Types abstraits

Types de base

entiers, réels, booléens, caractères, pointeurs (adresses),…

Constructeurs

puissance	E^n	type_E ... […]
produit cartésien	E x F x …	struct { … }
itération (étoile)	E^*	

Types abstraits - objets

indépendants du langage et de l'implémentation
décrits par :
- interface (ensemble de base, opérations, relations, axiomes)
- implémentation (codage en C ou autre)

Exemples :	Vecteur	Ensemble	Liste
	Arbre	Graphe	…

Entiers

- Ensemble : intervalle de $\mathbb{N}$
- Opérations (exemples)

  \+ - * / mod

  = ^{1} < £ > ^{3}

  succ pred maxint

Signature

$\_ + \_ : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$

succ : $\mathbb{N} \rightarrow \mathbb{N}$

maxint : $\rightarrow \mathbb{N}$

$\_ \leq \_ : \mathbb{N} \times \mathbb{N} \rightarrow$ Booléens, *etc*.

- **Axiomes**

  $(a + b) + c = a + (b + c)$,

  $a \leq b$ et $b \leq c \Rightarrow a \leq c$, *etc*.

- **Implémentation** (courante)

  $r = 547_{10}$ Bin (r) 0 0 0 0 0 0 1 0 0 0 1 0 0 0 1 1

  sur 16 bits

  $r = -547_{10}$ Bin ($2^{16} + r$) 1 1 1 1 1 1 0 1 1 1 0 1 1 1 0 1

Réels

Ensemble sous ensemble (fini) de $\mathbb{R}$

Opérations

\+ - * / = <

cos log int sqrt *etc.*

Signature

$\_ + \_ : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$

$\cos : \mathbb{R} \to \mathbb{R}$

$< : \mathbb{R} \times \mathbb{R} \to$ Booléens

$\text{puiss} : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$

Axiomes

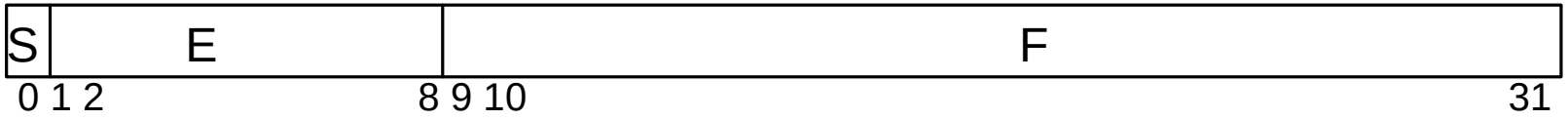
corps ordonné

$x \times (y + z) = x \times y + x \times z$ *etc.*

Implémentation (standard IEEE)

$$x = (-1)^{s} . 2^{E - 127} . 1,F$$

signe — caractéristique — mantisse



$0,8 = (-1)^{0} . 2^{126-127} . 1,10011001100 \ldots_{2}$

0	0 1 1 1 1 1 1 0	1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0

$0,75 = (-1)^{0} . 2^{126-127} . 1,100 \ldots_{2}$

0	0 1 1 1 1 1 1 0	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

$0,8 - 0,75$

0	0 1 1 1 1 0 1 0	1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0	0 0 0 0

$0,05 = (-1)^{0} . 2^{122-127} . 1,10011001100 \ldots_{2}$

0	0 1 1 1 1 0 1 0	1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0	1 1 0 0

Vecteurs

Ensemble

ensemble des suites finies indicées d'éléments d'un même type

Opérations

Vect : Entier x Entier ➜ Vecteur

ième : Vecteur x Entier ➜ Elément

chg-ième : Vecteur x Entier x Elément ➜ Vecteur

etc.

Axiomes (exemples) [*v* vecteur, *i, j* indices, *e* élément]

ième (chg-ième (*v, i, e*), *i*) = *e*

$i \neq k \Rightarrow$ ième (chg-ième (*v, i, e*), *k*) = ième (*v, k*)

quand indice_min $\leq i, j \leq$ indice_max

Implémentations possibles

type v [ ]

table de hachage

Listes

Concept

- mémorisation d'une suite d'éléments

  $(e_1, e_2, e_3, \ldots, e_n)$

- accès séquentiel

  $e_1 \rightarrow e_2 \rightarrow e_3 \rightarrow \ldots \rightarrow e_n$

- parcours : - tête
  - successeur
  - fin
- position / adresse

UMLV ©

```
fonction UNIQUE ( L  liste ) : liste ;
/* supprime de L les copies d'éléments */
 p,q sont des adresses
début
    p ¬ Tête ( L ) ;
    tant que p défini faire
        q ¬ p ;
        tant que Succ ( q, L ) défini faire
                si  Val ( p, L ) = Val ( Succ (q, L )) alors Enlever ( Succ (q, L ))
                sinon  q ¬ Succ ( q, L ) ;
        p ¬ Succ ( p, L ) ;
    retour  L ;
fin.
```

NOTE : Temps de calcul = O ($|L|^2$)

Exercice : Écrire un algorithme fonctionnant en O ($|L| \log |L|$)

Listes

Ensemble

Ensemble des suites finies d'éléments d'un même type.

