

Licence3 Informatique CDA

UE CASD

Partie « Algorithmique et Structures de Données »

V.-A.Nicolas
vnicolas@univ-brest.fr

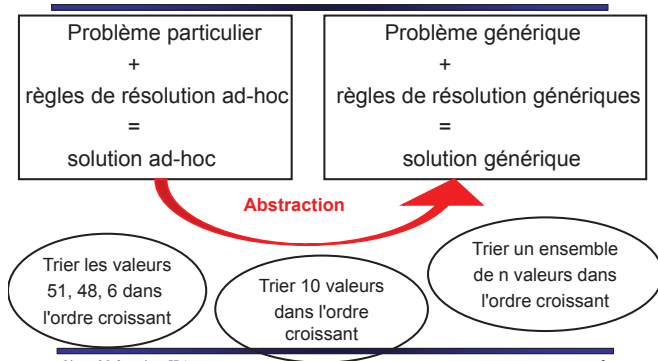
Introduction

- Algorithmique = cœur de l'informatique
- Rappels sur quelques concepts fondamentaux en informatique :
 - récursivité et itération
 - complexité et optimisation des algorithmes (illustration sur les algorithmes de tri)
 - modélisation des données (listes, tableaux, listes chaînées, piles, files...)
- **Bibliographie :**
 - "Mathématiques pour l'informatique", A. Arnold et I. Guessarian, Masson
 - "Introduction à l'algorithmique", T. Cormen, C. Leiserson et R. Rivest, Dunod

Licence3 Informatique CDA

2

Comment résoudre un problème ?

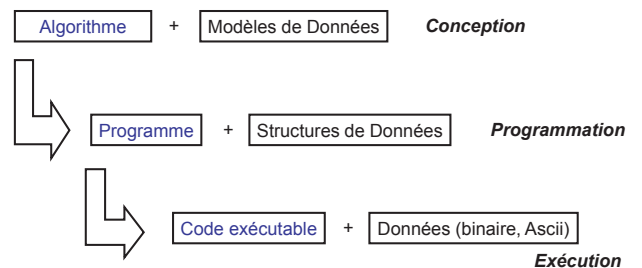


Licence3 Informatique CDA

3

Algorithmes et Modèles de Données

Règles de résolution Problème + solution



Licence3 Informatique CDA

4

Algorithmes et Structures de Données

- **Qu'est-ce qu'un algorithme ?**
 - Un algorithme est un ensemble de règles ou de procédures bien défini qu'il faut suivre pour obtenir la solution d'un problème ou accomplir une tâche en un nombre fini d'étapes.
- **Structures de Données ?**
 - *Données* : entrées, sorties, données intermédiaires manipulées par un algorithme
 - *Modèle de données* : abstraction de données
 - *Structure de données* : implantation d'un modèle de données dans un langage de programmation (impératif, déclaratif - fonctionnel ou logique -, objet)

Licence3 Informatique CDA

5

Algorithmique en conception

- **Objectif** : permettre de résoudre des problèmes (complexes)
- **En proposant :**
 - Une base théorique : calculabilité, décidabilité
 - Différentes méthodes de résolution (récurrence, "diviser pour résoudre", algorithmes d'approximation, algorithmes gloutons, programmation dynamique...)
 - Des métriques : *complexité* en temps, *complexité* en espace...
 - Une classification des algorithmes et problèmes étudiés : P, NP, NP-complet
 - Des techniques pour vérifier la *correction (ou sûreté)* et la *vivacité* des algorithmes construits

Licence3 Informatique CDA

6

Cadre formel de l'induction (récurrence)

Travailler sur un ensemble muni d'un *bon ordre* (i.e. *ordre total bien fondé*).

Définition :

Une relation d'ordre \mathcal{R} sur un ensemble E est un *ordre total* si \mathcal{R} vérifie

$$\forall e, e' \in E, e \neq e' \Rightarrow (e \mathcal{R} e' \text{ ou } e' \mathcal{R} e).$$

Définition :

Une relation d'ordre sur un ensemble E est *bien fondée* s'il n'y a pas de suite infinie strictement décroissante d'éléments de E .

Exemples

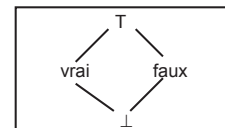
• Bons ordres :

- Ordre naturel sur \mathbb{N} : $0 < 1 < \dots < n$
- Ordre alphabétique sur les lettres : $a < b < c < \dots < y < z$
- Ordre lexicographique sur les chaînes : $ab < bf$, $aa < ab$, $al < ca$

• Ordre total mais pas bien fondé : ordre naturel sur \mathbb{R} ($-\infty$)

• Ordre bien fondé mais non total :

- Treillis



Premier principe d'induction sur \mathbb{N} (principe de récurrence mathématique)

Théorème :

Soit $P(n)$ une propriété dépendant de l'entier n .

Si les deux conditions suivantes sont vérifiées :

(B) $P(0)$ est vrai,

(I) $\forall n \in \mathbb{N}, (P(n) \Rightarrow P(n+1))$,

alors $\forall n \in \mathbb{N}, P(n)$ est vrai.

\Rightarrow Une *preuve par récurrence* est l'application de ce théorème.

Exemple de preuve par récurrence

Montrer que la somme des entiers de 1 à n est égale à $n(n+1)/2$.

$$P(n) : (\sum_{i=1..n} i = n(n+1)/2)$$

Preuve :

(B) $P(1)$ est vraie ?

$$P(1) : \sum_{i=1..1} i = 1 \cdot (1+1)/2 ?$$

$$1 = 1 \cdot 2/2 ? \text{ Ok, } \mathbf{P(1) \text{ est vraie.}}$$

(I) $\forall n > 0 \in \mathbb{N}, (P(n) \Rightarrow P(n+1)) ?$

Suite de l'exemple : cas de récurrence

(I) $\forall n > 0 \in \mathbb{N}, (P(n) \Rightarrow P(n+1)) ?$

ou : **sachant $P(n)$, peut-on démontrer $P(n+1)$?**

$$P(n+1) : \sum_{i=1..(n+1)} i = (n+1)((n+1)+1)/2 ?$$

$$\begin{aligned} \bullet \quad \sum_{i=1..(n+1)} i &= (\sum_{i=1..n} i) + (n+1) \\ &= (n(n+1)/2) + (n+1) \\ &= (n(n+1)/2) + (2(n+1)/2) \\ &= (n(n+1) + 2(n+1))/2 \\ &= (n+1)(n+2)/2 \\ &= (n+1)((n+1)+1)/2 \end{aligned}$$

Hypothèse
de récurrence
 $P(n)$

Ok, $P(n+1)$ est vraie sachant $P(n)$.

