Edgar Morin - La pensée complexe

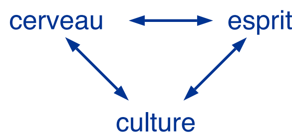
Nous vivons sous l'empire des principes de disjonction, de réduction et d'abstraction dont l'ensemble constitue ce que j'appelle « le paradigme de simplification ».

Le tout est plus que la somme des parties (émergence)

L'intelligence aveugle détruit les ensembles et les totalités, elle isole tous ses objets de leur environnement.
[…/…] Tandis que les média produisent la basse crétinisation, l'Université produit la haute crétinisation.

La difficulté de la pensée complexe est qu'elle doit affronter le fouillis (le jeu infini des inter-rétroactions), la solidarité des phénomènes entre eux, le brouillard, l'incertitude, la contradiction.

Nous pouvons élaborer quelques outils conceptuels […/…] que nous pouvons utiliser. Ainsi, au paradigme de disjonction/réduction/unidimensionnalisation, il faudrait substituer un paradigme de disjonction/conjonction qui permette de distinguer sans disjoindre, d'associer sans identifier ou réduire.



Edgar Morin
Introduction à la pensée
Complexe

Stockage de données → tableaux

→ N données (contiguës) du même type

→ accès à la i-ème donnée en $O(1)$ → temps constant

T[i] ⇔ *(T+i)

→ augmentation / diminution N : $O(N)$

→ stockage contigu ⇒ organisation "linéaire" des données

→ stockage hiérarchique (arbres

accès ($O(i)$)

mod. taille $O(1)$

listes
piles
files

≠ ordre d'accès aux données

+ tables hachage

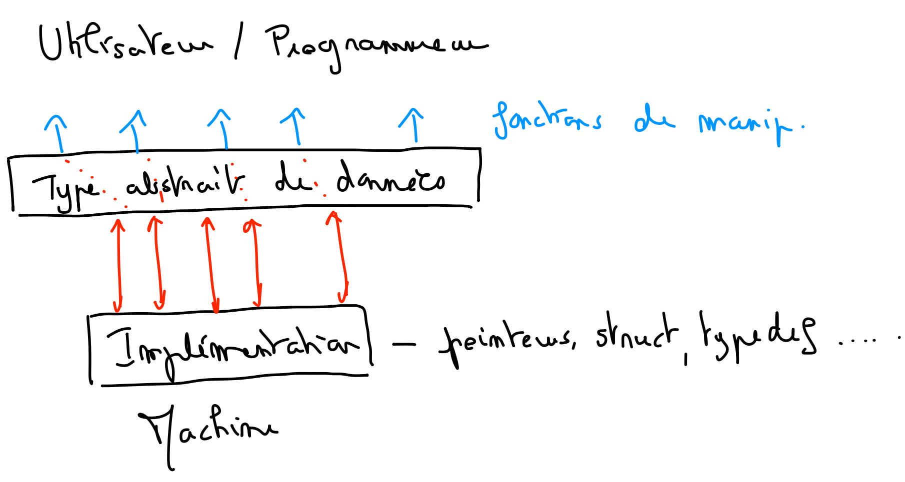
I. Types abstraits de données

Type abstrait de données : (TAD) spécification d'un type (≠ implémentation)

↓

se fait en donnant un ensemble de fonctions qui permettent de manipuler le type ⊕ leurs interactions

vient du monde de la logique
|
preuves de prog
|
prog. fonctionnelle



Type abstrait de données :

- → fonctions de manipulation du type
- → pré-conditions (conditions devant être satisfaites pour chaque fonction)
- → axiomes (relations entre les fonctions)