E = ensemble des éléments

$E^* = \{(e_1, e_2, \ldots, e_n) / n \geq 0,\ e_i \in E \text{ pour } i = 1,\ldots,n\}$

longueur $((e_1, e_2, \ldots, e_n)) = |(e_1, e_2, \ldots, e_n)| = n$

liste vide = ( ) = e

position de e_i dans $(e_1, e_2, \ldots, e_n) = i$

accès séquentiel :

p = place ou adresse de e_i

Succ (p) = adresse de e_{i+1} si elle existe

Tête (L) = adresse de e_1

Axiomes

p adresse d'un élément de $L \Rightarrow \exists k \geq 0\ \ p = \text{Succ}^k(\text{Tête}(L))$

$k \geq$ longueur (L) $\Rightarrow \text{Succ}^k$ (Tête (L)) non défini

Opérations de base

Liste_vide : ® Liste

Tête : Liste ® Adresse

Fin : Liste ® Liste

Cons : Elément x Liste ® Liste

Premier : Liste ® Elément

Elt : Adresse ® Elément

Succ : Adresse ® Adresse

Axiomes (exemples)

Tête (*L*), Fin (*L*), Premier (*L*) définis ssi $L \neq e$

$L \neq e$ Þ Premier (*L*) = Elt (Tête (*L*))

Fin (Cons (*e, L*)) = *L* Premier (Cons (*e, L*)) = *e*

$L \neq e$ Þ Succ (Tête(*L*)) = Tête (Fin(*L*))

Implémentations possibles

par tables, pointeurs, curseurs, ...

Extensions

Concaténation

Produit : Liste x Liste ® Liste

$(e_1, \ldots, e_n) . (f_1, \ldots, f_m) = (e_1, \ldots, e_n, f_1, \ldots, f_m)$

Modifications

Ajouter : Elément x Adresse x Liste ® Liste

ajouter *e* après l'élément d'adresse *p*

Supprimer : Adresse x Liste ® Liste

Elément : Elément x Liste ® Booléen

Autres

Place : Elément x Liste ® Adresse

Position : Elément x Liste ® Entier

Ième : Entier x Liste ® Elément

Tri : Liste ® Liste

etc.

Choix d'une implémentation

Problème faisant intervenir une ou des listes

Modélisation

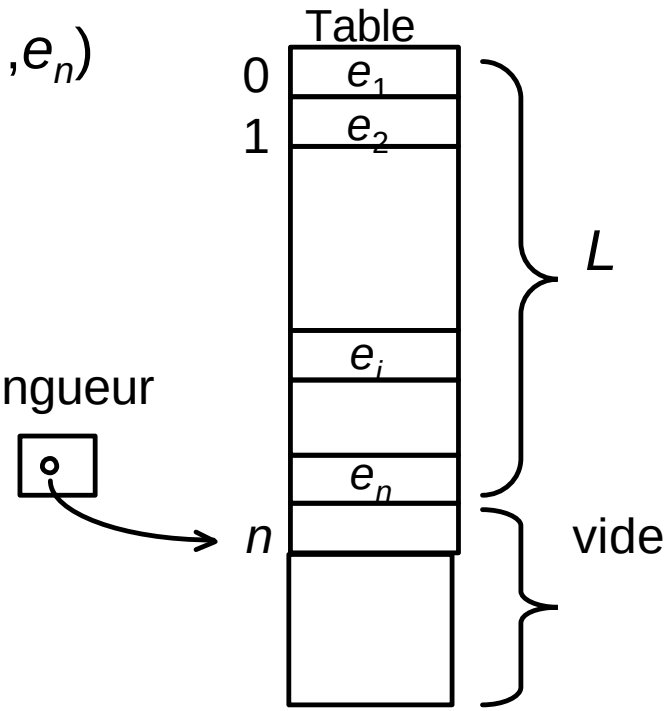
Opérations élémentaires sur les listes

Critères de la réalisation
(simplicité, rapidité, langage, ...)

Implémentation des listes

Implémentation par table

$L = (e_1, e_2, \ldots, e_n)$



Positions explicites

Adresse (e_i) = i - 1 = Position (e_i) - 1

Succ (p) = p + 1 (si $1 \leq p < n$ -1)

Avantages

simple, économique en espace

Inconvénients

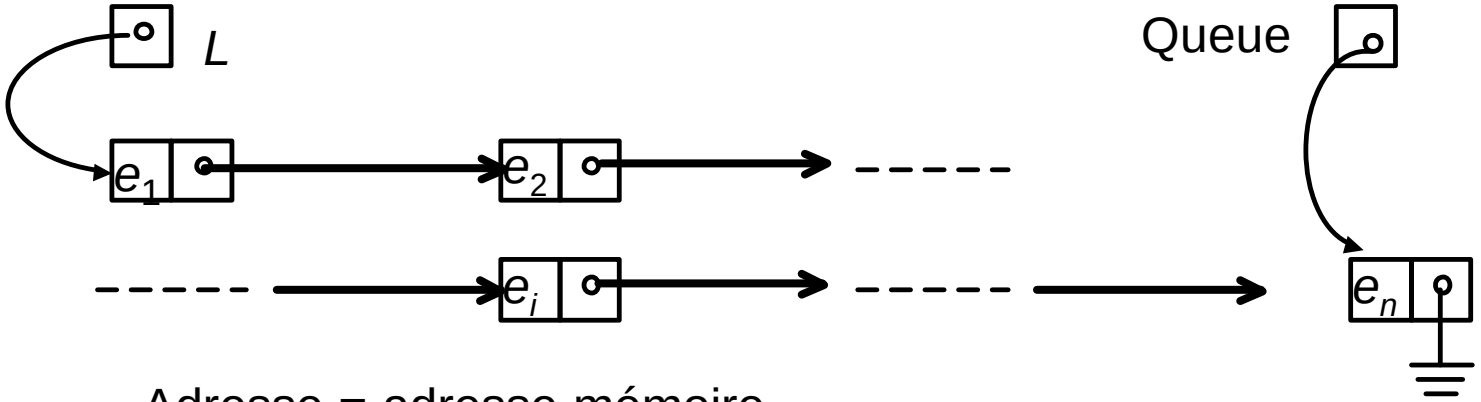
concaténation lente, opérations de modifications lentes

```
struct {  int  long ;
          Element  table [MAX] ;
}  Liste;

Fonction  ajouter (L  Liste, p  Adresse, e  Element) : Liste ;
début
    si  L. long = MAX  alors
        erreur ("opération impossible")
    sinon  si  p < -1  ou  p ³ L. long  alors
        erreur ("opération impossible")
    sinon
        pour  k ¬ L. long - 1  à  p + 1  pas  - 1  faire
                L . table [k + 1] ¬ L. table [k] ;
        L. table [p + 1] ¬ e ;
        L. long ¬ L. long + 1 ;
    retour  (L) ;
fin  .
```