Second principe d'induction sur \mathbb{N}

Théorème :

Soit $P(n)$ une propriété dépendant de l'entier n .

Si la proposition suivante est vérifiée :

(I') $\forall n \in \mathbb{N}, ((\forall k < n, P(k)) \Rightarrow P(n))$

alors $\forall n \in \mathbb{N}, P(n)$ est vrai.

- **Remarque 1** : le cas de base est "caché" dans (I')
- **Remarque 2** : les deux principes d'induction sont équivalents sur \mathbb{N} . Le second est plus utile pour les preuves de propriétés à récursivité multiple.

Exemple de preuve par récurrence selon le second principe

Montrer que tout entier $n > 1$ est décomposable en un produit de nombres premiers.

$P(n)$: (n est décomposable en produit de nombres premiers)

Preuve : (I) $\forall n \in \mathbb{N}$, $((\forall k < n, P(k)) \Rightarrow P(n))$?

ou : sachant $P(k) \forall k < n$, peut-on démontrer $P(n)$?

Soit $n > 1$. Deux cas possibles :

- Soit n est premier : CQFD, **$P(n)$ est vraie.**
- Soit n n'est pas premier et alors $\exists a, b \in [2, n-1]$ tels que $n = a \cdot b$.
Par hypothèse de récurrence, $P(a)$ et $P(b)$ sont vraies, i.e. a et b sont décomposables en produits de nombres premiers.
Donc, n est décomposable en le produit des décompositions de a et b et donc **$P(n)$ est vraie.**

Définitions inductives (ou récursives)

Définition inductive de l'ensemble E :

L'ensemble E est composé :

- (B) d'éléments simples de base,
- (I) d'éléments définis récursivement à partir d'autres éléments (moins complexes) de E et d'un ensemble d'opérations de construction K sur E .

Remarques :

- Attention ! Pas de suite infinie décroissante de constructions d'éléments, on doit toujours finir par atteindre (B).
- Tout élément de E peut ainsi s'obtenir à partir de la base (B) en appliquant un nombre fini d'étapes inductives (I).

Exemples de définitions récursives (1)

Exemple : La partie X de \mathbb{N} définie inductivement par

(B) $0 \in X$

(I) $n \in X \Rightarrow n+1 \in X$

correspond en fait à \mathbb{N} tout entier.

Définition d'une expression arithmétique :

Une expression arithmétique est

(B) soit un entier, un réel ou une variable,

(I) soit, si E_1 et E_2 sont des expressions arithmétiques,

$(E_1 + E_2)$ $(E_1 - E_2)$

$(E_1 * E_2)$ (E_1 / E_2)

$(-E_1)$

Exemples de définitions récursives (2)

Définition d'une chaîne de caractères :

Une chaîne de caractères est

(B) soit la chaîne vide (ϵ),

(I) soit un caractère suivi d'une chaîne de caractères.

Définition d'une liste d'éléments de type t :

Une liste d'éléments de type t est

(B) soit la liste vide (nil),

(I) soit un élément de type t suivi (*constructeur*.) d'une liste d'éléments de type t .

Exemples de listes : nil , $3.nil$, $7.2.5.nil$, $toto.titi.nil$, $'a'.k'.nil$

Exemples de définitions récursives (3)

Définition d'une suite de chiffres/lettres :

Une suite de chiffres/lettres est

(B) soit rien,

(I) soit une lettre ou un chiffre suivis d'une suite de chiffres/lettres.

Définition (non récursive) d'un identificateur :

Un identificateur est une lettre suivie d'une suite de chiffres/lettres.

Définition du langage de Dyck (bons parenthésages) :

Sur l'alphabet $\{ (,) \}$, le langage de Dyck D est défini par :

(B) $\epsilon \in D$,

(I) si x et y sont dans D , alors (x) et xy sont aussi dans D .

Preuve par induction structurelle

- La preuve par induction structurelle d'une propriété $P(x)$, $\forall x \in E$, est une preuve par récurrence calquée sur la définition inductive de E .

• Preuve :

Si les conditions suivantes sont vérifiées :

– $\forall x \in B$, $P(x)$ est vraie

– $\forall k \in K, (\forall x \in \bar{x}, P(x)) \Rightarrow P(k(\bar{x}))$

alors $\forall x \in E$, $P(x)$ est vraie.

($\bar{x} \in E^n$,
est un n -uplet sur E)

Exemple de preuve par induction structurelle

Montrer que tout mot du langage de Dyck a autant de parenthèses ouvrantes que fermantes.

$P(m)$: m a autant de parenthèses ouvrantes que fermantes.

Preuve : montrons

(B) $\forall x \in B$, $P(x)$ est vraie

(I) $\forall k \in K$, $(\forall x \in \bar{x}, P(x)) \Rightarrow P(k(\bar{x}))$

- (Base) **$P(\epsilon)$ est vraie** (aucune parenthèse).
- (Induction) Soit $m = (x)$, soit $m = xy$ avec x et y dans D :
 - si $m = (x)$, **$P(m)$ est vraie** par HR car $P(x)$ est vraie et on rajoute autant de parenthèses ouvrantes que fermantes pour construire m ,
 - si $m = xy$, **$P(m)$ est vraie** par HR car $P(x)$ et $P(y)$ sont vraies et on ne rajoute aucune parenthèse.

