U.F.R SCIENCES ET TECHNIQUES
Département d'Informatique
B.P. 1155
64013 PAU CEDEX
Téléphone secrétariat : 05.59.40.79.64
Télécopie : 05.59.40.76.54

II- TYPES DES STRUCTURES LINEAIRES

I-Les structures linéaires
II-Type abstrait Vecteur
III-Type abstrait Liste
IV-Types abstraits Pile et File

I- LES STRUCTURES LINEAIRES

Dans une **structure linéaire** les objets sont disposés selon un certain **ordre**.

Mais, l'ordre est purement **séquentiel**:

- tout objet, sauf le dernier, a un **successeur**,
- tout objet, sauf le premier, a un **prédécesseur**.

Ainsi, dans une telle structure, on peut accéder **directement** aux objets à partir de leur **rang**.

De même, dans une telle structure, on ajoute un objet en spécifiant son **successeur ou prédécesseur.**

CAS D'UN TABLEAU

Un **tableau** est la forme la plus simple d'une structure linéaire.

Cependant, il présente l'inconvénient d'être une **structure statique**.

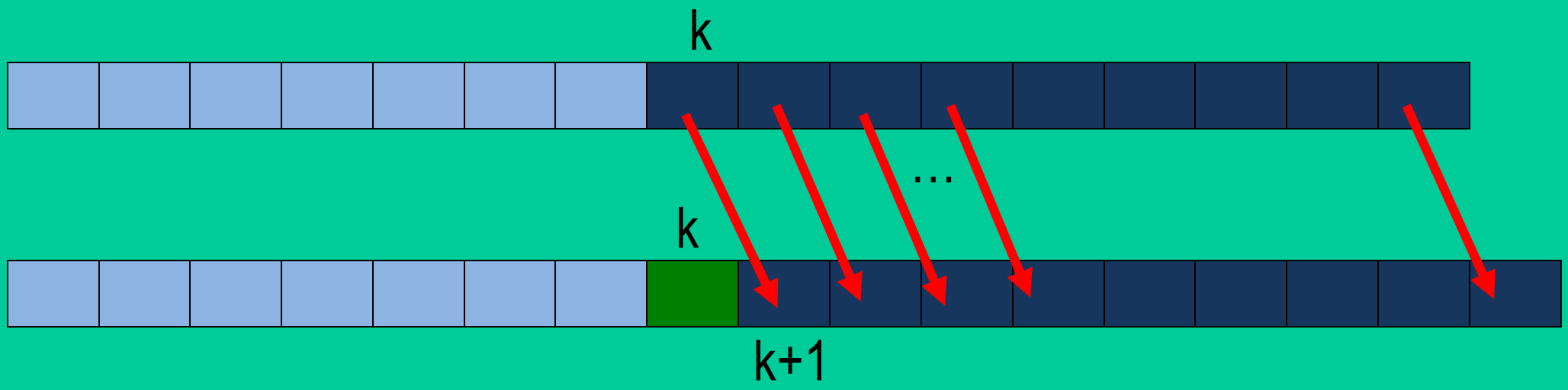
Ainsi, le tableau possède les **propriétés** suivantes:

i) c'est une structure de **taille fixe**,

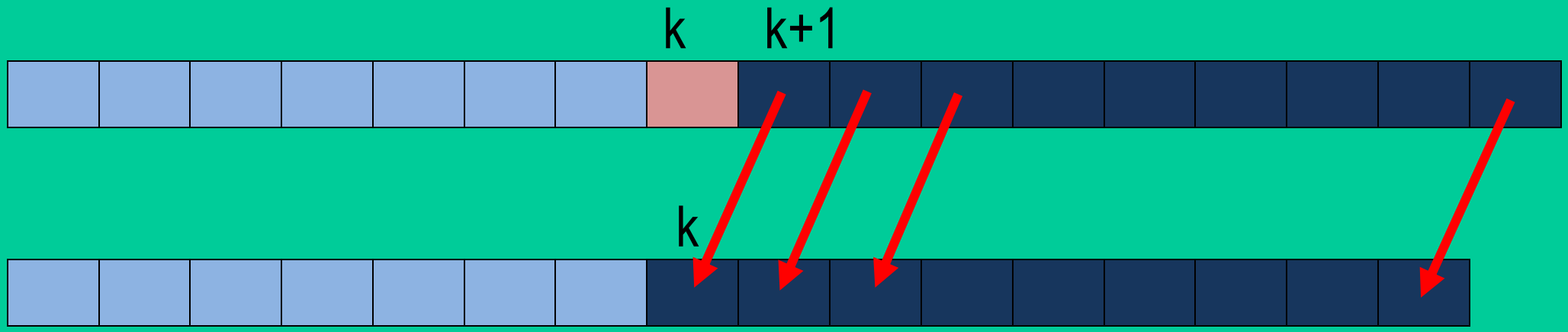
ii) on y accède aux objets par leur **indice** (**rang**),

iii) une **insertion** ou **suppression** d'objet provoque la modification des indices des objets d'indice supérieur.

Insertion en k



Suppression en k



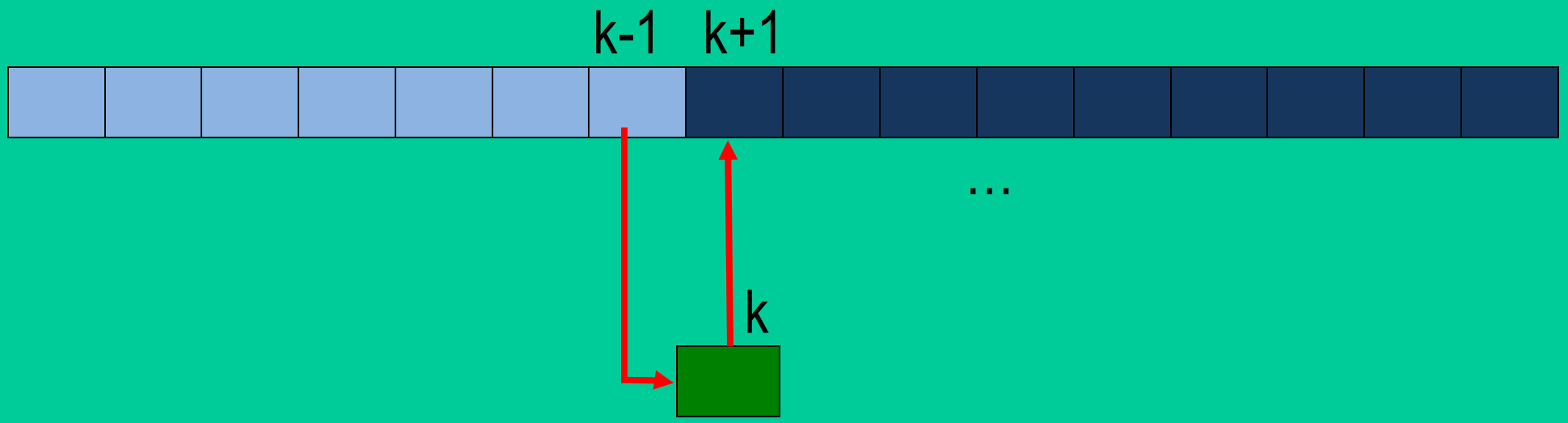
Par la suite, on s'intéressera exclusivement aux structures linéaires **dynamiques**.

