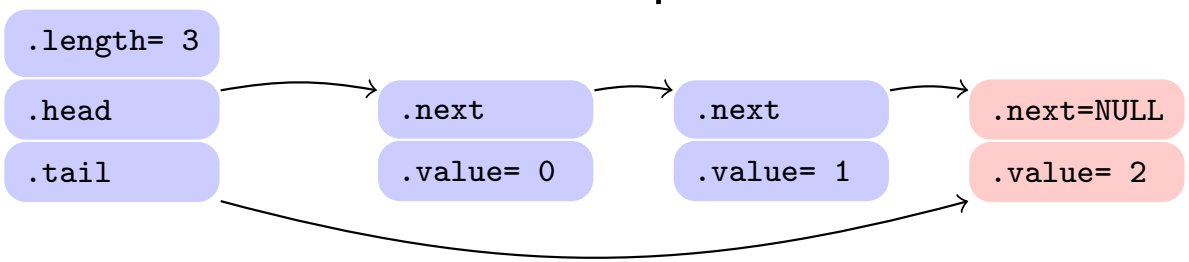
```
typedef struct Link {
  struct Link * next;
  int value;
} Link;
```

```
typedef struct List {
  int length;
  Link * head, * tail;
} List;
```

Création de `(1, 2, 3, 4)` par 4 insertions à la fin :

```
int main (void) {
  List list;
  List_Init (& list);
  List_AddLast (& list, 0);
  List_AddLast (& list, 1);
  List_AddLast (& list, 2);
  List_AddLast (& list, 3);
  List_Empty (& list);
  return 0;
}
```

On a l'évolution suivante pour l'état de `list` :



Variante 2 : Liste simplement chaînée avec accès à la fin

Dans cette variante, la liste a aussi un accès `tail` à la fin :

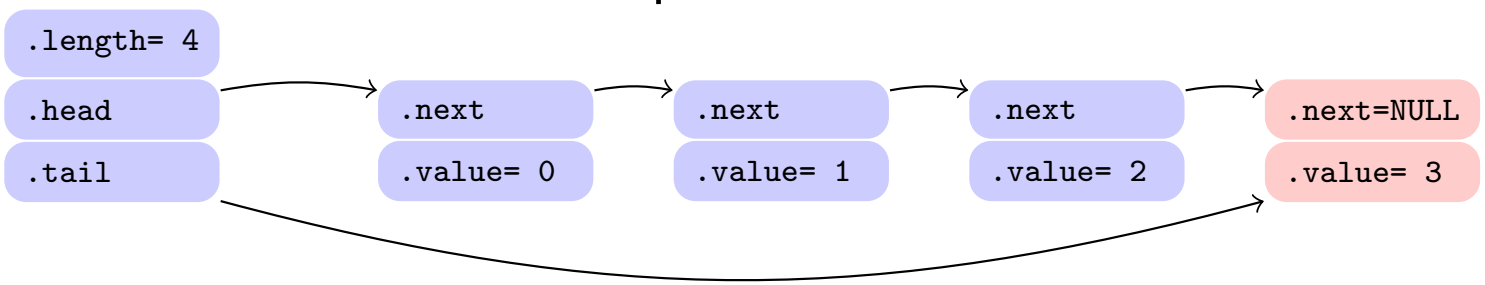
```
typedef struct Link {
  struct Link * next;
  int value;
} Link;
```

```
typedef struct List {
  int length;
  Link * head, * tail;
} List;
```

Création de `(1, 2, 3, 4)` par 4 insertions à la fin :

```
int main (void) {
  List list;
  List_Init (& list);
  List_AddLast (& list, 0);
  List_AddLast (& list, 1);
  List_AddLast (& list, 2);
  List_AddLast (& list, 3);
  List_Empty (& list);
  return 0;
}
```

On a l'évolution suivante pour l'état de `list` :



Variante 2 : Liste simplement chaînée avec accès à la fin

Dans cette variante, la liste a aussi un accès `tail` à la fin :

```
typedef struct Link {
  struct Link * next;
  int value;
} Link;
```

```
typedef struct List {
  int length;
  Link * head, * tail;
} List;
```

Création de `(1, 2, 3, 4)` par 4 insertions à la fin :

```
int main (void) {
  List list;
  List_Init (& list);
  List_AddLast (& list, 0);
  List_AddLast (& list, 1);
  List_AddLast (& list, 2);
  List_AddLast (& list, 3);
  List_Empty (& list);
  return 0;
}
```

On a l'évolution suivante pour l'état de `list` :

`.length= 0`

`.head=NULL`

`.tail=NULL`

Variante 3 : Liste doublement chaînée

Dans cette variante, les maillons ont aussi un accès arrière `prev` :

```
typedef struct Link {
  struct Link * next, * prev;
  int value;
} Link;
```

```
typedef struct List {
  int length;
  Link * head, * tail;
} List;
```

Création de `(1, 2, 3, 4)` par 4 insertions à la fin :

```
int main (void) {
  List list;
  List_Init (& list);
  List_AddLast (& list, 0);
  List_AddLast (& list, 1);
  List_AddLast (& list, 2);
  List_AddLast (& list, 3);
  List_Empty (& list);
  return 0;
}
```

Variante 3 : Liste doublement chaînée

Dans cette variante, les maillons ont aussi un accès arrière `prev` :

```
typedef struct Link {
  struct Link * next, * prev;
  int value;
} Link;
```

```
typedef struct List {
  int length;
  Link * head, * tail;
} List;
```