Pointeurs

$L = (e_1, e_2, \ldots, e_n)$



Adresse = adresse mémoire
Successeur explicite
L peut être identifié à Tête (*L*)

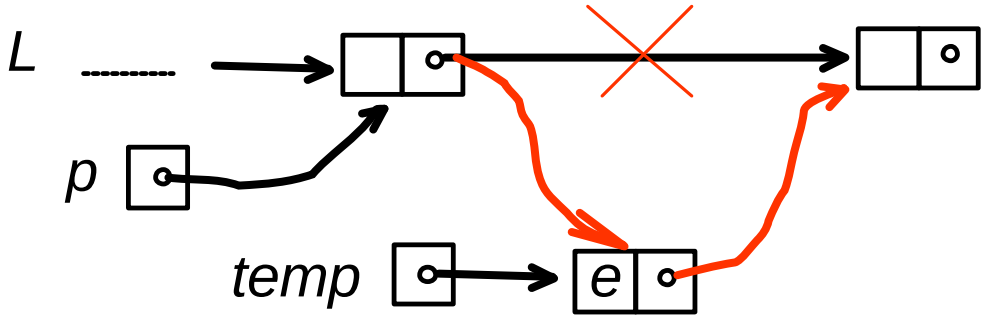
Avantages

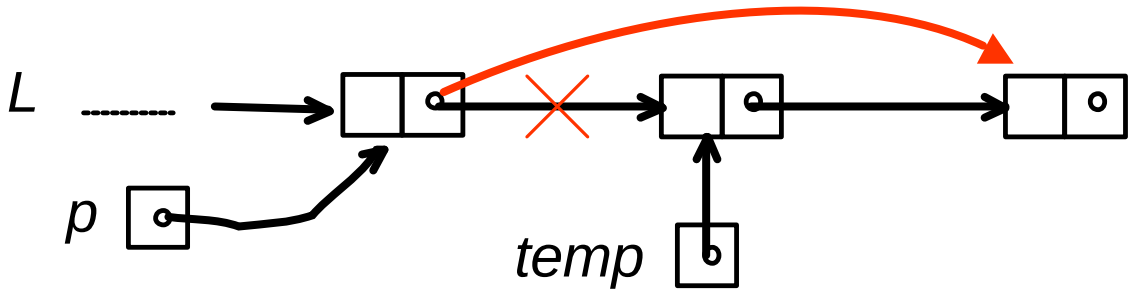
nombreuses opérations rapides
gestion souple, autorise plusieurs listes
accès à tout l'espace mémoire disponible

Inconvénients

pas d 'accès direct
un peu gourmand en mémoire

Ajout/Suppression



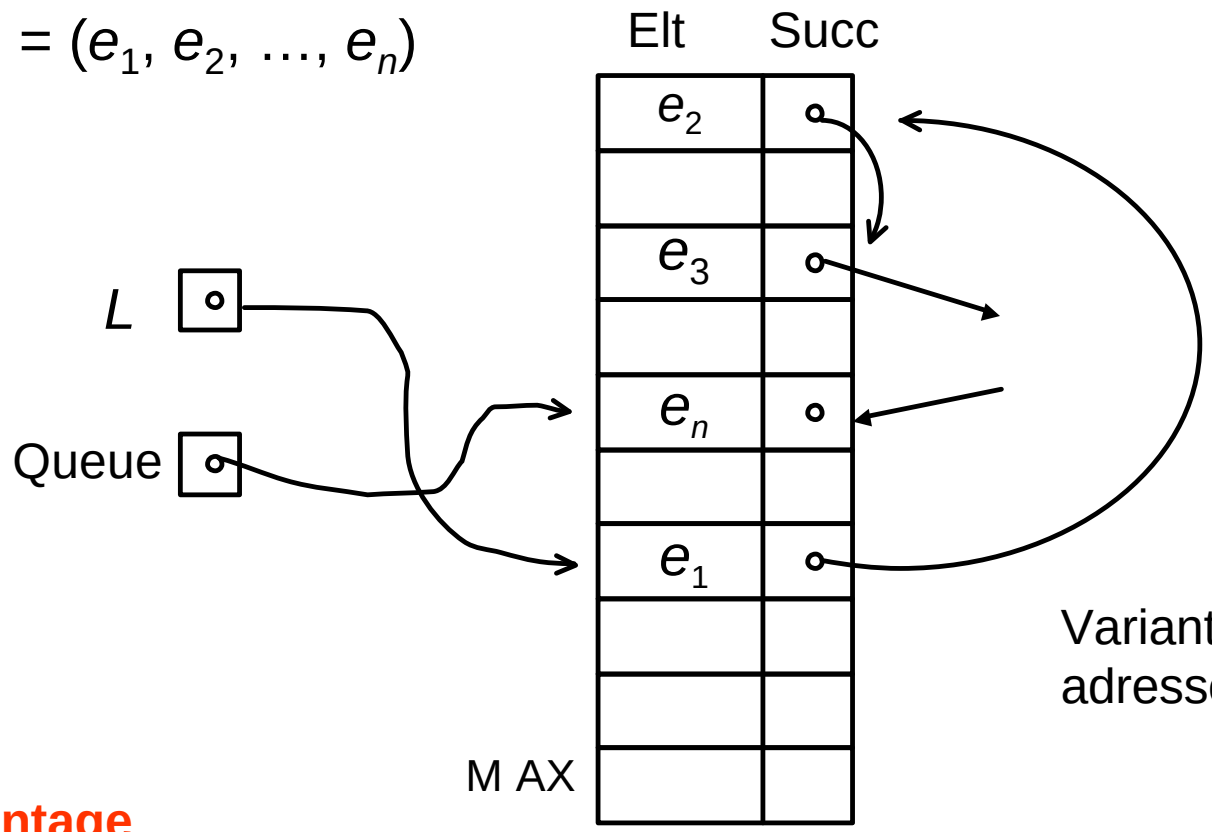


```
typedef struct CELLULE {
     element elt ;
     struct CELLULE  *  succ ;
} cellule ;
typedef  cellule  *  adresse ;
typedef  adresse  liste ;

...
liste ajouter ( liste L, adresse p, element e )
{
     temp = (adresse) malloc(sizeof(cellule)) ;
     if ( temp == NULL ) erreur () ;
     else {
          temp -> elt = e ;
          temp -> succ  =  p -> succ ;
          p -> succ  =  temp ;
     }
return L ;
}
```