De telles structures sont **évolutives**.

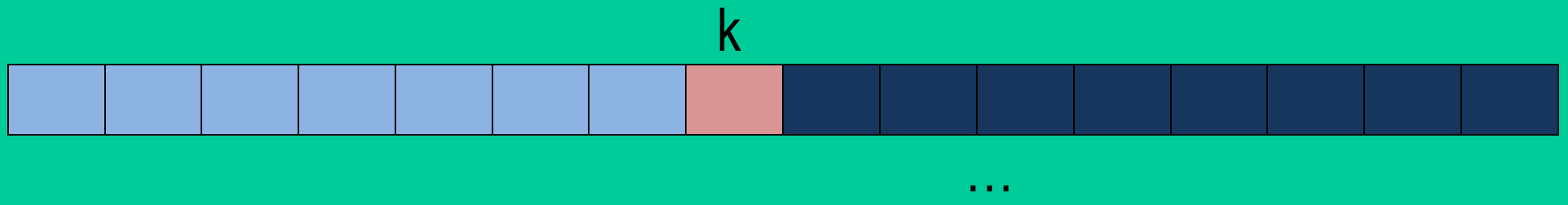
On y peut facilement **ajouter** ou **supprimer** des objets **sans toucher au reste** de la structure.

Avant Insertion en k :

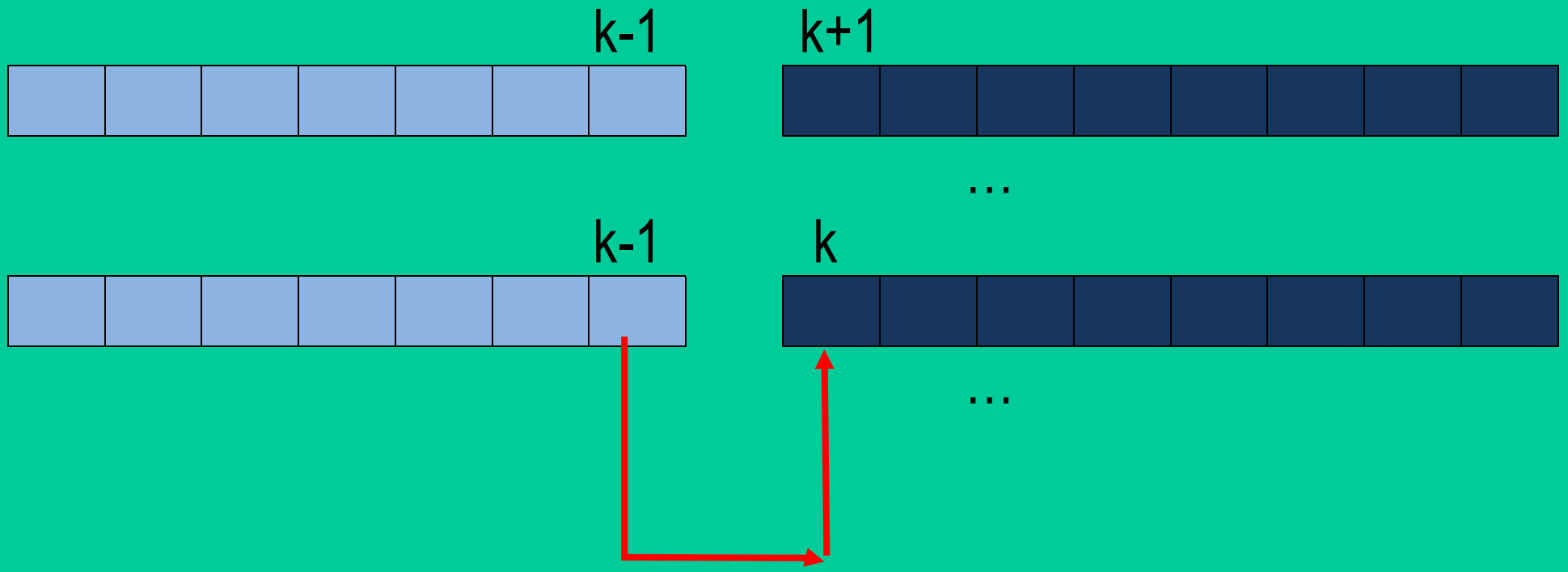
Après insertion en k :



Avant suppression en k:



Après suppression en k:



Dans le développement de logiciels, les principales structures linéaires utilisées sont :

-les **vecteurs**,

-les **listes**,

-les **piles** et les **files**.

Elles sont **mieux adaptées** que les tableaux pour **modéliser** des **systèmes réels**.

Ainsi, un **groupe de patients** en salle d'attente peut être modélisé à l'aide d'une **structure linéaire**.

Il en va de même, en atelier, pour **une gamme de tâches d'assemblage** exécutées séquentiellement par un seul robot.

Ou pour une **file de véhicules** traversant, **sans doubler**, un tunnel.

II- Type abstrait Vecteur

Un **vecteur** est un ensemble **dynamique** d'objets occupant des rangs successifs permettant :

- la **consultation**,
- la **modification**,
- l'**insertion**
- et la **suppression**

d'éléments à des **rangs arbitraires**.