Utilité de la récursivité en informatique

- Définition et manipulation d'ensembles de données potentiellement infinis (*définitions inductives ou récursives*)
- Traitement des données structurellement récursives (*algorithmes récursifs*)
- Répétition finie ou non d'une même ou de différentes versions d'une tâche (*algorithmes récursifs ou itératifs*)

Récursivité vs itération

- **Itération** : répétition d'un traitement un certain nombre de fois, ou sous certaines conditions qui varient après chaque "tour" d'itération
- **Récursivité** : définition d'un concept, directement ou indirectement, en fonction de lui même. Notion proche de la répétition en faisant varier la taille du problème considéré

Définition itérative vs définition récursive : ordre lexicographique sur les chaînes

- **Définition itérative** :
Soit les chaînes $u = u_1u_2\dots u_n$ et $v = v_1v_2\dots v_m$.
 u est inférieure à v dans l'ordre lexicographique si l'une des deux conditions suivantes est vérifiée :
 - $n < m$ et $u_i = v_i$ pour tout $i = 1, 2, \dots, n$
 - $\exists i. 1 \leq i \leq \min(n, m)$ et $u_i < v_i$ (ordre alphabétique) et $u_j = v_j$ pour tout $j = 1, 2, \dots, i-1$
- **Définition récursive** :
(B1) la chaîne vide est inférieure à toute chaîne non vide
(B2) $\forall v, \forall w, c_1v$ est inférieure à c_2w si $c_1 < c_2$ (ordre alphabétique)
(I) $\forall c, cv < cw$ si v est inférieure à w

Conception d'algorithmes récursifs

- **Algorithme récursif = définition récursive**
 - Condition d'arrêt (cas de base)
 - Cas général (récursif)
- **Bon fonctionnement d'un algorithme récursif = preuve par induction structurelle**
 - Arrêt : fonction de convergence (vivacité)
 - Sûreté : invariant, preuve par récurrence
 - Complexité : nombre d'appels récursifs induits
- **Erreurs courantes** : l'algorithme ne s'arrête pas, le résultat est faux, il ne se passe rien...
→ Une méthodologie est nécessaire !

Méthodologie de conception d'algorithmes récursifs

- **Appliquer une analyse par cas à la spécification du problème**
 - Extraire le(s) cas particuliers
 - Extraire le(s) cas général(aux)
- **Construire un algorithme récursif répondant au problème posé**
 - Déterminer les différents paramètres d'entrée et de sortie avec leurs types
 - Déterminer le(s) paramètre(s) sur le(s)quel(s) est basée la récurrence
 - Construire le(s) cas particuliers (cas d'arrêt de la récursivité)
 - Construire le(s) cas général(aux)
- **Montrer la correction de l'algorithme (vivacité et sûreté) et évaluer sa complexité**

Méthodologie de conception d'algorithmes récursifs : correction de l'algorithme

Montrer la *correction* de l'algorithme :

- Montrer la *vivacité* (l'algorithme calcule quelque chose en un temps fini acceptable)
 - Montrer que quelle que soit la valeur des paramètres de l'algorithme lors de l'appel initial, la suite des valeurs des paramètres engendrés au cours des différents appels récursifs consécutifs converge toujours vers la valeur du cas d'arrêt (extraction d'une *fonction de convergence* strictement décroissante et atteignabilité du cas d'arrêt).
- Montrer la *sûreté* (l'algorithme répond bien au problème posé)
 - Montrer la correction du calcul pour le(s) cas d'arrêt
 - Montrer la correction du calcul pour le(s) cas général(aux)

Schéma général d'algorithme récursif

```
Si cond_arrêt1 alors res_cas_arrêt1
Sinon si cond_arrêt2 alors res_cas_arrêt2
    sinon ...
        sinon si cond_arrêt_n alors res_cas_arrêt_n
            sinon traitement_récursif
                fsi
            fsi ...
        fsi
    fsi
fsi
```

- Dans **traitement_récursif**, l'algorithme se rappelle lui-même
- traitement_récursif peut aussi être composé de différents cas récursifs

Le langage algorithmique

Le *langage algorithmique* est ce qu'on appelle un *pseudo-langage*. Il s'agit souvent d'un sous-ensemble bien déterminé d'un langage naturel (comme le français)

Pourquoi ? Pour la *formalisation* et la *généricité* des programmes

Exemple :

```
entier max (entier x, entier y)
{
    Entier m;
    Si x > y alors m ← x
        sinon m ← y
    Fsi;
    Retourner m;
}
```

Exemple d'algorithme récursif : somme des n premiers entiers naturels strictement positifs

- **Analyse par cas :**
 - Si $n=0 \rightarrow 0$
 - Sinon ($n>0$) $\rightarrow n + (\text{somme des } n-1 \text{ premiers entiers}) > 0$

- **Algorithme :**

```
entier somme (entier n)
{
    si n=0 alors retourner(0)
    sinon retourner(n + somme(n-1))
    fsi
}
```

Complexité en $O(n)$

Application de **somme** avec $n=5$

Appel initial : somme(5)

$\rightarrow 5 + \text{somme}(5-1) = 5 + \text{somme}(4)$
 $\rightarrow 5 + (4 + \text{somme}(3))$
 $\rightarrow 5 + (4 + (3 + \text{somme}(2)))$
 $\rightarrow 5 + (4 + (3 + (2 + \text{somme}(1))))$
 $\rightarrow 5 + (4 + (3 + (2 + (1 + \text{somme}(0)))))$
 $\rightarrow 5 + (4 + (3 + (2 + (1 + 0))))$
 $= 5 + (4 + (3 + (2 + 1))) = 5 + (4 + (3 + 3))$
 $= 5 + (4 + 6) = 5 + 10$
 $= 15$

Preuve de l'algorithme **somme**

- **Vivacité :**
 - Appel initial somme(n), puis somme(n-1), puis somme((n-1)-1), etc. jusqu'à l'arrêt
 - Suite $n \rightarrow n-1 \rightarrow n-2 \rightarrow \dots \rightarrow 0$ *strictement décroissante et convergente vers 0 (cas d'arrêt)*
 - *Fonction de convergence* : $f(x) = x-1$ pour $x>0$
- **Sûreté : preuve par récurrence de $P(n)$**
 $P(n) : \text{somme}(n) = n(n+1)/2$
 - **cas d'arrêt $P(0)$** : $\text{somme}(0) \rightarrow 0$, et $0*(0+1)/2 = 0$ donc $P(0)$ ok.
 - **cas général** : On suppose $P(n)$ vraie. A-t-on $P(n+1)$ vraie ?
 $P(n+1) : \text{somme}(n+1) = (n+1)(n+2)/2$? Par H.R.

$\text{somme}(n+1) \rightarrow (n+1) + \text{somme}((n+1)-1) = n+1 + \text{somme}(n)$
 $= n+1 + n(n+1)/2 = (2(n+1) + n(n+1))/2 = (n+1)(n+2)/2$ ok.

Retour sur les listes

Définition d'une liste d'éléments de type t :

Une liste d'éléments de type t est

(B) soit la liste vide (*nil*),

(l) soit un élément de type t suivi (*constructeur .*) d'une liste d'éléments de type t (exemple : *x.l*).