Curseurs

$L = (e_1, e_2, \ldots, e_n)$



Variante des pointeurs
adresses = indices

Avantage

Gestion directe de l ’espace

Inconvénient

Choix de MAX ?

UMLV ©

```
type  Liste = Adresse = -1 .. MAX ;
      Cellule =  structure {
            elt : Element ;  succ : Adresse ;
      } ;
Var   Espace :  table [0 .. MAX]  de  cellule ;

fonction  AJOUTER  (L Liste, p  Place, e  Element) : Liste ;
début
      temp  ¬  indice d 'une case libre de Espace ;
      Espace [ temp ] . elt   ¬   e ;
      Espace [ temp ] . succ   ¬   Espace [ p ] . succ ;
      Espace [ p ] . succ   ¬   temp ;
      retour  ( L ) ;
fin.
```

Temps constant

si la recherche d 'une case libre prend un temps constant

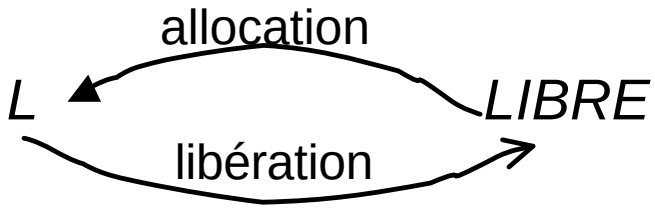
Question

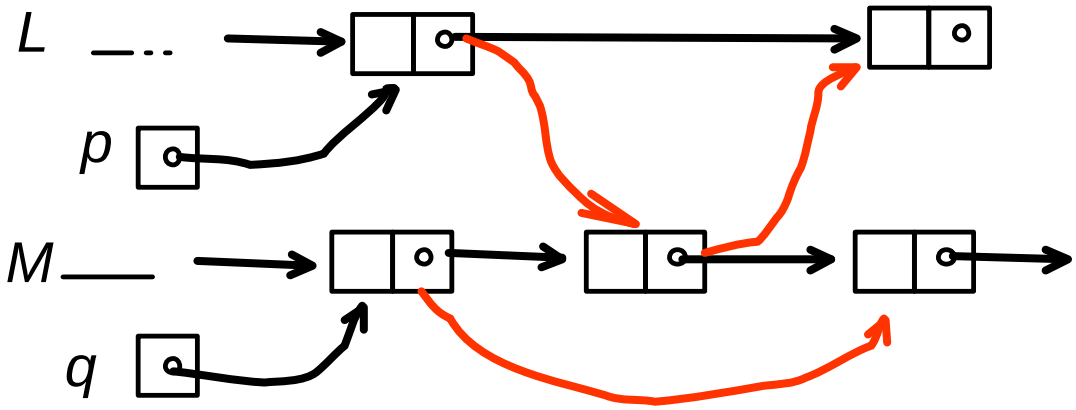
Comment trouver efficacement une case libre ?

Allocation/Libération

Transfert d'une cellule d'une liste à une autre

Exemple :





Transfert

fonction TRANSFERT (*L, M* Listes, *p, q* Adresses) : Adresse ;
début
temp ¬ *q->succ* ;
q->succ ¬ *temp->succ* ;
temp->succ ¬ *p->succ* ;
p->succ ¬ *temp* ;
retour (*temp*) ;
fin .

Temps
constant si implémentation par pointeurs ou curseurs

Ajout/Suppression

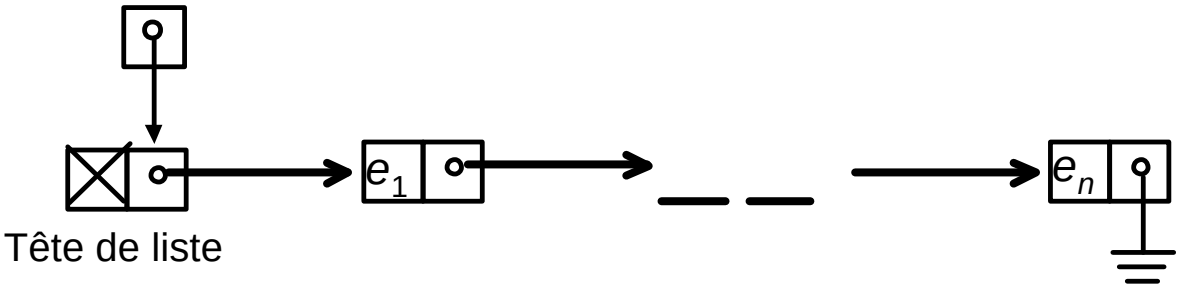
Listes possédant une tête de liste
LIBRE : liste des cellules disponibles

fonction AJOUTER (*L* Liste, *p* Adresse, *e* Element) : Liste ;
début
    si Vide (*LIBRE*) **alors**
        erreur (“mémoire saturée”)
    sinon
        temp ¬ TRANSFERT (*L, LIBRE, p, LIBRE*) ;
        temp->elt ¬ *e* ;
        retour (*L*) ;
fin.

```
fonction  ENLEVER (L  Liste, p  Adresse) : Liste ;
début
        si  p->succ  est défini  alors
                TRANSFERT (LIBRE, L, LIBRE, p) ;
        retour  (L) ;
fin.
```