Le type concret **Vector** existe en **java et C++**

```
01 #include <iostream>
02 #include <vector>
03
04 using namespace std;
05
06 int main(int argc, char** argv) {
07
08 /* Initialise vector0ne */
09 vector<int> vector0ne(10,5);
10
11 /* Display size of vector */
12 cout << "Size is " << vector0ne.size() << " elements." << endl;
13
14 /* run utilisant  size()*/
15 for (long index=0; index<(long)vector0ne.size(); ++index) {
16 cout << "Element " << index << ": " << vector0ne.at(index) << endl;
17 }
18
```

```
19 /* Changer size() de vector0ne*/
20 vector0ne.resize(7);
21
22 /* Afficher size de vector0ne */
23 cout << "Size is " << vector0ne.size() << " elements." << endl;
24
25 /* utiliser size() pour determiner limite */
26 for (long index=0; index<(long)vector0ne.size(); ++index) {
27 cout << "Element " << index << ": " << vector0ne.at(index) << endl;
28 }
29
30 return EXIT_SUCCESS;
31 }
```

1-Définition

Un vecteur est une **suite finie** d'objets repérés selon leur **rang**:

$$v = [\, e_1, \ldots, e_n \,]$$

Ordre dans un vecteur

L'**ordre** dans un vecteur est fondamental.

Cet ordre ne porte pas sur les **valeurs** des objets $\mathbf{e_i}$ mais sur les **rangs** occupées par ces objets.

L'objet occupant le **premier rang** d'un vecteur est sélectionné par la fonction **premier**():

premier: Vecteur[Elem] $\rightarrow$? Elem

Par ailleurs, il existe une fonction de **succession** notée **succ**:

succ: Vecteur[Elem] x Elem $\rightarrow$? Elem

Remarque:

Tout objet est accessible en appliquant itérativement la fonction **succ**.

Pour tout objet **e** d'un vecteur v, non vide, on a :

$$\exists\, k > 0 \bullet \; \mathbf{e} = \mathrm{succ}^k(\mathbf{premier}(v))$$

Le nombre d'objets dans un vecteur v est appelé la **taille** de v :

$$\mathbf{taille}: \mathrm{Vecteur}[\mathrm{Elem}] \rightarrow \mathrm{Nat}$$

A la différence d'un tableau, un vecteur est de **taille variable**.

La taille d'un vecteur **varie** lorsqu'on y **insère** ou on y **supprime** des objets

La **taille** est nulle lorsqu'on a un **vecteur vide**.

2- Comment parcourir un vecteur?

La fonction **succ** permet de parcourir **séquentiellement,** tous les objets d'un vecteur.

Le parcours s'effectue dans l'**ordre du rang** des objets.

Si durant le parcours, on applique un certain traitement **traiter()** aux objets, on a la procédure :

```
x ←premier(v);
traiter (x);
Pour i = 1 à taille(v)-1 faire
   debut
        x ← succ(x);
         traiter(x);
        fin
```

3- Autres opérations de base

Les autres opérations de **base** que l'on peut effectuer sur les **vecteurs** sont :

- **créer** un vecteur vide:

vecteurVide : Vecteur[Elem]

- **insérer** un nouvel élément qui sera de rang i:

insérer:Vecteur[Elem] x Nat x Elem → **?** Vecteur[Elem]

- **modifier** un élément de rang i:

modifier:Vecteur[Elem] xNat x Elem→ **?** Vecteur[Elem]

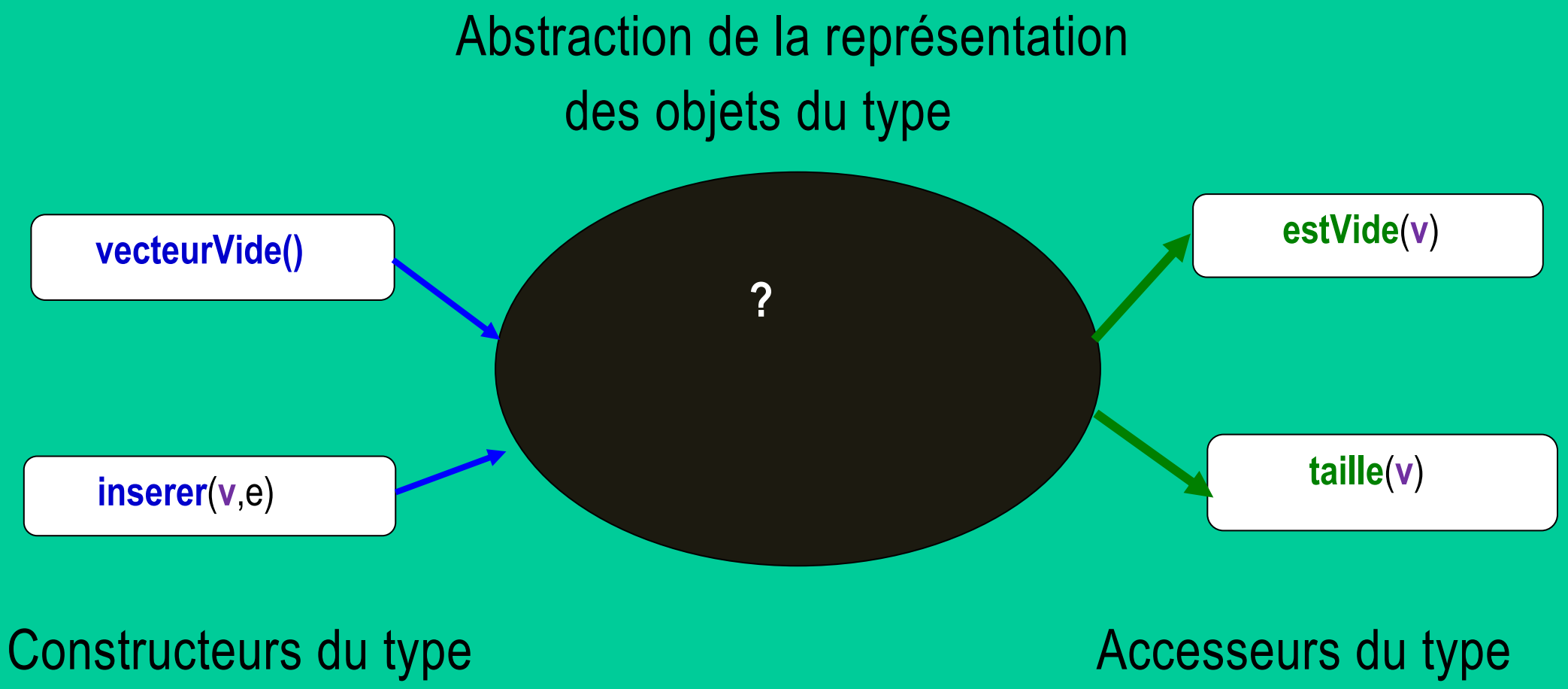
-**supprimer** l'élément de rang i:

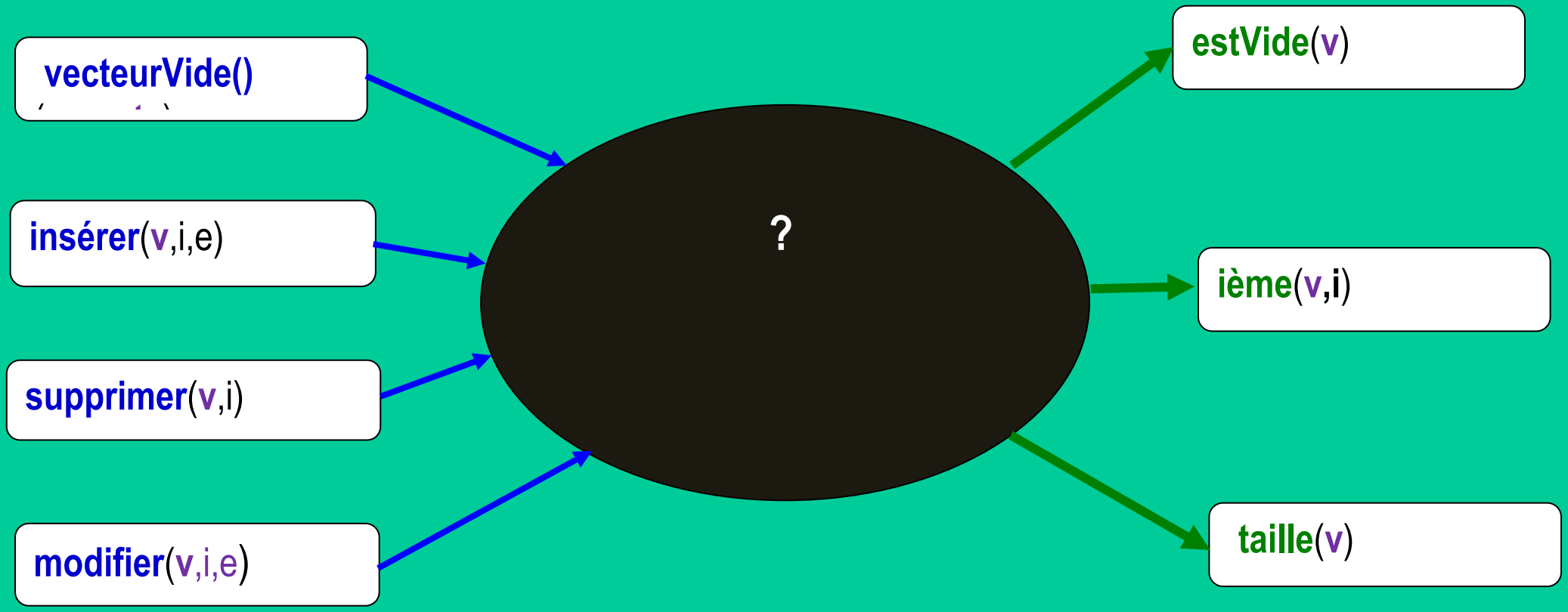
supprimer: Veteur[Elem] x Nat $\rightarrow$? Vecteur[Elem]

- **accéder** à l' élément de rang i :

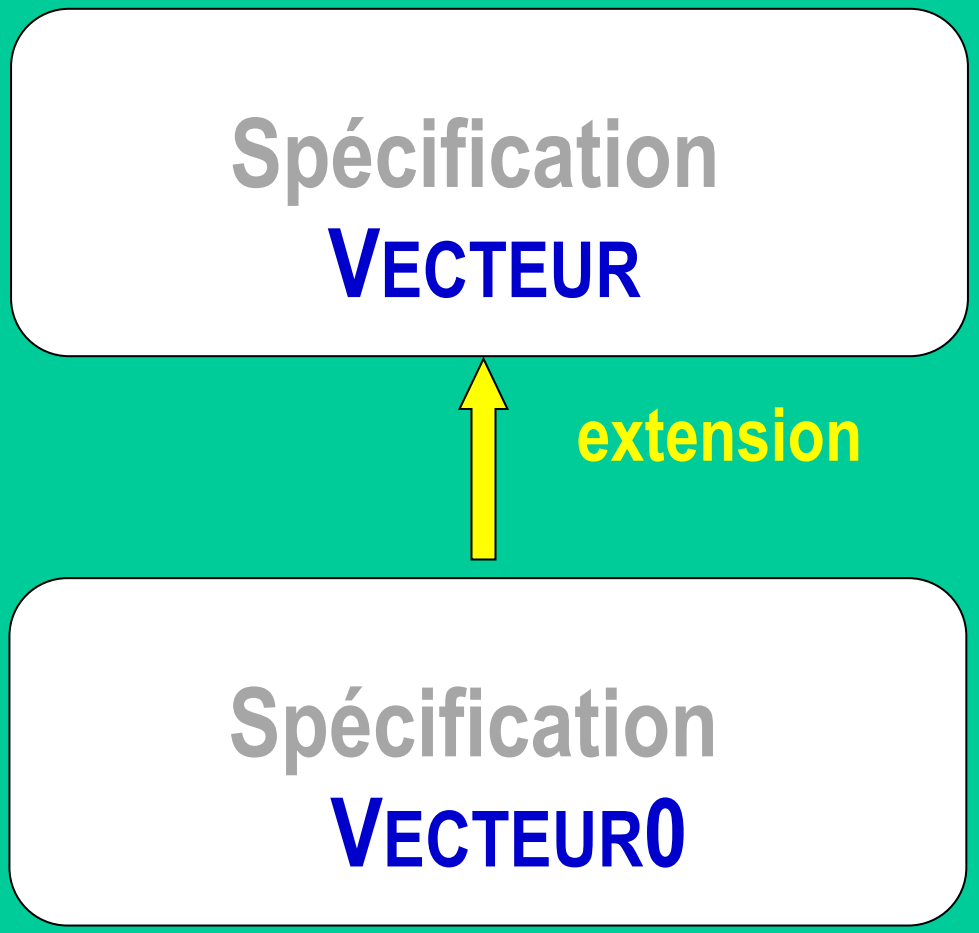
ieme : Vecteur[Elem] x Nat $\rightarrow$? Elem