II. Type Listes

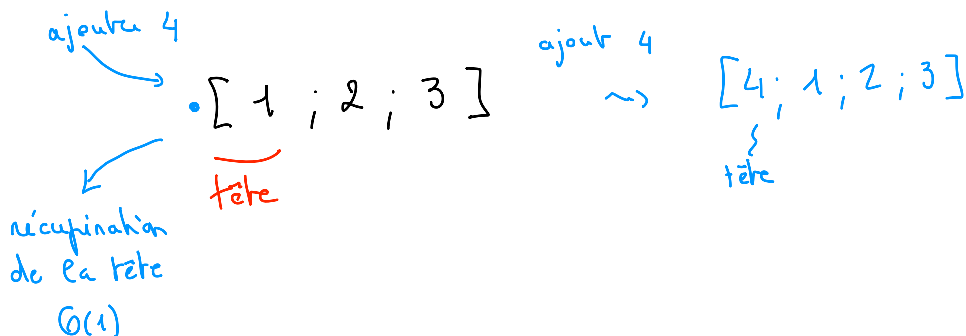
① Intuition

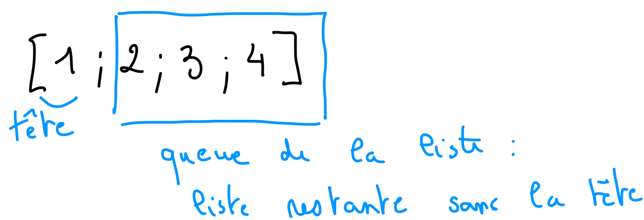
- ⇝ stockage linéaire / séquentiel des données
- ⇝ ajout et (recup.) suppression d'1 donnée en temps $O(1)$ ⊕ (≠ tableaux)
- ⇝ contre-partie : accès à la ième donnée : en temps $O(i)$ ⊖

⇝ ordre d'accès :

- → ajout en tête
- → récupération de donnée en tête

Notation intuitive :





② Type abstrait de données Listes

⇝ Fonctions du TAD Listes (de données de type Data) — int, float, int *, ...

- Liste créer_liste_vide ( )
- bool est_liste_vide (Liste)
- Liste ajouter (Data, Liste) ⟶ ajouter (4, [1;2;3]) ⇝ [4;1;2;3]
- Data tete (Liste) ⟶ tete ([4;1;2;3]) ⇝ 4
- Liste queue (Liste) ⟶ queue ([4;1;2;3]) ⇝ [1;2;3]

⇝ Pré-conditions

- tete(l) si NON(est_liste_vide(l))
- queue(l) "

implémentation ⟹ assert

C/C++ assert(cond)

⇝ Axiomes

$$\begin{cases} \text{est_liste_vide}(\text{creer_liste_vide}()) == \text{VRAI} \\ \text{est_liste_vide}(\text{ajouter}(e, \ell)) == \text{FAUX} \\ \text{tete}(\text{ajouter}(e, \ell)) == e \\ \text{queue}(\text{ajouter}(e, \ell)) == \ell \end{cases}$$

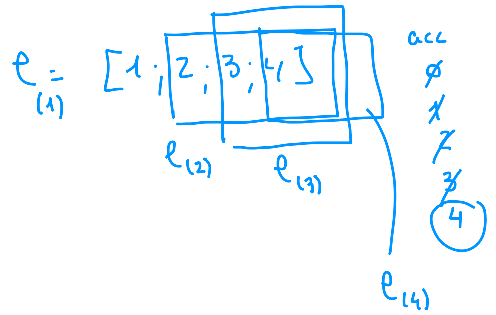
→ tests unitaires lorsqu'on implémente

3) Exemples d'algorithmes manipulant des listes

a) Longueur d'une liste

```
int longueur(Liste l)
{   int acc = 0;
    while (!est_liste_vide(l))
    {
        l = queue(l);
        acc++;
    }
    return acc;
}
```

$\ell_{(4)}$; $\ell_{(5)} = []$



Version récursive

BROUILLON

a) Cas général :

$\ell = [\bullet ; \underbrace{\ldots\ldots\ldots}_{\text{queue}(\ell)}]$

$\text{longueur}(\ell) = 1 + \text{longueur}(\text{queue}(\ell))$

b) Cas terminaux :

liste vide ⇝ renvoyer : 0

CODE

```
int longueur(Liste l)
{
    if (est_liste_vide(l))
        return 0;
    else
        return 1 + longueur(queue(l));
}
```

e) Rechercher un élément

voir TD

---

langage ⟷ esprit

Langage humain :
langage à double articulation
-> mots => phrases
-> phonèmes => mots