Tête de liste

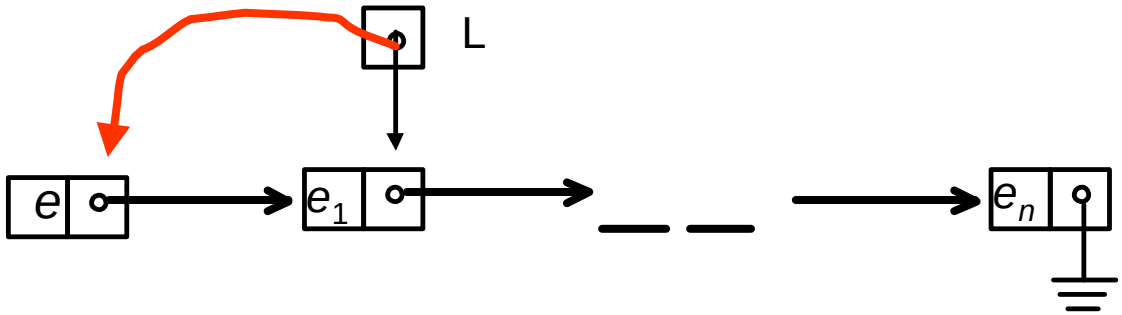
$L = (e_1, e_2, \ldots, e_n)$



Ajout en tête = ajout après Tête(L)

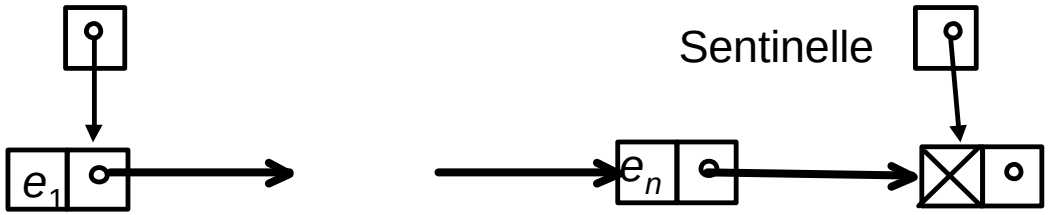
Tout ajout se fait après une cellule

… sinon



fonction AJOUTER_en_TETE (*L* Liste, *e* Element) Liste ;
début
temp ¬ adresse d ’une nouvelle cellule ;
temp->elt ¬ *e* ;
temp->succ ¬ *L* ;
L ¬ *temp* ;
retour (*L*) ;
fin .