4- Spécification du type abstrait Vecteur




vecteurVide()
insérer(v,i,e)
supprimer(v,i)
modifier(v,i,e)
?
estVide(v)
ième(v,i)
taille(v)

Stratégie de construction d'une spécification



Une spécification **minimale** en **CasI** .

```
library  vecteur
from Basic/Numbers                  get    Nat,  Int
from Basic/SimpleDatatypes          get    Boolean

spec VECTEUR0 [sort Elem] =
     Int and  Nat and Boolean
then
sort    Vecteur[Elem]
ops
   vecteurVide :    Vecteur [Elem]   ;
   inserer :        Vecteur[Elem]  * Int  * Elem  ->  ? Vecteur[Elem] ;
   estVide :        Vecteur [Elem]  -> Boolean ;
   taille:          Vecteur [Elem] -> Nat
```

```
forall e: Elem; i,k: Int; v:Vecteur[Elem]
    .def inserer(v,i,e) <=>  i>=1  /\  i <= taille(v)+1

    . estVide(vecteurVide) = True
    . estVide(inserer(v,i,e) ) = False

    . taille(vecteurVide )= 0
    . taille(inserer(v,k,e))= taille(v) +1

end
```

La spécification **VECTEUR0** peut être étendue comme suit :

```
spec VECTEUR [sort Elem] =
        VECTEUR0 [sort Elem]
then
    ops
    modifier:       Vecteur [Elem]   * Int  *  Elem-> ? Vecteur [Elem] ;
    supprimer:      Vecteur [Elem]   * Int -> ? Vecteur [Elem] ;
    ieme:           Vecteur [Elem]   * Int -> ? Elem
```

forall e: Elem; i,k: Int; v: Vecteur[Elem]

. **def** modifier(v,i,e) <=> 1<=i /\ i <= taille(v)

. **def** supprimer(v,i) <=> 1<=i /\ i <= taille(v)

. **def** ieme (v,i) <=> 1<=i /\ i <= taille(v)

. ieme(inserer(v,i,e) , i) = e

. 1<= i /\ i<k => ieme(inserer(v,k,e), i) = ieme(v,i)

. k<i /\ i <=taille(v)+1 => ieme(inserer(v,k,e), i) = ieme(v,i-1)

. ieme(modifier(v, i, e), k) = e **when** k = i **else** ieme (v,k)

. i= k => supprimer(inserer(v ,k, e),i) = v

. i> k => supprimer (inserer(v ,k, e),i) = inserer(supprimer(v,i-1),k,e)

. i< k => supprimer (inserer(v ,k, e),i) = inserer(supprimer(v,i),k-1,e)

end

III- Type abstrait Liste

1-Définition

Une liste linéaire λ est un ensemble d'objets :

- **dynamique**,
- **ordonné**,

dont les objets sont accessibles **relativement les uns aux autres**, sur la base de leur position.

On note :

$$\lambda = [\, e_1,\ldots,e_n \,]$$

Soit Ω l'ensemble des objets e_i; une liste peut être définie par l'application:

$$N^* \rightarrow \Omega$$

2-Opérations sur une liste

L'objet occupant le **premier rang** d'une liste est sélectionné par la fonction **tête**(v):

tête: Liste[Elem] $\rightarrow$? Elem

Le nombre d'objets d'une liste λ est appelé la **taille** de λ :

taille: Liste[Elem] $\rightarrow$ Nat

La liste qui ne contient aucun élément est la **liste vide**:

listeVide: Liste[Elem]

L'opération **cons** **construit** une liste λ en insérant un objet en **tête** d'une autre liste λ':

cons: Elem x Liste[Elem] $\rightarrow$ Liste[Elem]

L’opération **fin** retourne la liste amputée de son premier objet:

fin: Liste[Elem] $\rightarrow$? Liste[Elem]

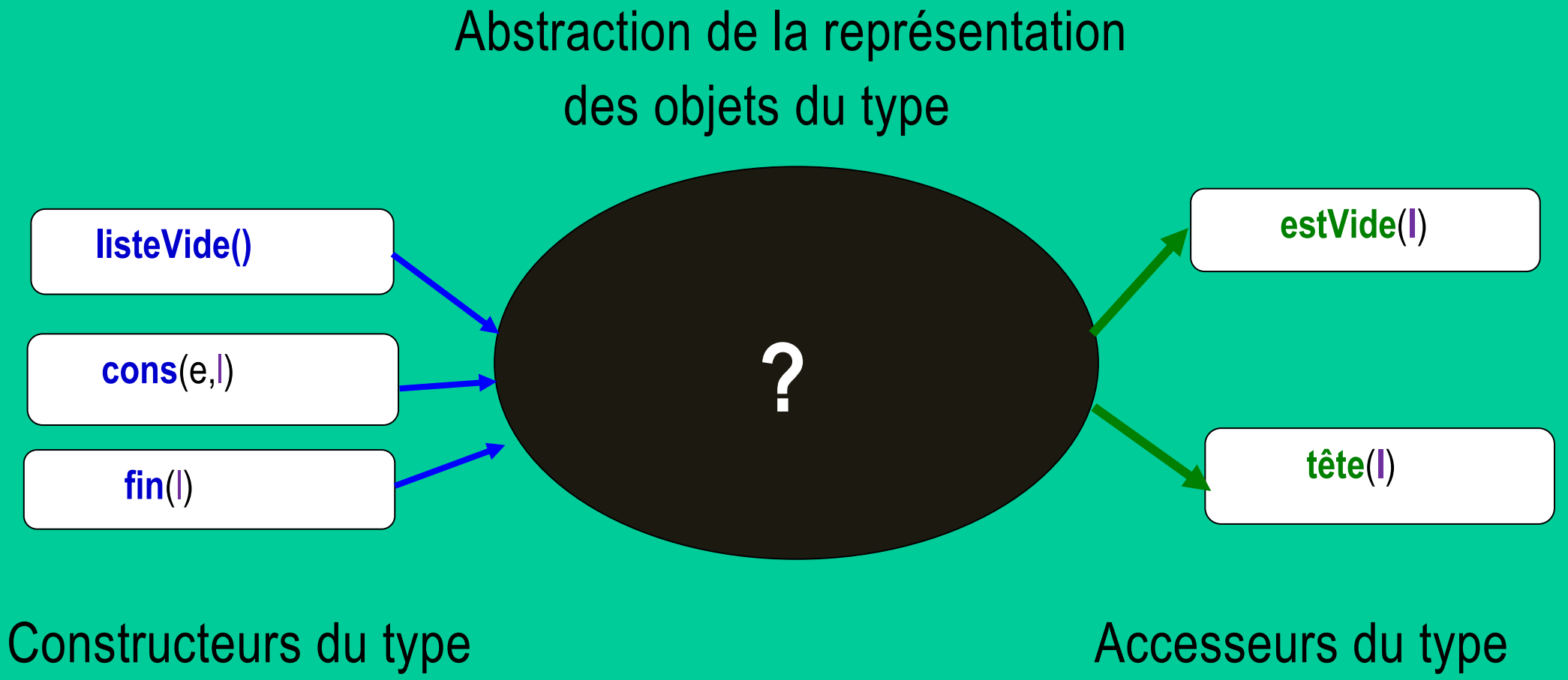
$$\textbf{def}\ \text{fin}(\lambda) <=> \text{estVide}(\lambda) = \text{False}$$