-> Intelligence animale (sans langage)

-> Une partie de la pensée est sub-linguistique

Chomsky, Piaget, Quine

« La pensée, à partir d'un certain seuil de complexité, est inséparable du langage. »

Nécessaire pour :
- concepts
- abstractions
- raisonnement

Langage au centre de :

- computation / cogitation
- inné / acquis
- individuel / collectif
  - -> transmission d'une mémoire, savoirs, normes, injonctions
  - -> création d'une culture, lien individus / société

Ex : une chaise, la beauté, un groupe commutatif …

Concrètement : message demandant la signature de contrats d'études à 2 jours de la deadline, un vendredi soir, sans avoir répondu aux demandes, pendant 2 mois, permettant de préparer ces contrats dans les temps pour 40 personnes :

Bonjour Mme Bac,

Voici mes contrats d'études.

Cordialement,

Bonjour Madame ,

J'espère que ce message vous trouve bien. Je tiens à m'excuser sincèrement pour le retard dans ma réponse. Actuellement, je finalise mes choix d'universités pour l'année prochaine et je souhaite partager cette information avec vous.

En pièces jointes, vous trouverez la liste des universités que j'ai sélectionnées. Je m'excuse de nouveau pour le délai dans cette communication.

Je vous remercie beaucoup pour votre compréhension et votre patience.

Cordialement,

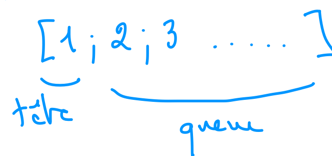
Bonsoir,
Je suis conscient de me réveiller bien tard, que nous sommes vendredi soir et donc que vous êtes en WE, que je n'ai pas pris part à la construction des contrats que vous avez organisée pour éviter justement de nous retrouver dans cette situation. Je suis conscient que vu le nombre d'élèves à gérer, une réaction tardive n'est pas envisageable à l'échelle du groupe.
Néanmoins, suite à notre conversation de mercredi, je regrette de ne pas tenter de poser ma candidature.
J'ai donc préparé mes contrats. Je ne sais pas si, vu la situation, vous accepterez de les signer, mais je me permets tout de même de vous les transmettre. Par avance : je comprends parfaitement que vous refusiez de les signer.
Avec mes excuses pour le retard et bien cordialement.

III. Implémentations des listes

Il y a essentiellement 2 implémentations :

→ par tableaux (cf. module fourni en TP)