Sentinelle

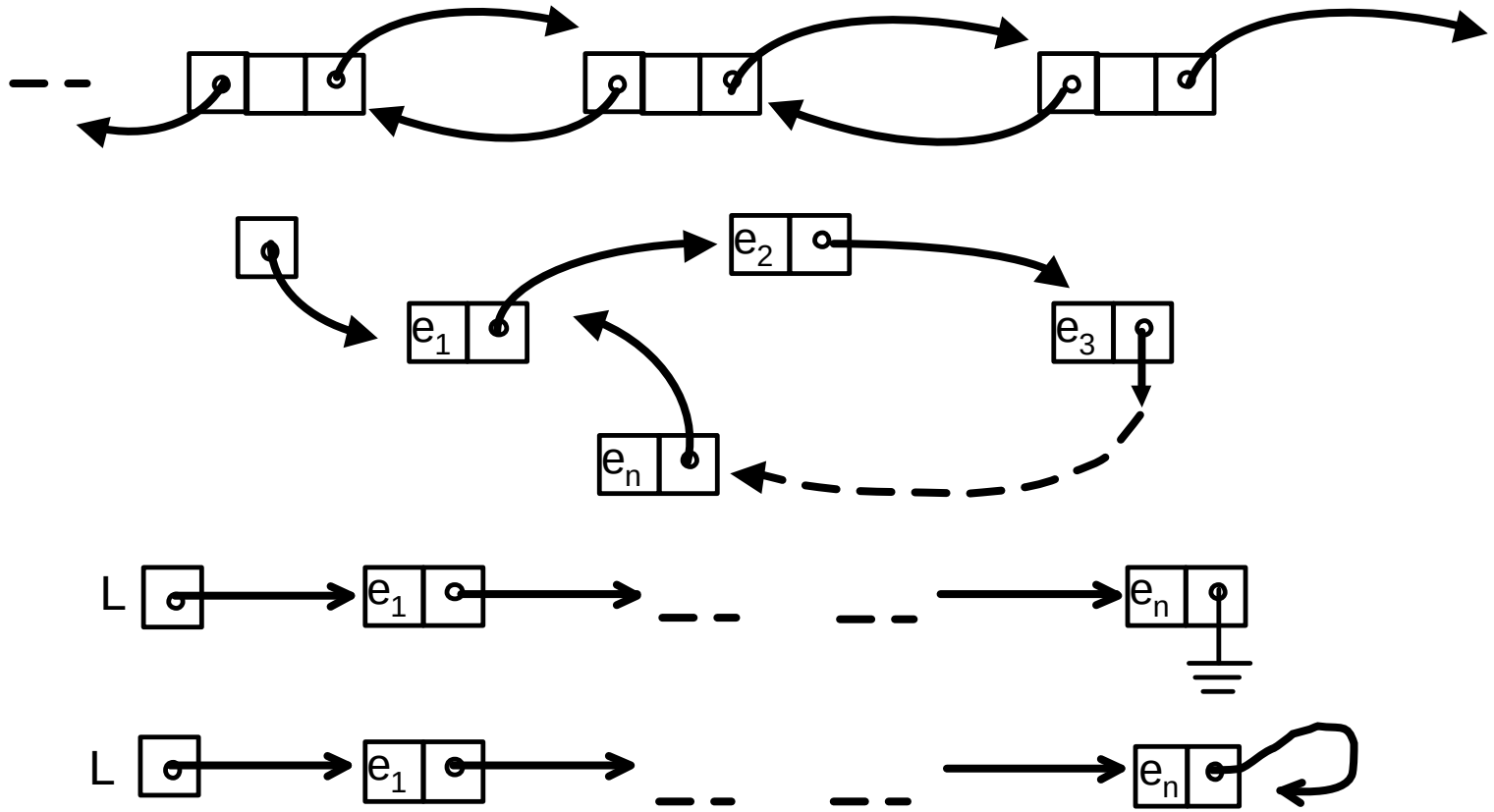


Accélère la recherche séquentielle

fonction Place (*e* Element, *L* Liste) : Adresse ;
début
    Sentinelle->*elt* ¬ *e* ;
    p ¬ Tête (*L*) ;
    tant que *p*->*elt* ≠ *e* **faire** *p* ¬ *p*->*succ* ;
    si *p* = Sentinelle **alors retour** (NULL)
    sinon retour (*p*) ;
fin

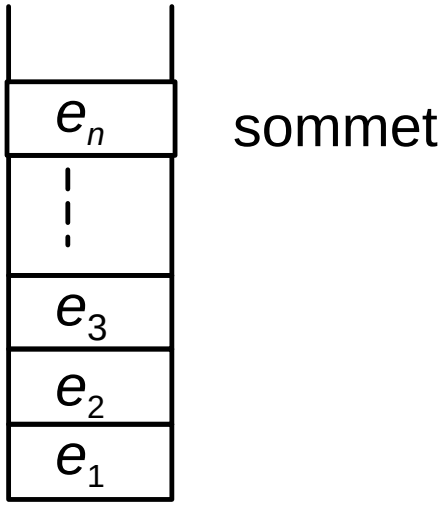
Variations

Double chaînage



Pile

Liste avec accès par une seule extrémité
liste LIFO : « Last In First Out »



Opérations

Pile_vide : ® Pile
Empiler : Pile x Elément ® Pile
Dépiler : Pile ® Pile
Sommet : Pile ® Elément
Vide : Pile ® Booléen

Axiomes

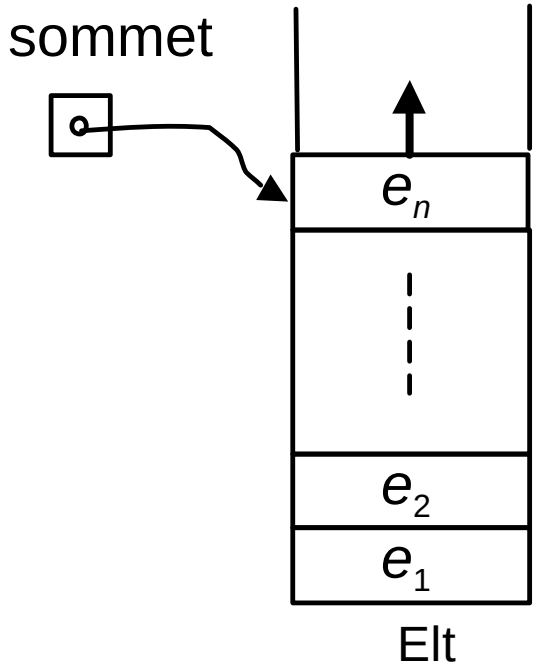
Dépiler (*P*) et Sommet (*P*) défini ssi Vide (*P*) = faux
Dépiler (Empiler (*P, e*)) = *P*
Sommet (Empiler (*P, e*)) = *e*
Vide (Pile_vide) = vrai
Vide (Empiler (*P, e*)) = faux

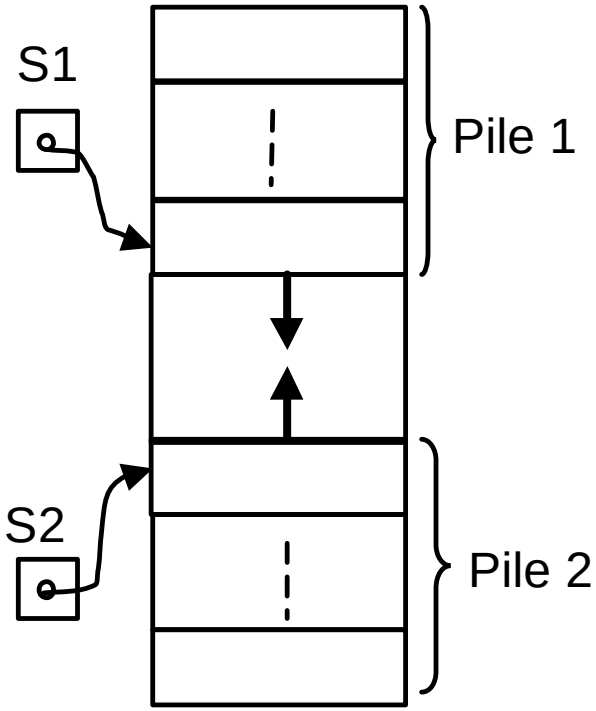
Implémentations possibles

1 pile

2 piles

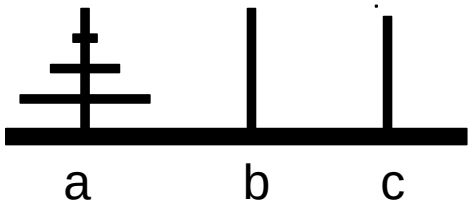
> 2 piles





Listes chaînées

Tours de Hanoï



Coup : *x*-*y* = déplacement d ’un disque de *x* vers *y*

H (*n, x, y, z*) = suite de coups pour déplacer *n* disques de *x* vers *y* avec le baton intermédiaire *z*