Création de `(1, 2, 3, 4)` par 4 insertions à la fin :

```
int main (void) {
  List list;
  List_Init (& list);
  List_AddLast (& list, 0);
  List_AddLast (& list, 1);
  List_AddLast (& list, 2);
  List_AddLast (& list, 3);
  List_Empty (& list);
  return 0;
}
```

On a l'évolution suivante pour l'état de `list` :

`.length=??`

`.head= ??`

`.tail= ??`

Variante 3 : Liste doublement chaînée

Dans cette variante, les maillons ont aussi un accès arrière `prev` :

```
typedef struct Link {
  struct Link * next, * prev;
  int value;
} Link;
```

```
typedef struct List {
  int length;
  Link * head, * tail;
} List;
```

Création de `(1, 2, 3, 4)` par 4 insertions à la fin :

```
int main (void) {
  List list;
  List_Init (& list);
  List_AddLast (& list, 0);
  List_AddLast (& list, 1);
  List_AddLast (& list, 2);
  List_AddLast (& list, 3);
  List_Empty (& list);
  return 0;
}
```

On a l'évolution suivante pour l'état de `list` :

`.length= 0`

`.head=NULL`

`.tail=NULL`

Variante 3 : Liste doublement chaînée

Dans cette variante, les maillons ont aussi un accès arrière `prev` :

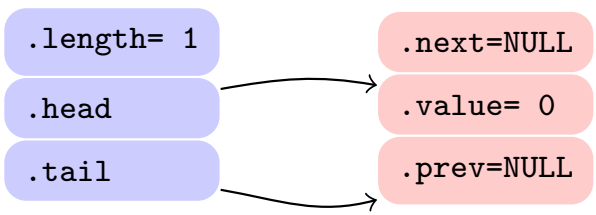
```
typedef struct Link {
  struct Link * next, * prev;
  int value;
} Link;
```

```
typedef struct List {
  int length;
  Link * head, * tail;
} List;
```

Création de `(1, 2, 3, 4)` par 4 insertions à la fin :

```
int main (void) {
  List list;
  List_Init (& list);
  List_AddLast (& list, 0);
  List_AddLast (& list, 1);
  List_AddLast (& list, 2);
  List_AddLast (& list, 3);
  List_Empty (& list);
  return 0;
}
```

On a l'évolution suivante pour l'état de `list` :



Variante 3 : Liste doublement chaînée

Dans cette variante, les maillons ont aussi un accès arrière `prev` :

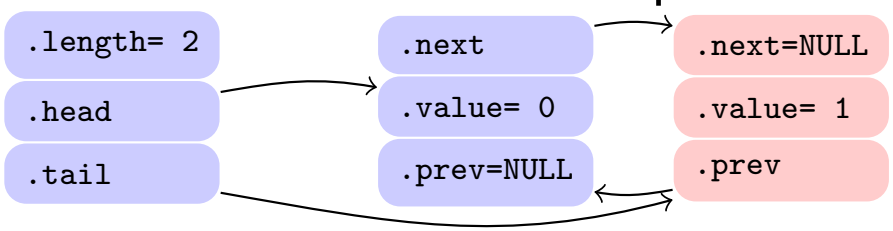
```
typedef struct Link {
  struct Link * next, * prev;
  int value;
} Link;
```

```
typedef struct List {
  int length;
  Link * head, * tail;
} List;
```

Création de `(1, 2, 3, 4)` par 4 insertions à la fin :

```
int main (void) {
  List list;
  List_Init (& list);
  List_AddLast (& list, 0);
  List_AddLast (& list, 1);
  List_AddLast (& list, 2);
  List_AddLast (& list, 3);
  List_Empty (& list);
  return 0;
}
```

On a l'évolution suivante pour l'état de `list` :



Variante 3 : Liste doublement chaînée

Dans cette variante, les maillons ont aussi un accès arrière `prev` :

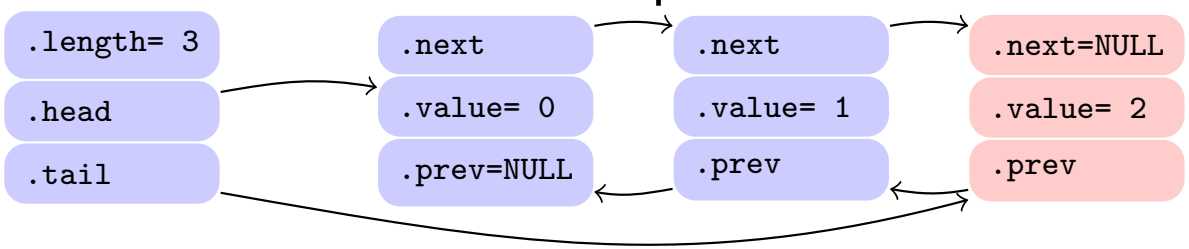
```
typedef struct Link {
  struct Link * next, * prev;
  int value;
} Link;
```

```
typedef struct List {
  int length;
  Link * head, * tail;
} List;
```

Création de `(1, 2, 3, 4)` par 4 insertions à la fin :

```
int main (void) {
  List list;
  List_Init (& list);
  List_AddLast (& list, 0);
  List_AddLast (& list, 1);
  List_AddLast (& list, 2);
  List_AddLast (& list, 3);
  List_Empty (& list);
  return 0;
}
```

On a l'évolution suivante pour l'état de `list` :



Variante 3 : Liste doublement chaînée

Dans cette variante, les maillons ont aussi un accès arrière `prev` :