- L’opération **tête** retourne le **premier objet** d'une liste :

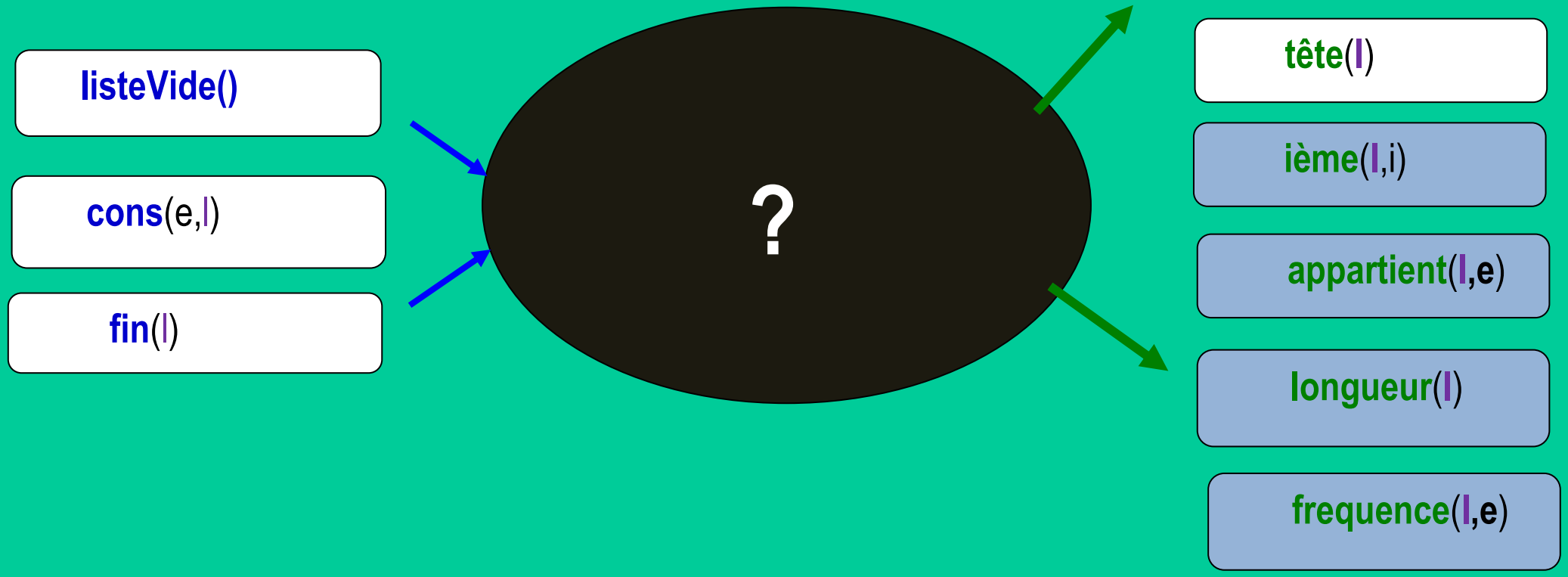
tête: Liste[Elem] $\rightarrow$? Elem

$$\textbf{def}\ \text{tete}(\lambda) <=> \text{estVide}(\lambda) = \text{False}$$