$$H(n, x, y, z) = \begin{cases} x\text{-}y & \text{si } n = 1 \\ H(n\text{-}1, x, z, y) \,.\, x\text{-}y \,.\, H(n\text{-}1, z, y, x) & \text{si } n > 1 \end{cases}$$

```
fonction  H (n, x, y, z) ;   / *  version itérative */
début
        P ← Pile_vide ;   Empiler (P, (n, x, y, z)) ;
        tant que non  Vide (P)  faire début
            e ← Sommet (P) ;   Dépiler (P) ;
            si  e = x'-y' ou e = (1, x', y', z')  alors
                 écrire (x'-y')
            sinon début     /*  e = (n, x', y', z') */
                 Empiler (P, (n-1, z', y', x')) ;
                 Empiler (P, x'-y') ;
                 Empiler (P, (n-1, x', z', y')) ;
            fin
        fin
fin .
```

Pile en table


```
Type  Pile = structure {
                    somm : -1 .. MAX ;
                    elt : table [0 .. MAX]  of  Element ;
              }

fonction  Pile_vide (P Pile) :  Pile ;
début  P.somm ← -1 ;  retour  (P) ;
fin ;

fonction Empiler (P Pile, e Element) : Pile ;
début  si  P.somm = MAX  alors  erreur
       sinon  début
          P.somm ← P.somm + 1 ; P.elt [P.somm] ← e ;
          retour  (P)
       fin
fin  ;
```

```
fonction  Dépiler (P Pile) : Pile ;
début  si  Vide (P)  alors  erreur ;
        sinon  début
                P.somm ← P.somm - 1 ;
                retour (P) ;
        fin
fin  ;

fonction  Sommet (P Pile) : Element ;
début  si  Vide (P)  alors  erreur
        sinon  retour (P.elt [P.somm]) ;
fin  ;

fonction  Vide (P Pile) : booléen ;
début retour  (P.somm = -1) ;
fin  ;
```

File

UMLV ©

Liste d'attente, queue
liste FIFO : « First In First Out »

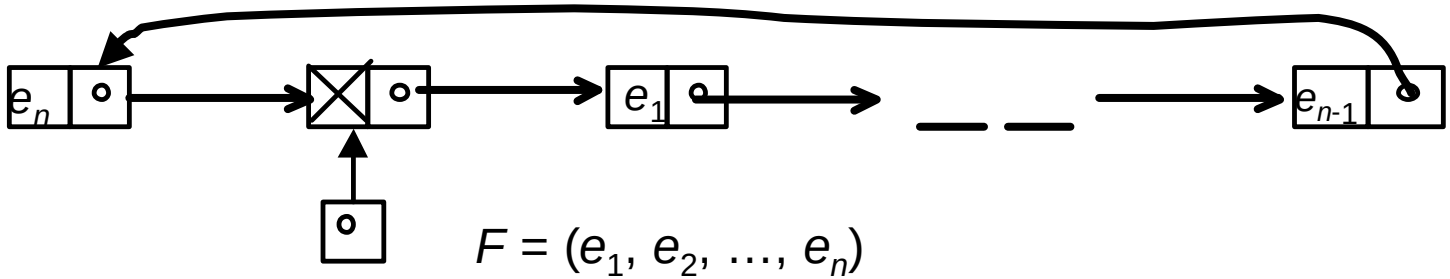
Opérations

File_vide : ® File
Ajouter : File x Element ® File
Enlever : File ® File
Premier : File ® Element
Vide : File ® Booléen

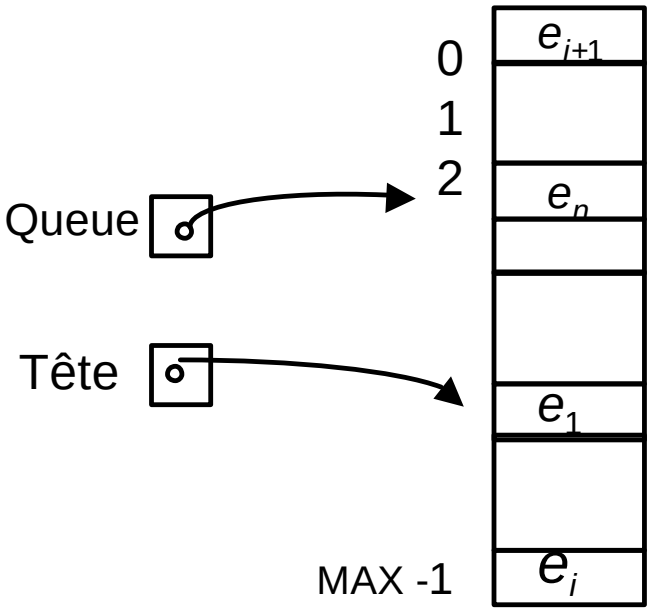
Axiomes

Enlever (*F*) et Premier (*F*) définis ssi Vide (*F*) = faux
Vide (*F*) = vrai ⇒ Premier (Ajouter (*F, e*)) = *e*
Vide (*F*) = faux ⇒ Premier (Ajouter (*F, e*)) = Premier (F)
Vide (*F*) = vrai ⇒ Enlever (Ajouter (*F, e*)) = file_vide
Vide (*F*) = faux ⇒ Enlever (Ajouter (*F, e*)) = Ajouter (Enlever (*F, e*))
Vide (file_vide) = vrai
Vide (Ajouter (*F,e*)) = faux