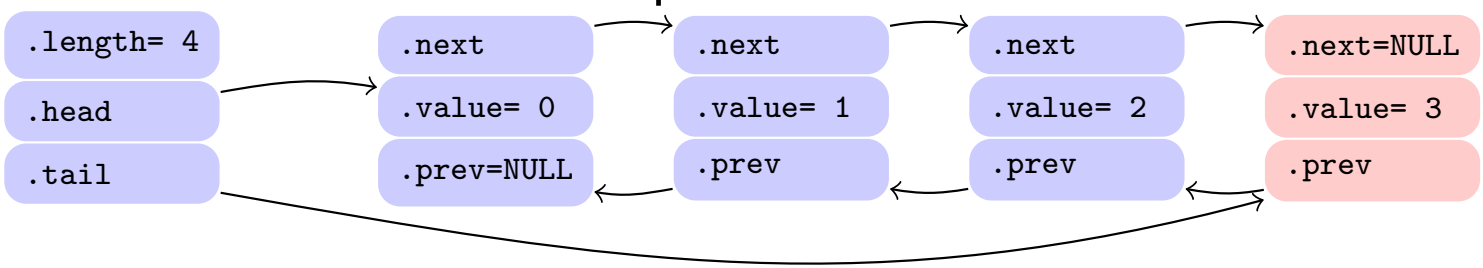
```
typedef struct Link {
  struct Link * next, * prev;
  int value;
} Link;
```

```
typedef struct List {
  int length;
  Link * head, * tail;
} List;
```

Création de `(1, 2, 3, 4)` par 4 insertions à la fin :

```
int main (void) {
  List list;
  List_Init (& list);
  List_AddLast (& list, 0);
  List_AddLast (& list, 1);
  List_AddLast (& list, 2);
  List_AddLast (& list, 3);
  List_Empty (& list);
  return 0;
}
```

On a l'évolution suivante pour l'état de `list` :



Variante 3 : Liste doublement chaînée

Dans cette variante, les maillons ont aussi un accès arrière `prev` :

```
typedef struct Link {
  struct Link * next, * prev;
  int value;
} Link;
```

```
typedef struct List {
  int length;
  Link * head, * tail;
} List;
```

Création de `(1, 2, 3, 4)` par 4 insertions à la fin :

```
int main (void) {
  List list;
  List_Init (& list);
  List_AddLast (& list, 0);
  List_AddLast (& list, 1);
  List_AddLast (& list, 2);
  List_AddLast (& list, 3);
  List_Empty (& list);
  return 0;
}
```

On a l'évolution suivante pour l'état de `list` :

- `.length= 0`
- `.head=NULL`
- `.tail=NULL`

Variante 4 : Liste doublement chaînée circulaire

Dans cette variante, on chaîne le premier maillon au dernier :

```
typedef struct Link {
  struct Link * next, * prev;
  int value;
} Link;
```

```
typedef struct List {
  int length;
  Link * head;
} List;
```

Création de (1, 2, 3, 4) par 4 insertions à la fin :

```
int main (void) {
  List list;
  List_Init (& list);
  List_AddLast (& list, 0);
  List_AddLast (& list, 1);
  List_AddLast (& list, 2);
  List_AddLast (& list, 3);
  List_Empty (& list);
  return 0;
}
```

Variante 4 : Liste doublement chaînée circulaire

Dans cette variante, on chaîne le premier maillon au dernier :

```
typedef struct Link {
  struct Link * next, * prev;
  int value;
} Link;
```

```
typedef struct List {
  int length;
  Link * head;
} List;
```

Création de `(1, 2, 3, 4)` par 4 insertions à la fin :

```
int main (void) {
  List list;
  List_Init (& list);
  List_AddLast (& list, 0);
  List_AddLast (& list, 1);
  List_AddLast (& list, 2);
  List_AddLast (& list, 3);
  List_Empty (& list);
  return 0;
}
```

On a l'évolution suivante pour l'état de `list` :

`.length=??`

`.head= ??`

Variante 4 : Liste doublement chaînée circulaire

Dans cette variante, on chaîne le premier maillon au dernier :

```
typedef struct Link {
  struct Link * next, * prev;
  int value;
} Link;
```

```
typedef struct List {
  int length;
  Link * head;
} List;
```

Création de `(1, 2, 3, 4)` par 4 insertions à la fin :

```
int main (void) {
  List list;
  List_Init (& list);
  List_AddLast (& list, 0);
  List_AddLast (& list, 1);
  List_AddLast (& list, 2);
  List_AddLast (& list, 3);
  List_Empty (& list);
  return 0;
}
```

On a l'évolution suivante pour l'état de `list` :

`.length= 0`

`.head=NULL`

Variante 4 : Liste doublement chaînée circulaire

Dans cette variante, on chaîne le premier maillon au dernier :

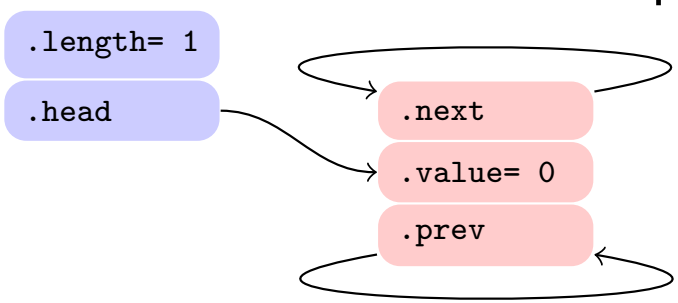
```
typedef struct Link {
  struct Link * next, * prev;
  int value;
} Link;
```

```
typedef struct List {
  int length;
  Link * head;
} List;
```