Opérations primitives sur les listes :

car : liste \rightarrow élément

car(*nil*) = *indéfini*, car(*x.l*) = *x*

cdr : liste \rightarrow liste

cdr(*nil*) = *indéfini*, cdr(*x.l*) = *l*

Exemples : car(3.*nil*) = 3, cdr(7.2.5.*nil*) = 2.5.*nil*

Exemple d'algorithme récursif : longueur d'une liste l

• Analyse par cas :

– Si $l = \text{nil} \rightarrow 0$

– Sinon ($l = x.l'$) $\rightarrow 1 + (\text{longueur de } l')$

• Algorithme :

```
entier longueur (liste l)
{
  si l=nil alors retourner(0)
  sinon retourner(1 + longueur(cdr(l)))
}
fsi
```

Complexité
en $O(n)$

avec
 $n = \text{taille de } l$

Application de **longueur** sur la liste 2.3.1.6.1.*nil*

Appel initial : longueur(2.3.1.6.1.*nil*)

$\rightarrow 1 + \text{longueur}(\text{cdr}(2.3.1.6.1.\text{nil})) = 1 + \text{longueur}(3.1.6.1.\text{nil})$

$\rightarrow 1 + (1 + \text{longueur}(\text{cdr}(3.1.6.1.\text{nil})))$

$= 1 + (1 + \text{longueur}(1.6.1.\text{nil}))$

$\rightarrow 1 + (1 + (1 + \text{longueur}(\text{cdr}(1.6.1.\text{nil}))))$

$= 1 + (1 + (1 + \text{longueur}(6.1.\text{nil})))$

$\rightarrow 1 + (1 + (1 + (1 + \text{longueur}(\text{cdr}(6.1.\text{nil}))))))$

$= 1 + (1 + (1 + (1 + \text{longueur}(1.\text{nil}))))$

$\rightarrow 1 + (1 + (1 + (1 + (1 + \text{longueur}(\text{cdr}(1.\text{nil}))))))$

$= 1 + (1 + (1 + (1 + (1 + \text{longueur}(\text{nil}))))))$

$\rightarrow 1 + (1 + (1 + (1 + (1 + 0)))) = 1 + (1 + (1 + (1 + 1)))$

$= 1 + (1 + (1 + 2)) = 1 + (1 + 3) = 1 + 4 = 5$

Preuve de l'algorithme **longueur**

• Vivacité :

- Appel initial longueur(*l*), puis longueur(cdr(*l*)), puis longueur(cdr(cdr(*l*))), etc. jusqu'à l'arrêt
- Suite $l \rightarrow \text{cdr}(l) \rightarrow \text{cdr}(\text{cdr}(l)) \rightarrow \dots \rightarrow \text{nil}$ *strictement décroissante et convergente vers nil (cas d'arrêt)*
- *Fonction de convergence* : décroissance de taille de *l*

• Sûreté : preuve par récurrence de $P(l)$

$P(l) : \text{longueur}(l) = n$, avec n la taille de *l*, $\forall l$

– **cas d'arrêt** $P(\text{nil})$: longueur(*nil*) $\rightarrow 0$, donc $P(\text{nil})$ ok.

– **cas général** : On suppose $P(l)$ vraie. A-t-on $P(x.l)$ vraie ?

$P(x.l) : \text{longueur}(x.l) = n+1, \forall x \text{ de type } t ?$

$\text{longueur}(x.l) \rightarrow 1 + \text{longueur}(\text{cdr}(x.l)) = 1 + \text{longueur}(l)$

$= 1 + n$ ok.

Par H.R.

Modèle de données

- C'est une abstraction de données résultant en une collection d'éléments ayant des caractéristiques communes

- Un modèle de données est défini par :

- un ensemble d'éléments *E*
- un ensemble de propriétés, relations ou opérations sur *E*
- [un ensemble d'éléments particuliers de *E*]

Exemple 1 : définition inductive des entiers naturels

Modèle de données "EntNat" :

- Élément de base : 0

- Opération : Successeur

Succ: EntNat \rightarrow EntNat

tel que : $n \in \text{EntNat} \Rightarrow \text{Succ}(n) \in \text{EntNat}$

Exemples :

- 0 est un entier naturel : $0 \in \text{EntNat}$
- $\text{succ}(0) \in \text{EntNat}$ (= 1, entier naturel successeur de 0)
- $0, \text{succ}(0), \text{succ}(\text{succ}(0)), \dots, \text{succ}(\dots \text{succ}(0) \dots) \in \text{EntNat}$

Exemple 2 : modélisation d'un circuit électronique

Modèle de données "Circuit" :

- Éléments de base : vrai, faux
- Éléments complexes : formules booléennes
- Opérations de construction : et, ou, non

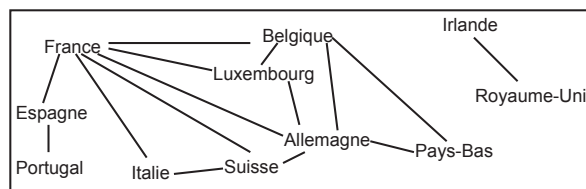
Exemple : la formule ((a et b) ou c) modélise le circuit :



Exemple 3 : modélisation d'une carte géographique

Modèle de données "Carte" :

- Éléments : graphes dont les noeuds sont des pays
- Propriétés : deux noeuds sont liés par un arc ssi les pays correspondants sont frontaliers



Exemple 4 : modèle de données "Tableau"

Modèle de données "Tableau" :

- Éléments : un tableau est une séquence d'éléments de type t indicés par un entier (début à 0)
- Propriétés :
 - Tous les éléments du tableau sont de même type (t)
 - L'accès à chaque élément du tableau est direct
 - Implantation efficace ("colle" à la mémoire)
- Opérations :
 - Création_tableau : taille_tableau, type_éléments \rightarrow tableau
 - Accès_elt_tableau : indice, tableau \rightarrow élément
 - Ajout_elt_tableau : indice, tableau, élément \rightarrow tableau

Exemple de manipulation d'objet du modèle de données "Tableau"

$T \leftarrow \text{Création_tableau}(3, \text{entier})$

T

--	--	--

$\text{Ajout_elt_tableau}(0, T, 5)$

T

5		
---	--	--

$\text{Ajout_elt_tableau}(2, T, 7)$

T

5		7
---	--	---

$\text{Ajout_elt_tableau}(1, T, 9)$

T

5	9	7
---	---	---

$x \leftarrow \text{Accès_elt_tableau}(1, T)$

$\text{Ajout_elt_tableau}(1, T, 4)$

T

5	4	7
---	---	---

$y \leftarrow \text{Accès_elt_tableau}(0, T)$

T

5	4	7
---	---	---

$z \leftarrow \text{Accès_elt_tableau}(1, T)$

T

5	4	7
---	---	---

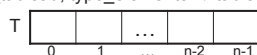
$u \leftarrow \text{Accès_elt_tableau}(2, T)$