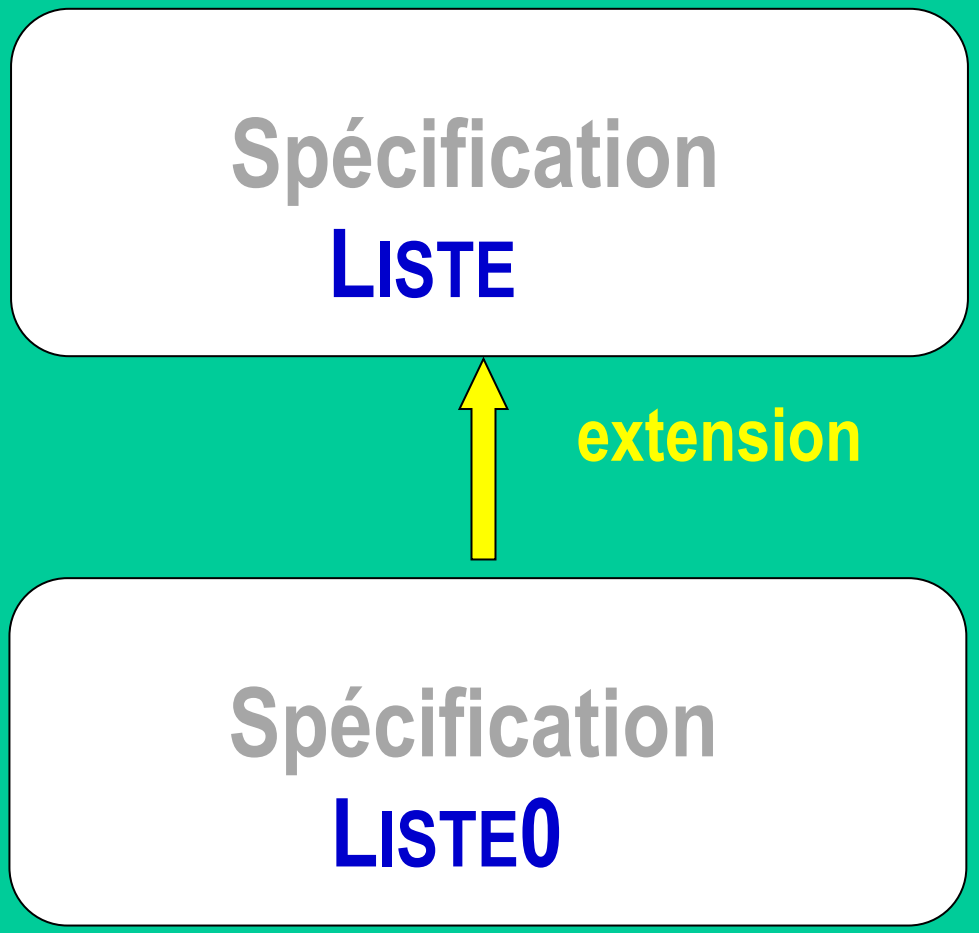
3- Spécification du type abstrait Liste



Abstraction de la représentation des objets du type



Stratégie de construction d'une spécification



On a d'abord une spécification **minimale**:

```
library Liste
from Basic/Numbers get Nat, Int
from Basic/SimpleDatatypes get Boolean


spec LISTE0[sort Elem] =
        Nat  and Int and Boolean
then sort Liste[Elem]
 ops              listeVide : Liste[Elem];
                   cons : Elem * Liste[Elem] -> Liste[Elem];
                   fin : Liste[Elem] -> Liste[Elem];
                   estVide : Liste[Elem] -> Boolean;
                   tete : Liste[Elem] ->? Elem
```

```
        forall l : Liste[Elem]; e : Elem
        . def tete(l) <=> estVide(l) = False
        . def fin(l) <=> estVide(l) = False
        . estVide(listeVide) = True
        . estVide(cons(e, l)) = False
        . tete(cons(e, l)) = e
        . fin(cons(e, l)) = l
end
```

Cette spécification peut être étendue comme suit:

```
spec   LISTE [sort Elem] =
          LISTE0 [sort Elem]
then
   ops
     appartient    : Elem * Liste[Elem] ->  Boolean;
      longueur     : Liste[Elem] ->  Nat;
      ieme         : Liste[Elem] * Nat ->  Elem;
      frequence    : Liste[Elem] * Elem ->  Nat
```

```
forall l : Liste[Elem] ; i:Nat;  x,y,e :Elem
   . def ieme(l, i) <=> i>0 /\ l <= longueur(l)
   . appartient(x, listeVide) = False
   . appartient(x, cons(y, l))=True  <=> x = y \/ appartient(x,l)=True
   . longueur(listeVide) = 0
   . longueur(cons(e,l)) = longueur(l)+1
   . ieme(cons(x,l),1) = x
   . ieme(cons(x,l), i+1) = ieme(l, i)
   . frequence(listeVide, x ) = 0
   .frequence(cons(x,l), y) = frequence(l,y)+1 when x = y else frequence(l,y)
end
```

4- Comment parcourir une liste?

Les fonctions **tête** **et** **fin** permettent de parcourir de façon **séquentielle** une liste.

Le parcours s'effectue dans l'**ordre du rang des objets**.

Durant le parcours, on peut appliquer un traitement aux objets.

```
x ←tête(v);
y ← fin(v)
traiter (x);
Tant que y ≠ listeVide faire
        debut
        x ← tête(y);
        y ← fin(y)
        traiter(x);
        fin
```

IV- Types abstraits Pile et File

Pour beaucoup de traitements, les seules opérations à effectuer sur les **listes** sont:

- des **insertions aux extrémités**,
- des **suppressions aux extrémités**.

D'où l'importance particulière accordée aux notions de **pile** et de **file**.

1- Type abstrait Pile

Dans les piles les **insertions** et les **suppressions** se font à une **seule** extrémité appelée **sommet** de la pile.

Les opérations de **base** sur les piles sont:

- créer une **pile vide** :

  pileVide : Pile[Elem]

- **empiler** un objet:

empiler : Pile[Elem] x Elem $\rightarrow$ Pile[Elem]

- **retirer** l'objet qui se trouve au sommet:

dépiler : Pile[Elem] $\rightarrow$? Pile[Elem]

def depiler(p) <=> estVide(p) = False

-tester si une pile **est vide** :

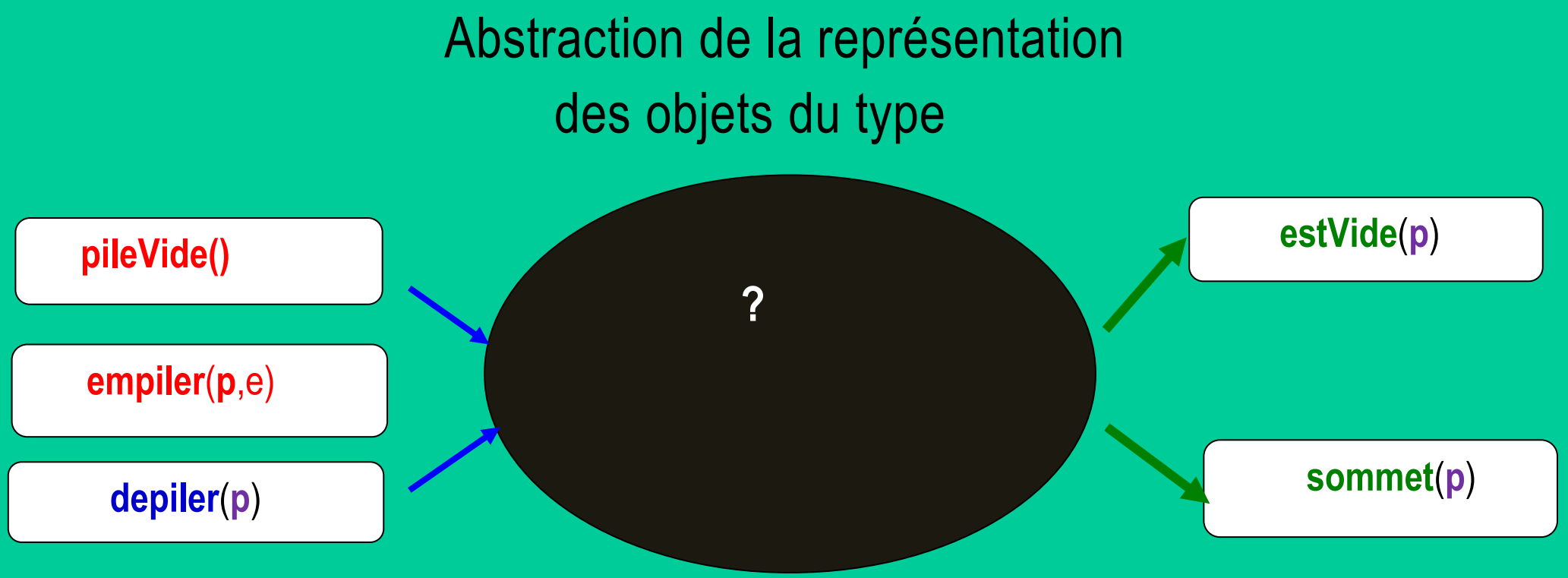
estVide : Pile Pile[Elem] $\rightarrow$Booléen