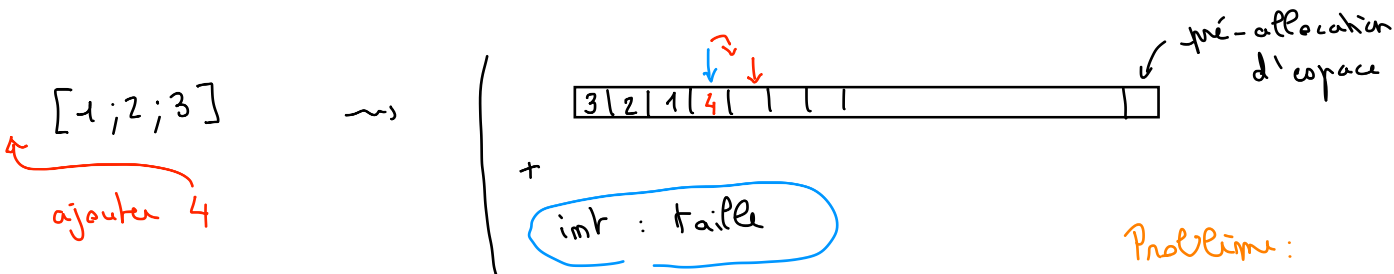
→ par listes chaînées



TAD

→ creer_liste_vide
→ est_liste_vide
→ ajouter
→ tete
→ queue

① Implémentation par tableaux



Problème : listes non bornées

quand le tableau est plein : il faut pouvoir augmenter sa taille

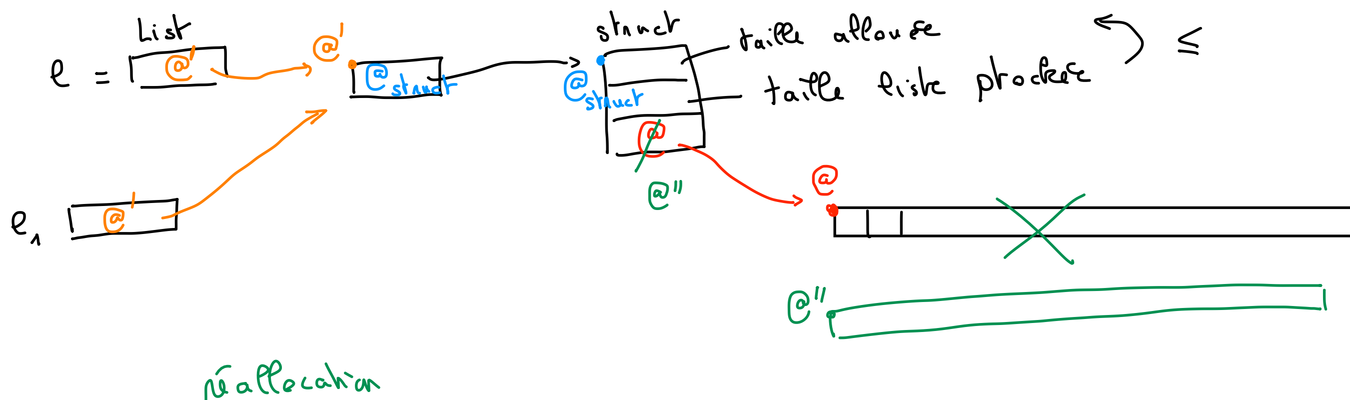
l ⇝ liste

$l_1 = l;$

l = ajouter$(4, l);$

l_1 ?

pour pouvoir gérer proprement les realloc à la volée :