Création de `(1, 2, 3, 4)` par 4 insertions à la fin :

```
int main (void) {
  List list;
  List_Init (& list);
  List_AddLast (& list, 0);
  List_AddLast (& list, 1);
  List_AddLast (& list, 2);
  List_AddLast (& list, 3);
  List_Empty (& list);
  return 0;
}
```

On a l'évolution suivante pour l'état de `list` :



Variante 4 : Liste doublement chaînée circulaire

Dans cette variante, on chaîne le premier maillon au dernier :

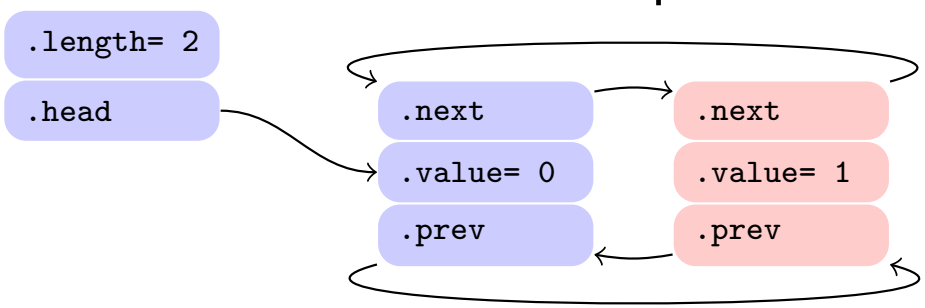
```
typedef struct Link {
  struct Link * next, * prev;
  int value;
} Link;
```

```
typedef struct List {
  int length;
  Link * head;
} List;
```

Création de `(1, 2, 3, 4)` par 4 insertions à la fin :

```
int main (void) {
  List list;
  List_Init (& list);
  List_AddLast (& list, 0);
  List_AddLast (& list, 1);
  List_AddLast (& list, 2);
  List_AddLast (& list, 3);
  List_Empty (& list);
  return 0;
}
```

On a l'évolution suivante pour l'état de `list` :



Variante 4 : Liste doublement chaînée circulaire

Dans cette variante, on chaîne le premier maillon au dernier :

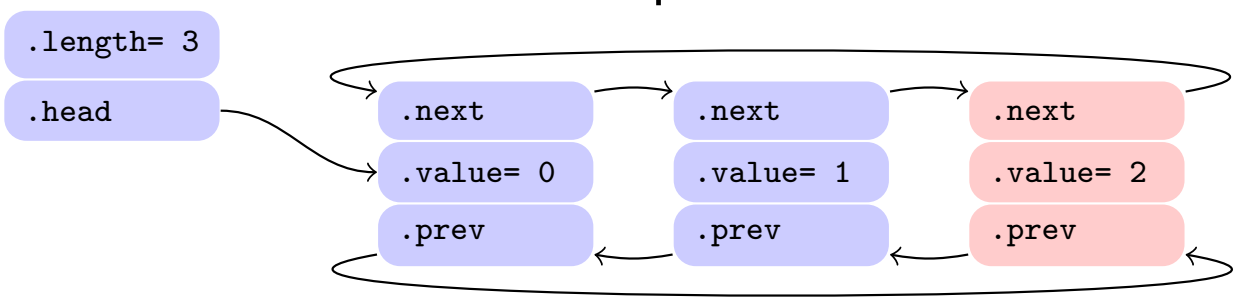
```
typedef struct Link {
  struct Link * next, * prev;
  int value;
} Link;
```

```
typedef struct List {
  int length;
  Link * head;
} List;
```

Création de `(1, 2, 3, 4)` par 4 insertions à la fin :

```
int main (void) {
  List list;
  List_Init (& list);
  List_AddLast (& list, 0);
  List_AddLast (& list, 1);
  List_AddLast (& list, 2);
  List_AddLast (& list, 3);
  List_Empty (& list);
  return 0;
}
```

On a l'évolution suivante pour l'état de `list` :



Variante 4 : Liste doublement chaînée circulaire

Dans cette variante, on chaîne le premier maillon au dernier :

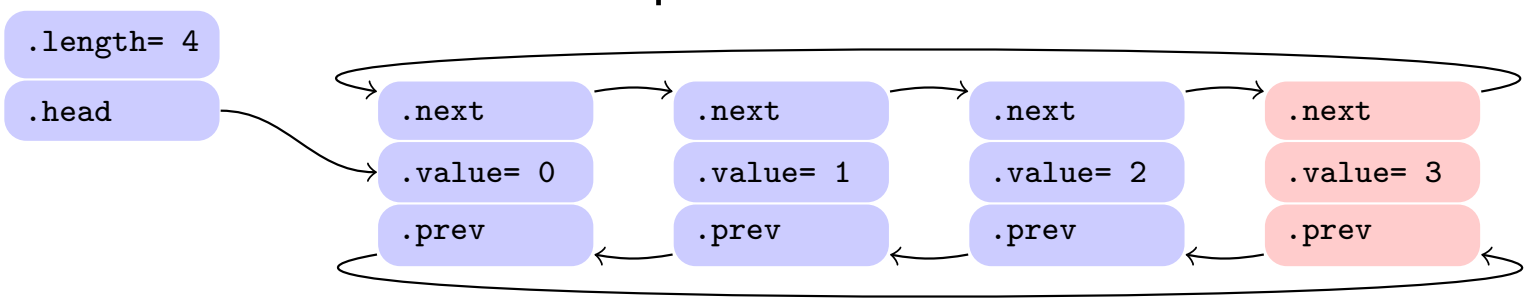
```
typedef struct Link {
  struct Link * next, * prev;
  int value;
} Link;
```

```
typedef struct List {
  int length;
  Link * head;
} List;
```