- accéder au **sommet** d'une pile :

sommet: Pile Pile[Elem] $\rightarrow$? Elem

def sommet(p) <=> estVide(p) = False

Une première spécification du type PILE peut être établie comme suit :



```
library libraryPile
from Basic/SimpleDatatypes get Boolean

spec PILE [sort Elem]  =
                Boolean
then
generated type Pile[Elem]  ::=  pileVide  |
                                empiler(Pile[Elem] ; Elem)
ops
        depiler:     Pile[Elem]  ->? Pile[Elem]  ;
        sommet:     Pile[Elem]  ->? Elem;
        estVide :     Pile[Elem]  -> Boolean
```

forall e: Elem; p: Pile[Elem]

. **def** sommet(p) <=> estVide(p) = False
. **def** depiler(p) <=> estVide(p) = False
. estVide(pileVide) =True
. estVide(empiler(p,e)) = False
. sommet(empiler(p,e)) = e
. depiler(empiler(p,e)) = p

end

2-Type abstrait File

Une **file** est une **liste** où on fait:

- les **adjonctions** à une extrémité,
- les accès et les **suppressions** à l'autre extrémité.

Par analogie avec les **files d'attente** on dit que l'objet présent depuis le **plus longtemps** est le **premier**.

Les opérations de **base** sur les files sont:

1-Trois constructeurs pour :

-créer une file vide

fileVide : File[Elem]

- **ajouter** un élément dans la file :

enfiler: File[Elem] x Elem $\rightarrow$ File

- **retirer** le premier élément de la file

defiler: File[Elem] $\rightarrow$? File

def defiler(f) <=> estVide(f) = False

1-Deux accesseurs pour :

-accéder au premier élément de la file :

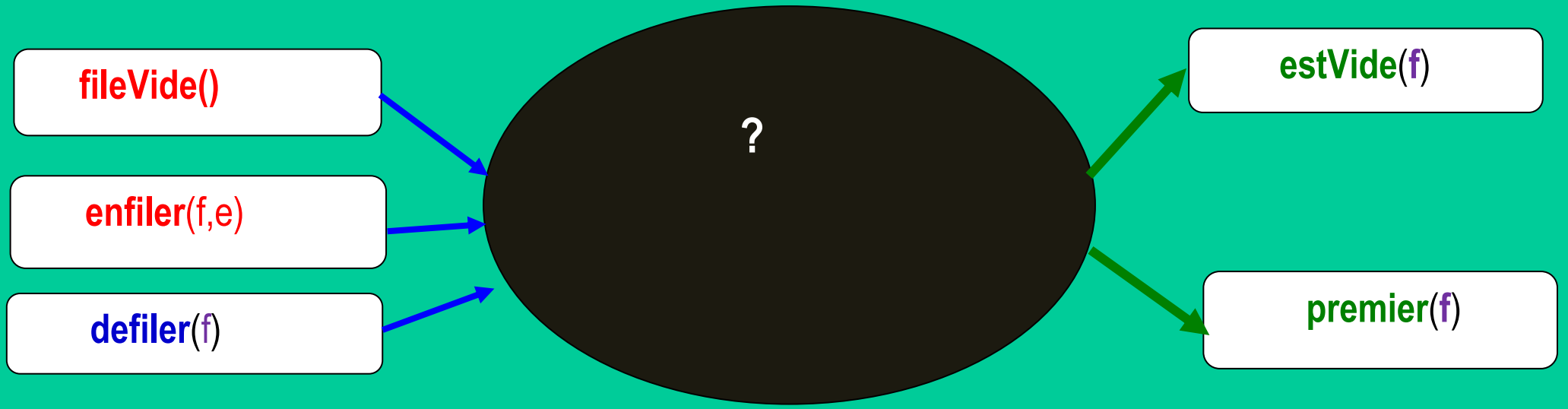
premier : File [Elem] $\rightarrow$? Elem

def premier(f) <=> estVide(f) = False

-tester si une file est vide

estVide : File[Elem] $\rightarrow$ Boolean

Abstraction de la représentation
des objets du type



Constructeurs du type

Accesseurs du type

Une spécification du type abstrait **FILE** peut être établie comme suit :

```
library libraryFile
from Basic/SimpleDatatypes get Boolean

spec FILE [sort Elem] =
     Boolean
then generated type File [Elem]  ::=  fileVide  |
                                      enfiler(File[Elem]  ; Elem)
then
ops
        defiler:      File[Elem]  ->? File[Elem]  ;
        premier:      File[Elem]  ->? Elem;
        estVide :     File[Elem]  ->Boolean
```

```
forall e: Elem; f: File[Elem]
    . def premier(f)  <=>  estVide(f)    = False
    . def defiler(f)    <=>  estVide(f)    = False

    . estVide(f)  = True      <=>     f = fileVide
    . premier(enfiler (f,e)) = e when estVide(f)=True  else premier(f)
    . defiler(enfiler(f,e)) = fileVide  when estVide(f) = True
                                        else enfiler (defiler(f),e)
end
```