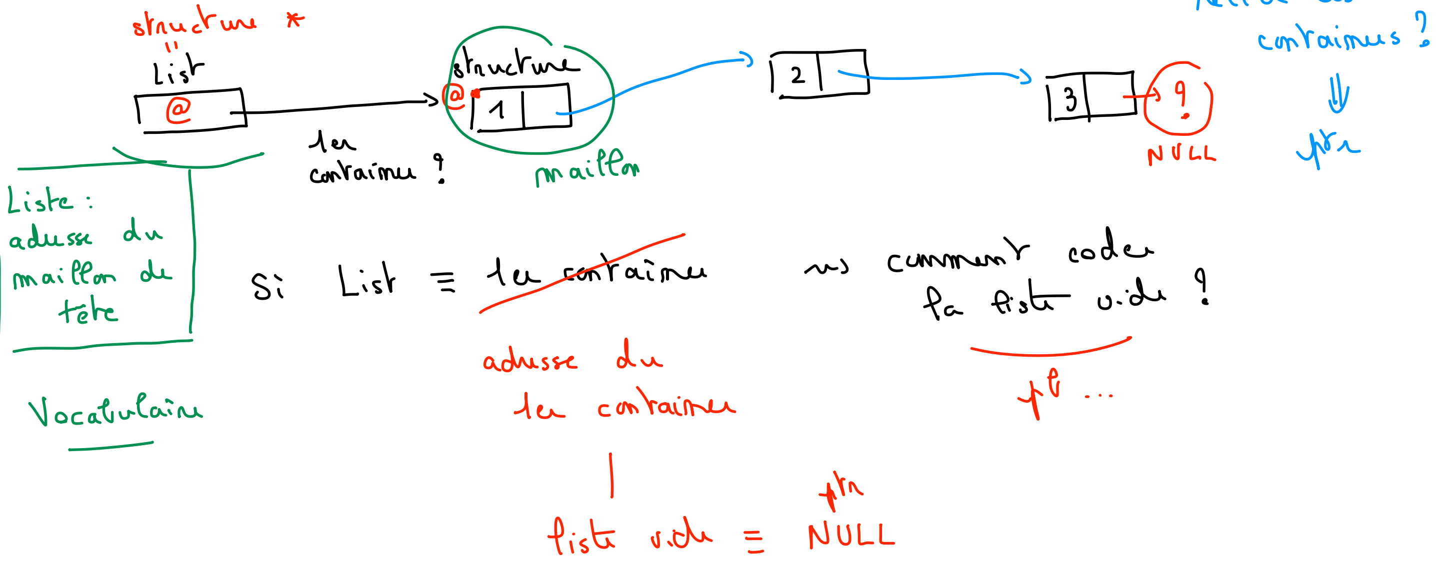
cf. cours piles / files

2) Implémentation par listes chaînées — 90%

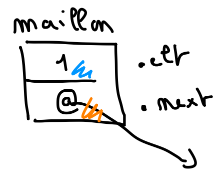
Implémentation : [1 ; 2 ; 3] → ordre de liste

↓

1 conteneur par donnée



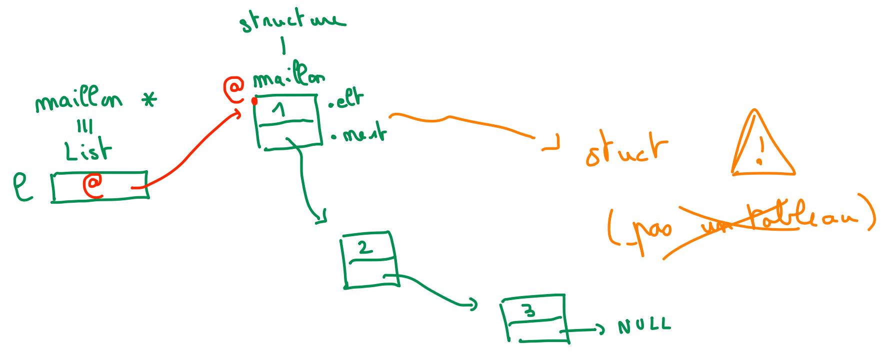

```
typedef int Data;
struct _maillon {
    Data elt;
    struct _maillon * next; } ;
typedef struct _maillon maillon;  // permet de créer un type maillon (plus court que struct maillon)
typedef maillon * List;  // liste : @ du maillon de tête
```



⟺

```
typedef struct _maillon {
    Data elt;
    struct _maillon * next; } maillon, * List;
```

ex : l = [1;2;3]



Implémentation du TAD :

- List creer_liste_vide ()

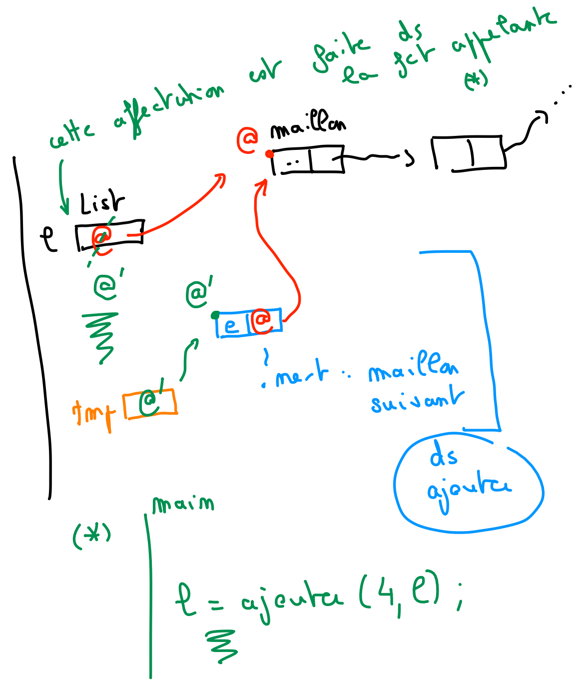
```c
List creer_liste_vide ()
{
    return NULL;
}
```