Création de `(1, 2, 3, 4)` par 4 insertions à la fin :

```
int main (void) {
  List list;
  List_Init (& list);
  List_AddLast (& list, 0);
  List_AddLast (& list, 1);
  List_AddLast (& list, 2);
  List_AddLast (& list, 3);  •
  List_Empty (& list);
  return 0;
}
```

On a l'évolution suivante pour l'état de `list` :



Variante 4 : Liste doublement chaînée circulaire

Dans cette variante, on chaîne le premier maillon au dernier :

```
typedef struct Link {
  struct Link * next, * prev;
  int value;
} Link;
```

```
typedef struct List {
  int length;
  Link * head;
} List;
```

Création de `(1, 2, 3, 4)` par 4 insertions à la fin :

```
int main (void) {
  List list;
  List_Init (& list);
  List_AddLast (& list, 0);
  List_AddLast (& list, 1);
  List_AddLast (& list, 2);
  List_AddLast (& list, 3);
  List_Empty (& list);
  return 0;
}
```

On a l'évolution suivante pour l'état de `list` :

`.length= 0`

`.head=NULL`

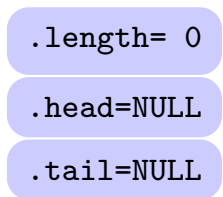
Implémentation de liste chaînée (variante 2)

On se donne les structures suivante pour les maillons et la liste :

```c
typedef struct Link {
  struct Link * next;
  int value;
} Link;
```

```c
typedef struct List {
  int length;
  Link * head, * tail;
} List;
```

La liste vide a la forme suivante :



On en déduit `List_Init()` qui initialise une liste comme vide :

```c
void List_Init (List * self) {
  self->length= 0;
  self->head= self->tail= NULL;
}
```

ainsi que `List_IsEmpty()` testant qu'une liste est vide :

```c
bool List_IsEmpty (List const * self) {
  return self->length == 0;
}
```

Implémentation de liste chaînée : création de maillon

Pour insérer un élément, il faut préalablement créer un maillon par allocation dynamique pour héberger l'élément, ce que fait `Link_Create()` :

```
Link * Link_Create (int value, Link const * next) {
  Link * self= malloc (sizeof * self);
  assert(self != NULL);
  self->value= value;
  self->next= (Link *) next;
  return self;
}
```

dont le destructeur associé est `Link_Destroy()` :

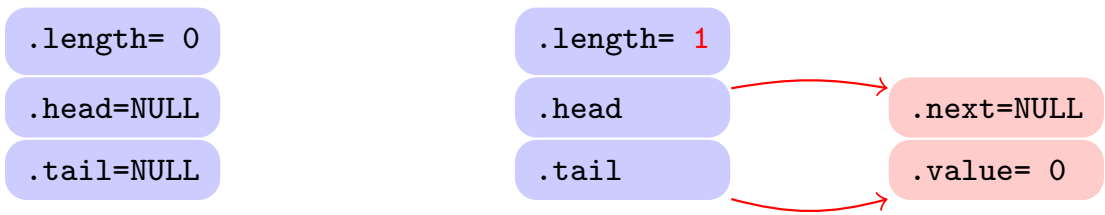
```
void Link_Destroy (Link * self) {
  free (self);
}
```

Implémentation de liste chaînée : insertion au début

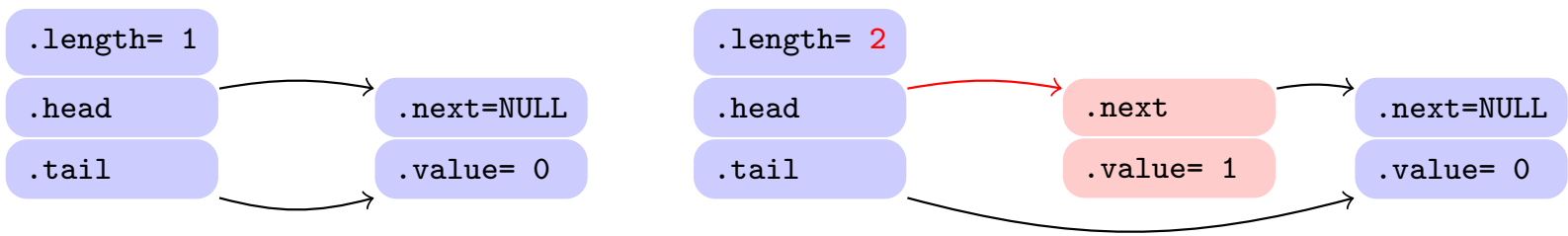
L'insertion au début `List_AddFirst()` s'écrit alors :

```
void List_AddFirst (List * self, int value) {
  Link * new_link= Link_Create (value, self->head);
  if (self->length == 0) self->tail= new_link;

  self->head= new_link;
  self->length++;
}
```

Le code distingue l'insertion dans la liste vide, où `tail` change :



Pour une liste non-vide, `tail` ne change pas :