$x:9, y:5, z:4, u:7$

Structure de données "Tableau"

Structure Tableau :

- Prédéfinie dans la plupart des langages de programmation
- **Création_tableau** : taille_tableau, type_éléments \rightarrow tableau
 - Entier $T[n]$
- **Accès_elt_tableau** : indice, tableau \rightarrow élément
 - $T[i]$
- **Ajout_elt_tableau** : indice, tableau, élément \rightarrow tableau
 - $T[i] \leftarrow \text{elt}$



Les différents schémas itératifs

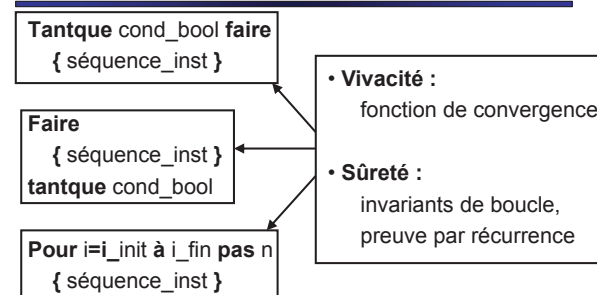


Illustration des schémas itératifs

```
entier facto (entier n)
{ entier i, f ← 1;
  Pour i=2 à n pas 1
    { f ← f*i; }
  retourner(f)
}
```

```
entier facto (entier n)
{ entier i ← 2, f ← 1;
  Tantque i ≤ n faire
    { f ← f*i;
      i ← i+1; }
  retourner(f)
}
```

```
entier facto (entier n)
{ entier i ← 1, f ← 1;
  Faire
    { f ← f*i;
      i ← i+1; }
  tantque i ≤ n;
  retourner(f)
}
```

Exemple de factorielle

```
entier facto (entier n)
{ entier i, f ← 1;
  Pour i=2 à n pas 1
    { f ← f*i; }
  retourner(f)
}
```

Sûreté :

P(k) : (à la fin de l'itération d'indice k, f vaut k!)

(B) k < 2 : pas d'itération et f = 1 = 0! = 1!

(B) k = 2 : une itération et f = 1*2 = 2 = 2!

(I) Supposons P(k). A-t-on P(k+1) ?

$$f_{k+1} = f_k * (k+1) = k! * (k+1) = (k+1)!$$

donc P(k+1) ok.

Vivacité :

- n < 2 : arrêt (après 0 itération)
- n ≥ 2 : ((n-2)/1+1) = n-1 itérations avant l'arrêt
- fonction de convergence : fc(i) = (n-i) converge vers 0

Recherche d'un élément dans un tableau (algorithme itératif séquentiel)

```
booléen recherche (t_elt e, t_elt T[n])
{ entier i ← 0;
  Tantque (i < n et T[i] ≠ e) faire
    { i ← i+1; }
  si i = n alors retourner(faux)
  sinon retourner(vrai)
  fsi
}
```

Complexité : O(n)

Recherche d'un élément dans un tableau **trié** (algorithme itératif dichotomique)

```
booléen recherche_dicho (t_elt e, t_elt T[n])
{ entier g ← 0, d ← n-1, i ← (g+d)/2;
  Tantque (g ≤ d et T[i] ≠ e) faire
    {
      si T[i] < e alors g ← i+1
      sinon d ← i-1
      fsi;
      i ← (g+d)/2;
    }
  si g > d alors retourner(faux)
  sinon retourner(vrai)
  fsi
}
```

Complexité :
O(log₂(n))

Algorithme utilisable
ssi T est trié en ordre
croissant

Recherche d'un élément dans un tableau (algorithme récursif séquentiel)

```
booléen recherche_rec (t_elt e, t_elt T[], entier i, entier n)
{
  si T[i] = e alors retourner(vrai)
  sinon
    si i < n-1 alors recherche_rec(e, T, i+1, n)
    sinon retourner(faux)
  fsi
  fsi
}
```

Appel initial : recherche_rec(e, T, 0, n)

Complexité : O(n)

Recherche d'un élément dans un tableau **trié** (algorithme récursif dichotomique)

```
booléen recherche_rec_dicho (t_elt e, t_elt T[], entier g, entier d)
{ entier i ← (g+d)/2;
  si g > d alors retourner(faux)
  sinon si T[i] = e alors retourner(vrai)
  sinon
    si T[i] < e alors recherche_rec_dicho(e, T, i+1, d)
    sinon recherche_rec_dicho(e, T, g, i-1)
  fsi
  fsi
}
```

Appel initial : recherche_rec_dicho(e, T, 0, n-1)

Complexité : O(log₂(n))

Autres modèles de données séquentiels

- Les listes, listes chaînées
- Les piles
- Les files

Modèle de données "Liste"

- Un des modèles de données les plus simples en algorithmique
- Permet de *regrouper séquentiellement des données de même type*, dont le nombre est fini mais non fixé à l'avance (c'est un modèle de données *dynamique*)
- On ne peut accéder directement qu'au premier élément (tête) d'une liste, l'accès au $n^{\text{ème}}$ élément se fait « à travers » les $n-1$ éléments le précédant (liste du reste des éléments = queue de liste)
- C'est donc un modèle de données *récuratif*

Exemples :

- la liste des jours de la semaine : (lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche)
- la liste des nombres premiers inférieurs à 20 en ordre croissant : (2, 3, 5, 7, 11, 13, 17, 19)

Modèle de données "Liste" Définition

- **Éléments** : un objet de type liste[elt_liste] est : soit vide (nil), soit composé d'un premier élément de type elt_liste et d'une seconde partie de type liste[elt_liste]
- **Propriétés** :
 - Tous les éléments de la liste sont de même type (elt_liste)
 - Accès direct au premier élément (tête) de liste
 - Accès séquentiel (récuratif) au reste de la liste
- **Opérations** :
 - Créer_liste : \rightarrow liste
 - Cons : elt_liste, liste \rightarrow liste
 - Est_vide? : liste \rightarrow booléen
 - Tête : liste \rightarrow elt_liste // ou primitive *car*
 - Queue : liste \rightarrow liste // ou primitive *cdr*

Opérations primitives sur les listes

(liste) créer_liste

ACT : retourne une liste vide.