Implémentations possibles



$F = (e_1, e_2, \ldots, e_n)$



Succ (i) = i + 1 mod MAX

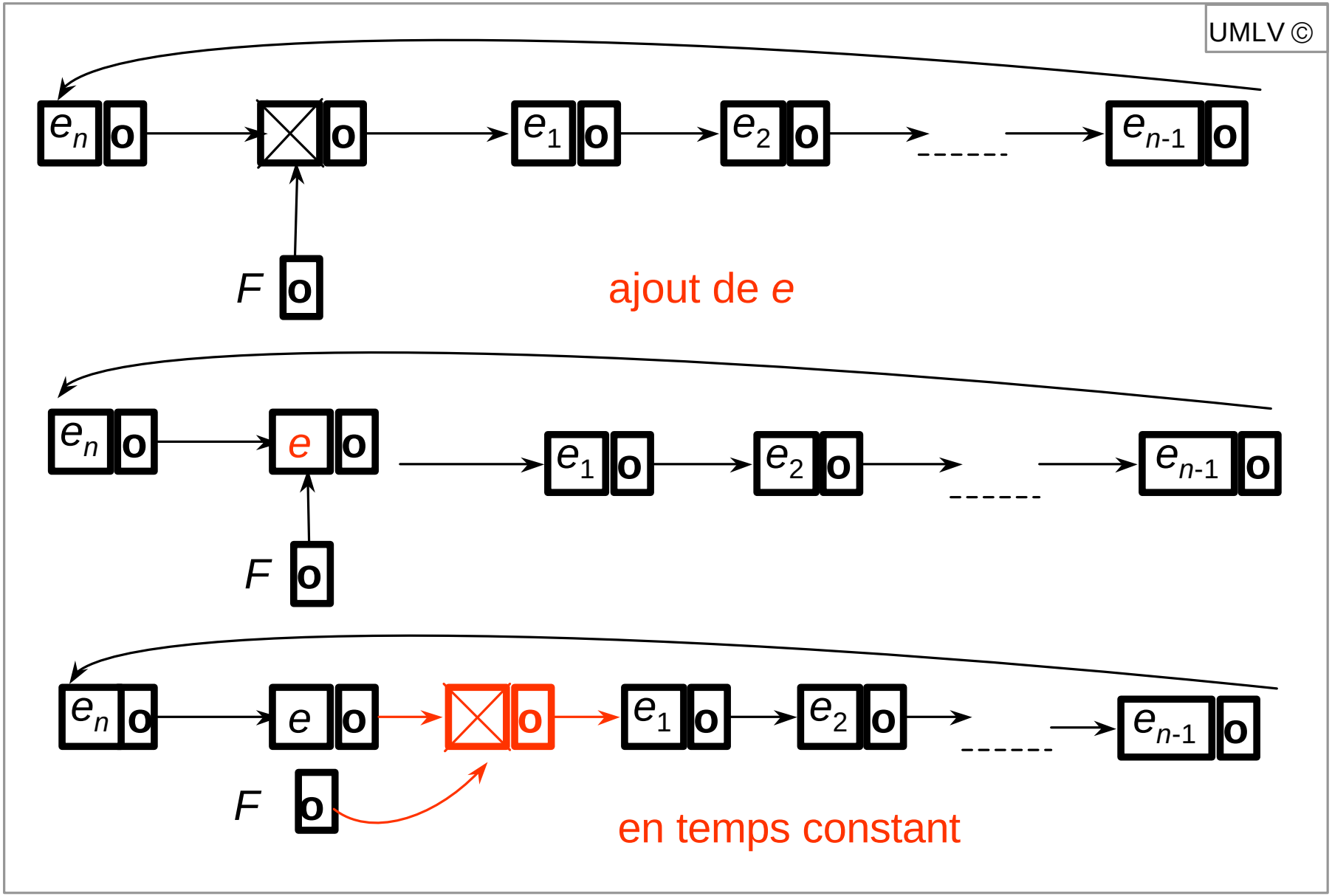
Condition :
MAX ≥ max | F | + 1
F

File Chaînée


```
Types File = cellule * ;
        cellule = structure {
                        elt  : Element ;   succ  : File ;
                  }

fonction  File_vide (F File)  : File ;
début
        créer une cellule d'adresse F ;
        F->succ  ¬  F ;   retour ( F ) ;
fin ;

fonction Ajouter (F File, e Element)  : File ;
début
        F->elt  ¬  e ;   Transfert (F, LIBRE, F, Tête (LIBRE)) ;
        F  ¬  F->succ ;   retour ( F)
fin ;
```


UMLV ©
e_n o
o
e_1 o
e_2 o
e_{n-1} o
F o
ajout de e
e_n o
e o
e_1 o
e_2 o
e_{n-1} o
F o
e_n o
e o
o
e_1 o
e_2 o
e_{n-1} o
F o
en temps constant

```
fonction Enlever  (F File )  : File ;
début
      Transfert ( LIBRE, F, Tête ( LIBRE), F ) ; retour ( F) ;
fin ;

fonction  Premier ( F File )  : File ;
début
      retour ( F->succ->elt ) ;
fin ;

fonction Vide ( F File )  : Booléen ;
début
      retour ( F = F->succ ) ;
fin ;
```

Ensemble

Ensembles de base

Sous-ensembles finis de E

Opérations principales

Union, Intersection, Partition, …

Ens_vide : ® Ens
Ajouter : Ens x Elément ® Ens
Enlever : Ens x Elément ® Ens
Elément : Ens x Elément ® Booléen
Vide : Ens ® Booléen

Implémentations

dépendantes du type d'opérations élémentaires

Vecteurs booléens, Tables, Tables hachées,
Listes chaînées, Arbre de recherche, ...

Polynomes

Ensemble

$$\mathbb{R}[X] = \left\{ \sum_{i=0}^{n} a_i x^i \quad / n \geq 0 , a_i \in \mathbb{R} \right\}$$

Opérations

somme : Poly x Poly ® Poly

produit : Poly x Poly ® Poly

Valeur : Poly x Réel ® Réel, …

Axiomes

structure d'anneau, …

Implémentations possibles