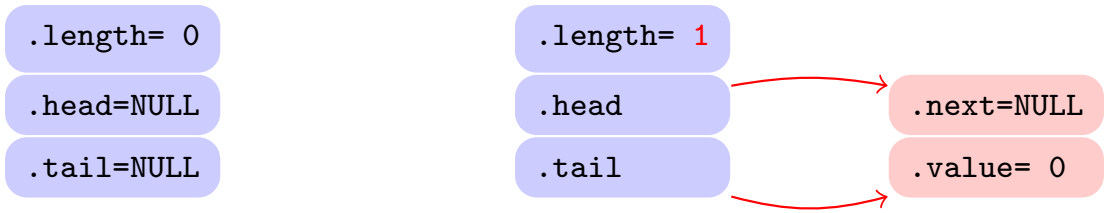


Implémentation de liste chaînée : insertion à la fin

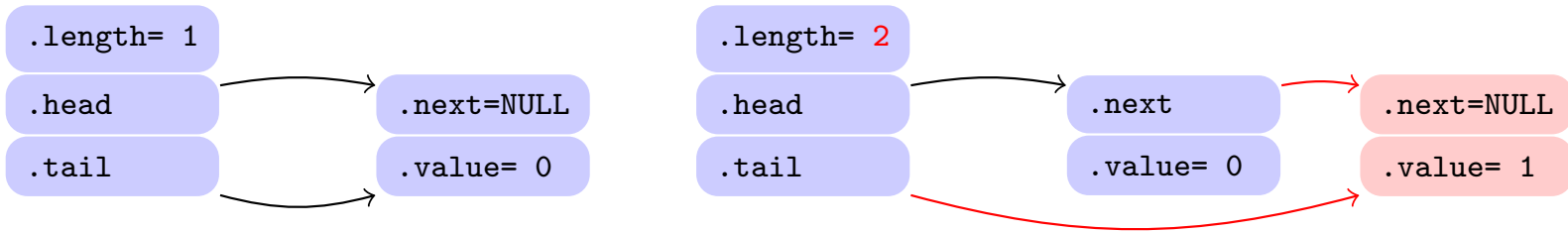
L'insertion à la fin `List_AddLast()` s'écrit :

```
void List_AddLast (List * self, int value) {
  Link * new_link= Link_Create (value, NULL);
  if (self->length == 0) self->head= new_link;
  else                   self->tail->next= new_link;

  self->tail= new_link;
  self->length++;
}
```

Le code distingue l'insertion dans la liste vide, où `head` change :



Pour une liste non-vide, c'est `tail->next` qui change :

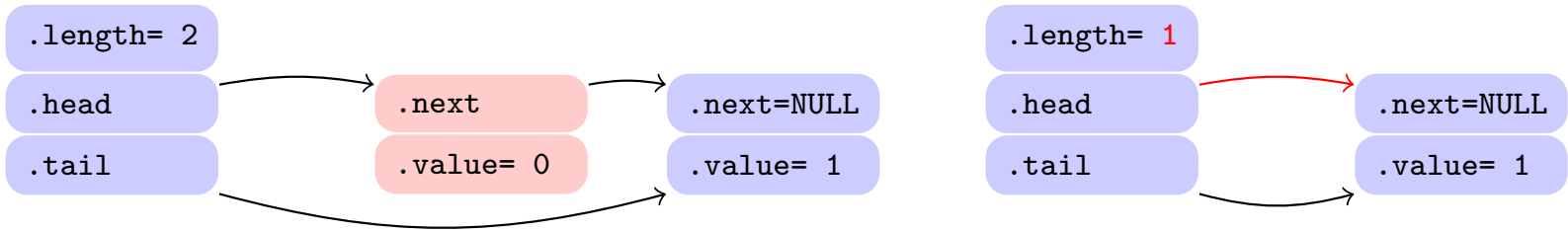


Implémentation de liste chaînée : suppression au début

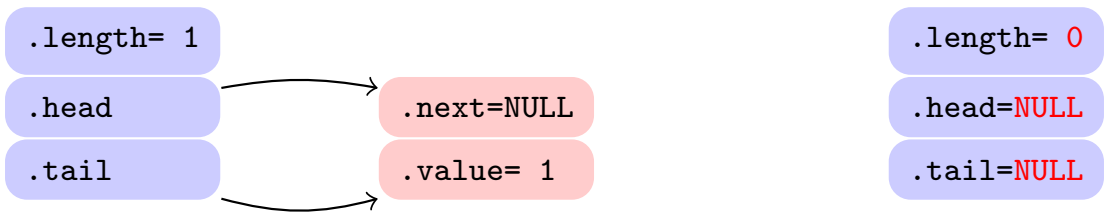
La suppression au début `RemoveFirst()` s'écrit :

```
int List_RemoveFirst (List * self) {
  assert ( ! List_IsEmpty (self));
  Link * removed_head= self->head;
  self->length --;
  self->head= removed_head->next;
  if (self->length == 0) self->tail= NULL;
  int removed_value= removed_head->value;
  Link_Destroy (removed_head);
  return removed_value;
}
```

`tail` ne change pas lorsque la liste ne devient pas vide :



`tail` change lorsque la liste devient vide :



Implémentation de liste chaînée : tests

On peut tester les ajouts en début et en fin comme suit :

```
void Test_List_AddFirst (void) {
  List l; List_Init (& l);
  List_AddFirst (& l, 2);
  List_AddFirst (& l, 1);
  List_AddFirst (& l, 0);
  assert(l.head->value              ==0);
  assert(l.head->next->value        ==1);
  assert(l.head->next->next->value==2);
  assert(l.tail->value              ==2);
  List_Clean (& l);
}
```

```
void Test_List_AddLast (void) {
  List l; List_Init (& l);
  List_AddLast (& l, 0);
  List_AddLast (& l, 1);
  List_AddLast (& l, 2);
  assert(l.head->value              ==0);
  assert(l.head->next->value        ==1);
  assert(l.head->next->next->value==2);
  assert(l.tail->value              ==2);
  List_Clean (& l);
}
```

et on peut tester la suppression au début comme suit :

```
void Test_List_RemoveFirst (void) {
  List list; List_Init (& list);
  List_AddLast (& list, 0);
  List_AddLast (& list, 1);
  List_AddLast (& list, 2);
  assert (List_RemoveFirst (& list) == 0);
  assert (List_RemoveFirst (& list) == 1);
  assert (List_RemoveFirst (& list) == 2);
  assert (List_IsEmpty (& list));
  List_Clean (& list);
}
```