(liste) cons(elt_liste e, mod liste l)

ACT : construit une liste dont la tête est l'élément e et la queue la liste l.

(bool) est_vide?(liste l)

ACT : retourne vrai si la liste l est vide (nil) et faux sinon.

(elt_liste) tête (liste l)

PRE : la liste l ne doit pas être vide.

ACT : retourne la tête (le premier élément) de la liste l.

(liste) queue(liste l)

PRE : la liste l ne doit pas être vide.

ACT : retourne la queue de la liste l, i.e. la liste l privée de son premier élément.

Les listes – propriétés, exemples

Les primitives cons, tête et queue vérifient les propriétés suivantes :

$\forall l \neq \text{nil}, l = \text{cons}(\text{tête}(l), \text{queue}(l))$
 $\forall x, l, \text{tête}(\text{cons}(x, l)) = x$
 $\forall x, l, \text{queue}(\text{cons}(x, l)) = l$

Exemple :

```
liste l1, l2;  
l1 ← créer_liste;  
si est_vide?(l1)  
  l1 ← cons(2, l1);  
sinon  
  écrire(tête(l1));  
l2 ← queue(l1);
```

ajouts et suppressions
toujours en tête de liste

recherche et parcours
en coût linéaire

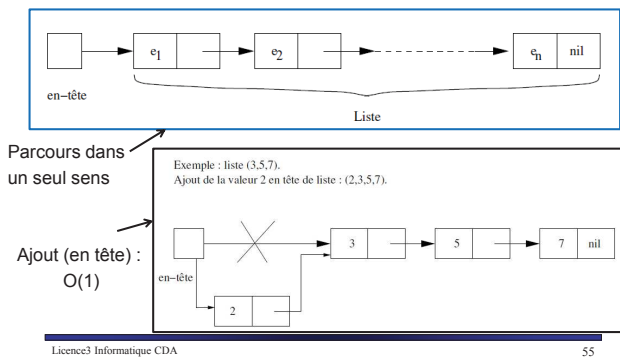
Exemple :

```
liste l ← créer_liste;  
l ← cons(dimanche, l);  
l ← cons(samedi, l);  
l ← cons(vendredi, l);  
l ← cons(jeudi, l);  
l ← cons(mercredi, l);  
l ← cons(mardi, l);  
l ← cons(lundi, l);
```

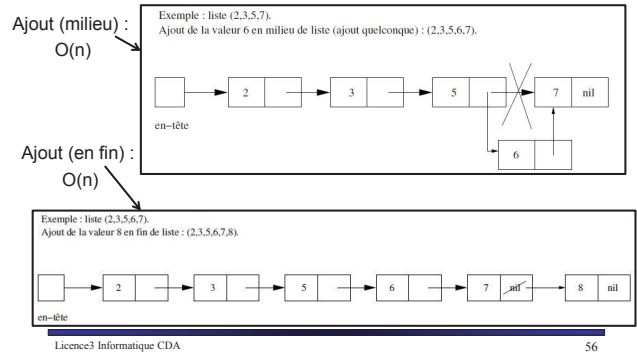
Implantation du modèle de données Liste

- Implantation par chaînage (listes chaînées) :
 - Simple chaînage avant
 - Simple chaînage arrière
 - Double chaînage
- Implantation par tableau

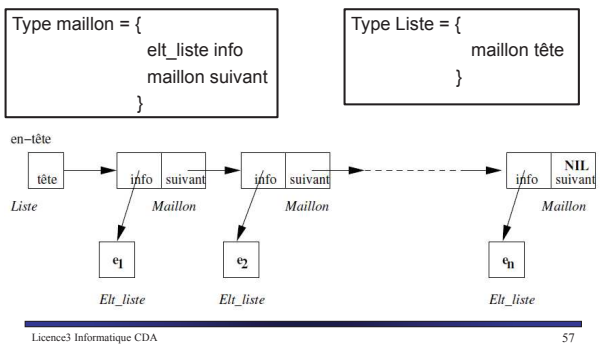
Représentation des listes en chaînage avant



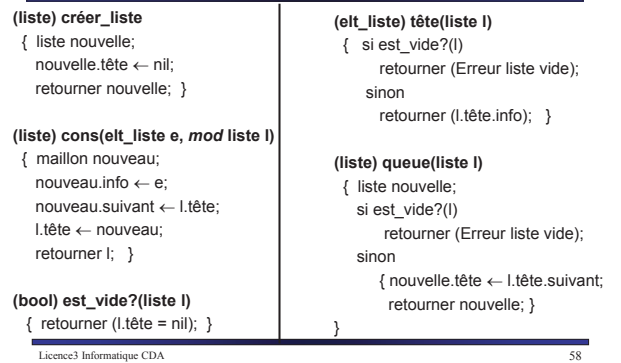
Représentation des listes en chaînage avant



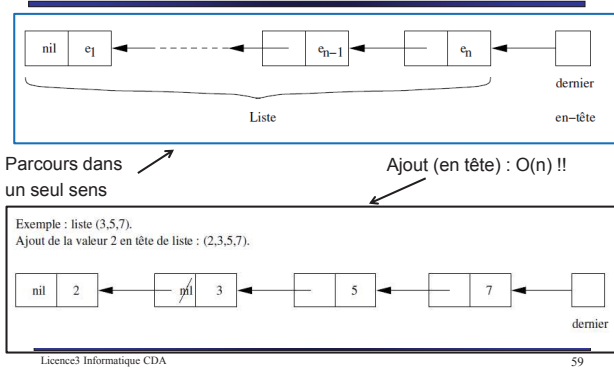
Implantation des listes en chaînage avant



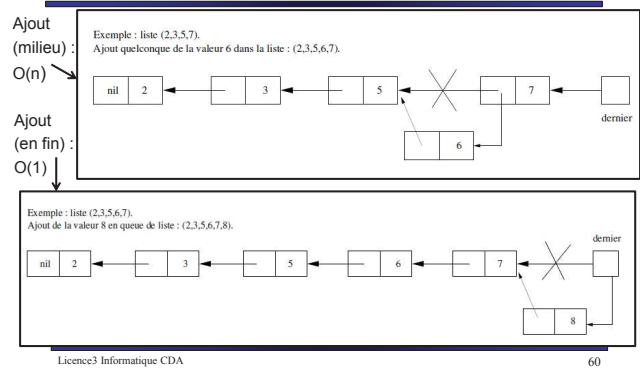
Implantation des listes en chaînage avant