- bool est_liste_vide (List l)

```c
bool est_liste_vide (List l)
{
    return l == NULL;
}
```

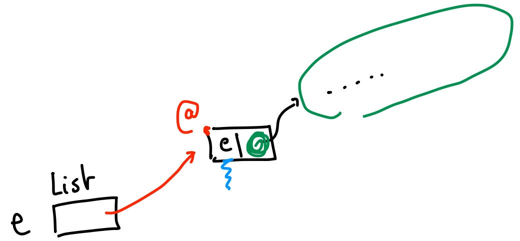
- List ajouter (Data e, List l) — List → renvoyer @'

```c
List ajouter (Data e, List l)
{
    maillon * tmp;
    tmp = malloc(sizeof(maillon));
    // Remplir le champ elt
    (*tmp).elt = e;
    // Equivalent à :  tmp -> elt = e;
    // Remplir le champ next
    tmp -> next = l;
    return tmp;  // @'
}
```



- Data tete (List l)

```c
Data tete(List l)
{   assert (!est_liste_vide (l));
    return l -> elt ;
}
```

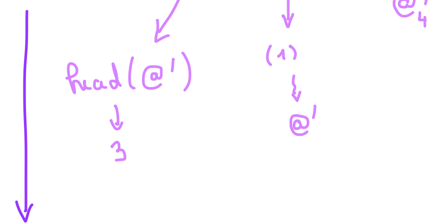


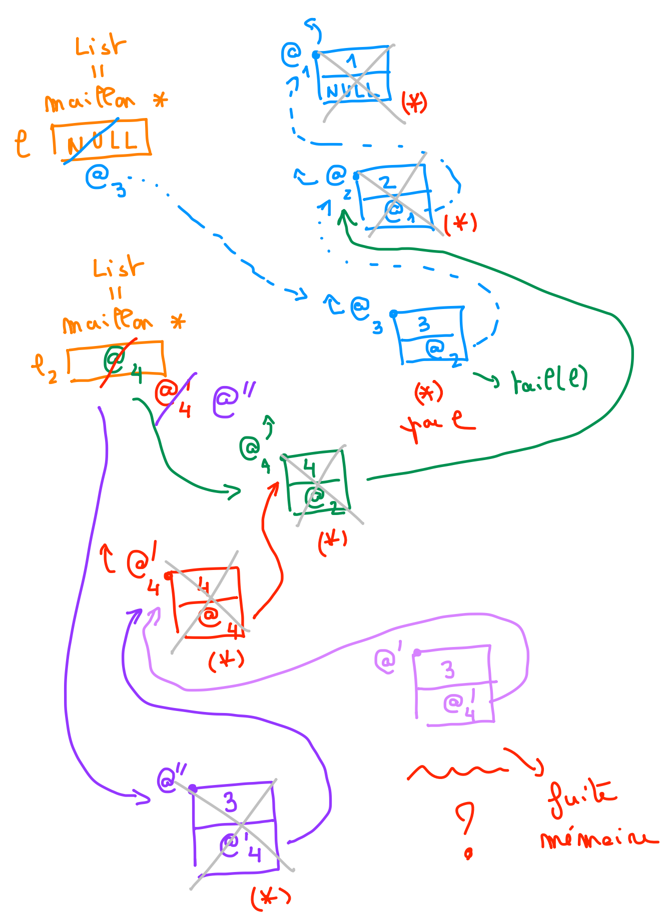
- List queue (List l)

```c
List queue (List l)
{   assert (!est_liste_vide (l));
    return l -> next;
}
```

maillon * (mon init...)

```
List l = create_empty_list(), l2 ;        (1)

l = add(3, add(2, add(1, l))) ;           (2)   l : [3;2;1]

l2 = add(4, tail(l)) ;                    (3)   ~> l2 : [4;2;1]

l2 = add(head(l2),l2) ;                   (4)   l2 : [4;4;2;1]

l2 = add(head(add(3,l2)), l2);            (5)
```





$l_2 : [3;4;4;2;1]$

accessible à la fin (*)

free(l_2)

free(l) ?