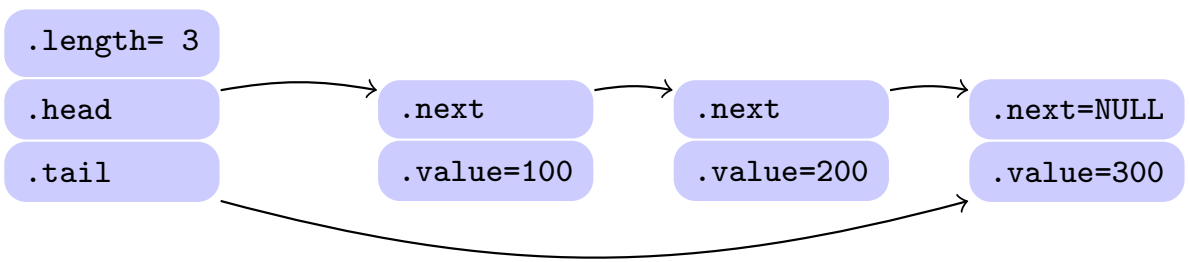
Il nous faut cependant `List_Clean()` qui parcourt les maillons.

Parcourir les maillons d'une liste chaînée (1/4)

Le getter `GetLinkAt()` est la forme de boucle la plus simple :

```
bool List_IndexIsValid (List const * self, int index) {
  return 0 <= index && index < self->length;
}

Link * List_GetLinkAt (List const * self, int index) {
  assert (List_IndexIsValid (self, index));
  Link * current= self->head;
  for (int k= 0; k < index; k++) current= current->next;
  return current;
}
```

```
void Test_List_GetLinkAt (void) {
  List list; List_Init (& list);
  List_AddLast (& list, 100);
  List_AddLast (& list, 200);
  List_AddLast (& list, 300);
  assert (List_GetLinkAt (& list, 0)->value == 100);
  assert (List_GetLinkAt (& list, 1)->value == 200);
  assert (List_GetLinkAt (& list, 2)->value == 300);
  List_Clean (& list);
}
```



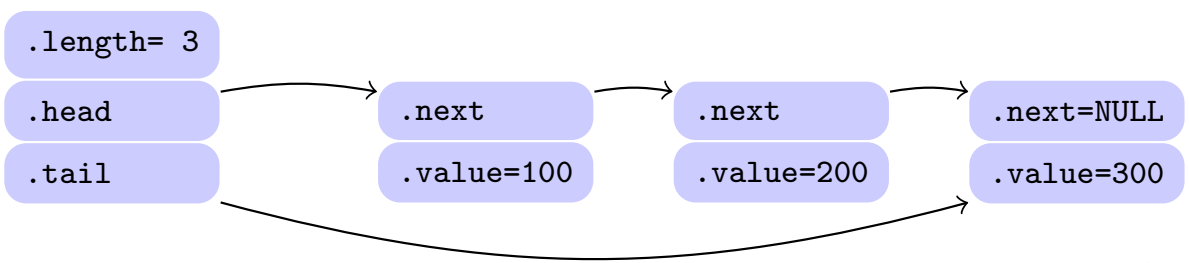
Parcourir les maillons d'une liste chaînée (2/4)

La forme de boucle classique est celle de la fonction d'affichage :

```
void List_Print (List const * self, FILE * file) {
  fprintf (file, "(");
  for (Link * current= self->head; current != NULL; current= current->next) {
     fprintf (file, "%d", current->value);
     if (current != self->tail) fprintf (file, ", ");
  }
  fprintf (file, ")");
}
```

```
int main (void) {
  List list; List_Init (& list);
  List_AddLast (& list, 100);
  List_AddLast (& list, 200);
  List_AddLast (& list, 300);
  List_Print (& list, stdout);
  fprintf (stdout, " length= %d\n", list.length);
  List_Clean (& list);
  return 0;
}
```

```
(100, 200, 300) length= 3
```

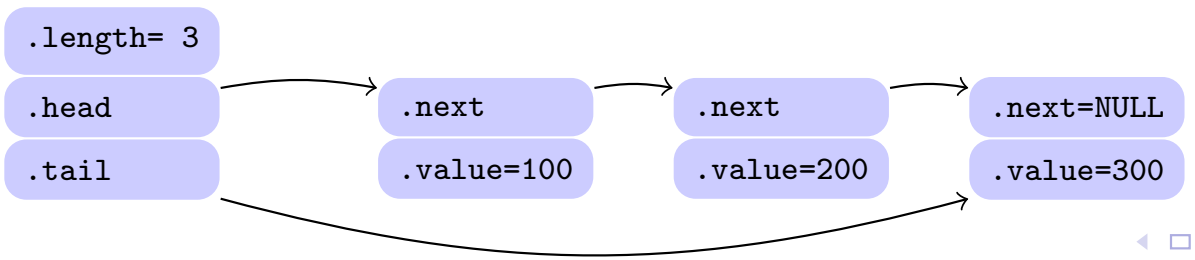


Parcourir les maillons d'une liste chaînée (3/4)