Représentation des listes en chaînage arrière



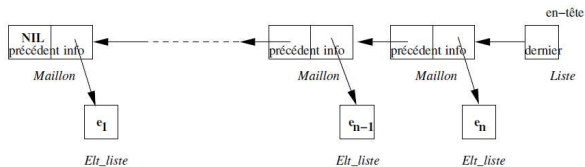
Représentation des listes en chaînage arrière



Implantation des listes en chaînage arrière

```
Type maillon = {
    elt_liste info
    maillon précédent
}
```

```
Type Liste = {
    maillon dernier
}
```



Licence3 Informatique CDA

61

Implantation des listes en chaînage arrière

(liste) créer_liste

```
{ liste nouvelle;
  nouvelle.dernier ← nil;
  retourner nouvelle; }
```

(bool) est_vide?(liste l)

```
{ retourner (l.dernier = nil); }
```

(liste) cons(elt_liste e, mod liste l)

```
{ maillon nouveau, m;
  nouveau.info ← e;
  nouveau.précédent ← nil;
  si (l.dernier = nil)
    l.dernier ← nouveau;
  sinon { m ← l.dernier;
    tantque (m.précédent ≠ nil)
      { m ← m.précédent; }
    m.précédent ← nouveau; }
  retourner l; }
```

(elt_liste) tête(liste l)

```
{ maillon m;
  si est_vide?(l)
    retourner (Erreur liste vide);
  sinon
    { m ← l.dernier;
      tantque (m.précédent ≠ nil)
        { m ← m.précédent; }
      retourner m.info; } }
```

Licence3 Informatique CDA

62

Implantation des listes en chaînage arrière

(liste) queue(liste l)

```
{ liste nouvelle;
  maillon m, n;
  si est_vide?(l)
    retourner (Erreur liste vide);
  sinon
    { nouvelle ← copie(l);
      m ← nouvelle.dernier;
      n ← nil;
      tantque (m.précédent ≠ nil)
        { n ← m;
          m ← m.précédent; }
      si (n ≠ nil)
        { n.précédent ← nil; } ..... }
```

```
..... sinon
  { nouvelle.dernier ← nil; }
  retourner nouvelle; }
```

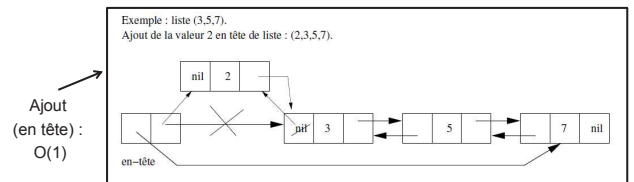
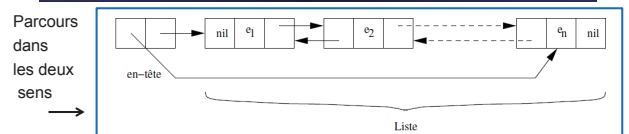
(liste) copie(liste l)

```
{ liste nouv;
  maillon m;
  nouv ← créer_liste;
  m ← l.dernier;
  tantque (m ≠ nil)
    { nouv ← cons(m.info, nouv);
      m ← m.précédent; }
  retourner nouv; }
```

Licence3 Informatique CDA

63

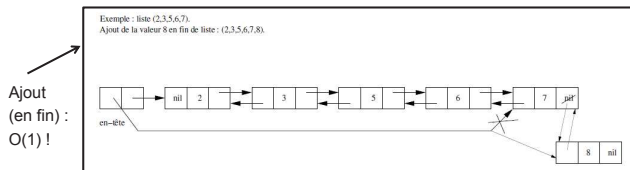
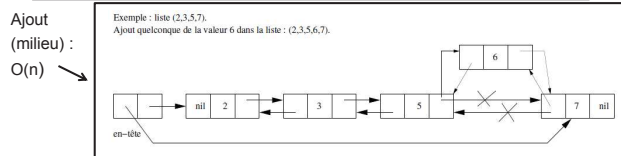
Représentation des listes en double chaînage



Licence3 Informatique CDA

64

Représentation des listes en double chaînage



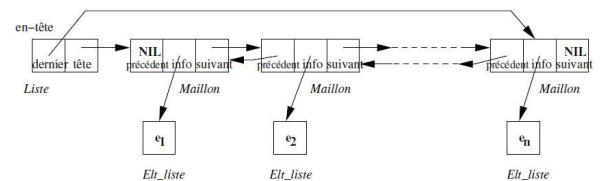
Licence3 Informatique CDA

65

Implantation des listes en double chaînage

```
Type maillon = {
    elt_liste info
    maillon précédent
    maillon suivant
}
```

```
Type Liste = {
    maillon tête
    maillon dernier
}
```



Licence3 Informatique CDA

66

Implantation des listes en double chaînage

```
(liste) créer_liste
{ ..... }

(elt_liste) tête(liste l)
{ ..... }

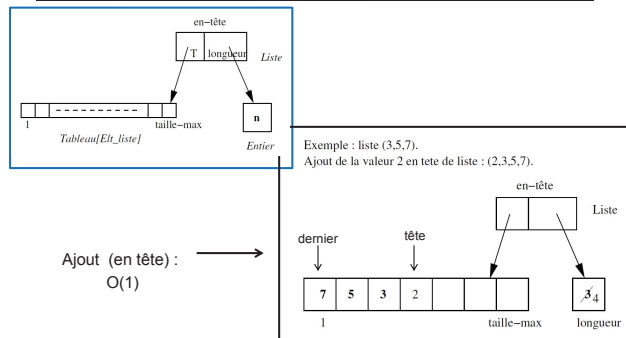
(bool) est_vide?(liste l)
{ ..... }

(liste) cons(elt_liste e, mod liste l)
{ ..... }

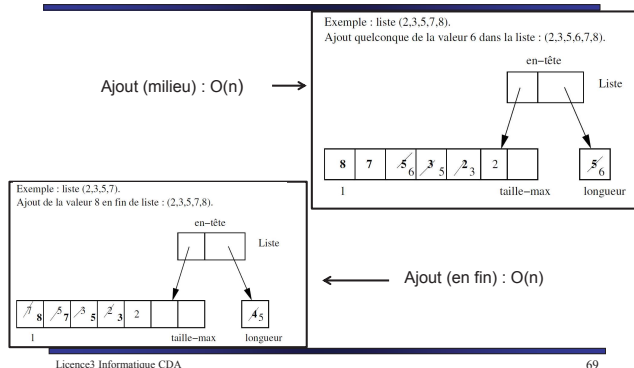
(liste) queue(liste l)
{ ..... }
```