La forme de boucle de la fonction de recherche d'une valeur est similaire à celle de l'affichage, mais avec du *early-exit* :

```
Link * List_FindLinkOfValue (List const * self, int value) {
  for (Link * current= self->head; current != NULL; current= current->next)
    if (current->value == value) return current;
  return NULL;
}

bool List_Contains (List const * self, int value) {
  return List_FindLinkOfValue (self, value) != NULL;
}
```

```
void Test_List_FindLinkOfValue(void) {
  List list; List_Init (& list);
  List_AddLast (& list, 100);
  List_AddLast (& list, 200);
  List_AddLast (& list, 300);
  assert (List_FindLinkOfValue (& list, 100) == list.head);
  assert (List_FindLinkOfValue (& list, 200) == list.head->next);
  assert (List_FindLinkOfValue (& list, 300) == list.tail);
  assert (List_FindLinkOfValue (& list, 666) == NULL);
  List_Clean (& list);
}
```

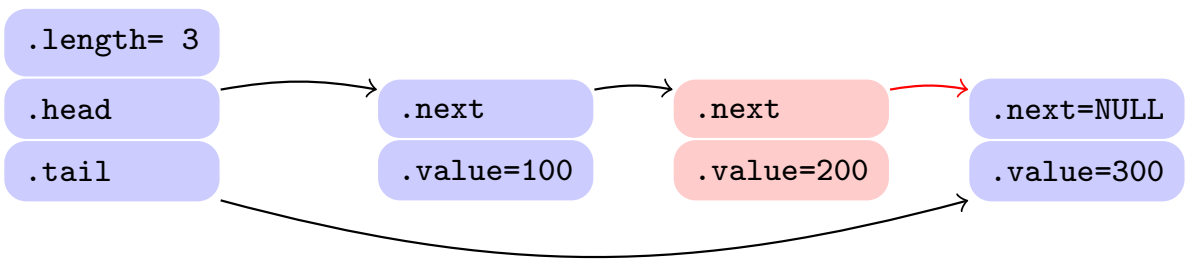


Parcourir les maillons d'une liste chaînée (4/4)

La boucle pour vider une liste est d'une forme particulière,
car on détruit les maillons en même temps qu'on les parcourt.

```
void List_Clean (List * self) {
  Link * next;
  for (Link * current= self->head; current != NULL; current= next) {
    next= current->next;
    Link_Destroy (current);
  }
}

void List_Empty (List * self) {
  List_Clean (self);
  List_Init  (self);
}
```

Il faut sauver `current->next` avant de détruire `* current`,
afin de pouvoir y sauter après la destruction de `* current`.



Listes chaînées : rassemblement des tests en batterie

Pour conclure sur les listes chaînées,
voici la batterie de tests pour le type `List` :

```
void Test_List (void) {
  Test_List_AddFirst();
  Test_List_AddLast();
  Test_List_RemoveFirst();
  Test_List_GetLinkAt();
  Test_List_FindLinkOfValue();
}
```

et le lancement de la batterie dans le programme principal :

```
void RunAllTests (void) {
  Test_VectorOfInt();
  Test_List();
}

int main (void) {
  RunAllTests();
  return 0;
}
```

Pile (Stack or LIFO) implémentée avec une liste chaînée

Toutes les variantes de listes permettent d'implémenter une pile (ou LIFO, Last In/First Out, le dernier entré est le 1er sorti) :

```
typedef struct Stack { List list; } Stack;
```

```
void Stack_Init   (Stack * self) { List_Init  (& self->list); }
void Stack_Clean  (Stack * self) { List_Clean (& self->list); }
void Stack_Empty  (Stack * self) { List_Empty (& self->list); }

bool Stack_IsEmpty(Stack * self) { return List_IsEmpty (& self->list); }
```

`Push()` empile et `Pop()` dépile :

```
void Stack_Push (Stack * self, int value) {
  List_AddFirst (& self->list, value);
}

int Stack_Pop (Stack * self) {
  assert ( ! Stack_IsEmpty (self));
  return List_RemoveFirst (& self->list);
}
```

`Peek()` consulte le sommet de la pile :

```
int Stack_Peek (Stack const * self) {
  assert ( ! Stack_IsEmpty (self));
  return self->list.head->value;
}
```

File (Queue or FIFO) implémentée avec une liste chaînée

La variante 2 des listes permet d'implémenter une file d'attente (ou FIFO, First-In/First-Out, le 1er entré est le 1er sorti) :

```
typedef struct Queue { List list; } Queue;
```

```
void Queue_Init   (Queue * self) { List_Init  (& self->list); }
void Queue_Clean  (Queue * self) { List_Clean (& self->list); }
void Queue_Empty  (Queue * self) { List_Empty (& self->list); }

bool Queue_IsEmpty(Queue * self) { return List_IsEmpty (& self->list); }
```

```
void Queue_Enqueue (Queue * self, int value) {
  List_AddLast (& self->list, value);
}

int Queue_Dequeue (Queue * self) {
  assert ( ! Queue_IsEmpty (self));
  return List_RemoveFirst (& self->list);
}
```

```
int Queue_Last (Queue const * self) {
  assert ( ! Queue_IsEmpty (self));
  return self->list.tail->value;
}

int Queue_First (Queue const * self) {
  assert ( ! Queue_IsEmpty (self));
  return self->list.head->value;
}
```