Voir correction en TD

Représentation des listes par tableau



Représentation des listes par tableau



Implantation des listes par tableau

```
Type Liste = {
  tableau[elt_liste][1..taille_max] T
  entier longueur
}

(liste) créer_liste
{ liste nouvelle;
  tableau[elt_liste][1..taille_max] Telt;
  nouvelle.T ← Telt;
  nouvelle.longueur ← 0;
  retourner nouvelle; }

(bool) est_vide?(liste l)
{ retourner (l.longueur = 0); }

(elt_liste) tête(liste l)
{ si est_vide?(l)
  retourner (Erreur liste vide);
  sinon
  retourner (l.T[l.longueur]); }

(liste) cons(elt_liste e, mod liste l)
{ si (l.longueur = taille_max)
  retourner (Erreur liste pleine);
  sinon
  { l.longueur ← l.longueur + 1;
    l.T[l.longueur] ← e;
    retourner l; } }
```

Implantation des listes par tableau

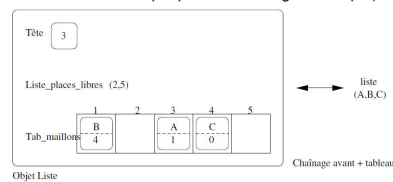
```
(liste) queue(liste l)
{ liste nouvelle;
  si est_vide?(l)
  retourner (Erreur liste vide);
  sinon
  { nouvelle ← créer_liste;
    nouvelle.longueur ← l.longueur - 1;
    pour i = 1 à nouvelle.longueur
    { nouvelle.T[i] ← l.T[i]; }
  retourner nouvelle; }
```

L'implantation par tableau limite l'aspect dynamique des listes représentées en bornant leur longueur maximum.

De plus, certaines opérations sur les listes deviennent très coûteuses (ajouts et suppressions d'éléments par exemple).

Implantation des listes par tableau (2)

Il existe une implantation intermédiaire consistant à implanter une liste par une liste chaînée (simplement ou doublement), elle-même implantée par un tableau. Les liens correspondent ainsi à des index dans le tableau contenant les éléments de la liste, et on gère en plus une liste des index des cases inutilisées du tableau. Cette technique permet de se débarrasser de la contrainte du stockage séquentiel des éléments de la liste, et ainsi de rendre les opérations d'ajout et suppression moins coûteuses (on retrouve le même coût que pour le chaînage classique).

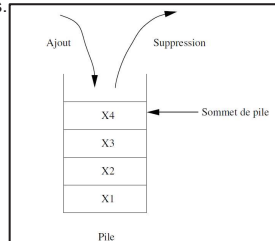


Modèle de données "Pile"

Une **pile** (LIFO, *Last in First Out*) est une liste particulière dans laquelle on ajoute (empile) et supprime (dépile) toujours en tête. Ces opérations sont les seules autorisées.

Dans ce contexte, la tête de liste est appelée **sommet de pile**.

Principe des piles :



Modèle de données "Pile" Définition

- **Éléments** : un objet de type `pile[elt_pile]` est un objet de type `liste[elt_pile]` pour lequel la tête est appelée **sommet** et où seules les primitives ci-après sont autorisées.
- **Propriétés** :
 - Tous les éléments de la pile sont de même type (`elt_pile`)
 - Accès uniquement au premier élément (**sommet**) de pile
- **Opérations** :
 - Créer_pile : `→ pile`
 - Pile_vide? : `pile → booléen`
 - Sommet_pile : `pile → elt_pile`
 - Empiler : `pile, elt_pile → pile`
 - Dépiler : `pile → pile`

Opérations primitives sur les piles

(pile) créer_pile

ACT : retourne une pile vide.

(bool) pile_vide?(pile p)

ACT : retourne vrai si la pile p est vide et faux sinon.

(elt_pile) sommet_pile (pile p)

PRE : la pile p ne doit pas être vide.

ACT : retourne la valeur du sommet de la pile p.

(pile) empiler (pile p, elt_pile e)

ACT : ajoute l'élément e au sommet de la pile p et retourne la pile modifiée.

(pile) dépiler (pile p)

PRE : la pile p ne doit pas être vide.

ACT : supprime l'élément au sommet de la pile p et retourne la pile modifiée.

Implantation du modèle de données Pile

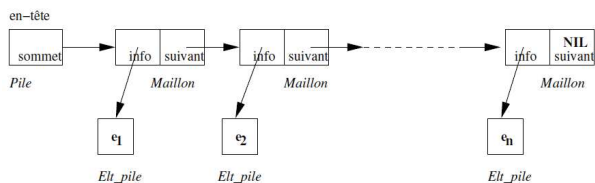
On peut implanter les piles :

- en utilisant le TAD Liste, par spécialisation. C'est élégant d'un point de vue conceptuel mais pas très efficace.
- par des tableaux. C'est efficace et très simple à mettre en oeuvre. L'inconvénient est de limiter l'aspect dynamique des piles.
- en utilisant le *simple chaînage avant* (le double chaînage est inutile car les ajouts/suppressions/consultations ne se font qu'en sommet de pile, et il n'existe pas de primitives de parcours de piles). Efficace et dynamique, c'est la *solution la mieux adaptée pour implanter les piles*.

Implantation des piles en chaînage avant

```
Type maillon = {
    elt_pile info
    maillon suivant
}
```

```
Type Pile = {
    maillon sommet
}
```



Implantation des piles en chaînage avant

(pile) créer_pile

```
{ pile nouvelle;
  nouvelle.sommet ← nil;
  retourner nouvelle; }
```

(bool) pile_vide?(pile p)

```
{ retourner (p.sommet = nil); }
```

(elt_pile) sommet_pile(pile p)

```
{ si pile_vide?(p)
  retourner (Erreur pile vide);
  sinon
  retourner (p.sommet.info); }
```

(pile) empiler(mod pile p, elt_pile e)

```
{ maillon nouveau;
  nouveau.info ← e;
  nouveau.suivant ← p.sommet;
  p.sommet ← nouveau;
  retourner p; }
```

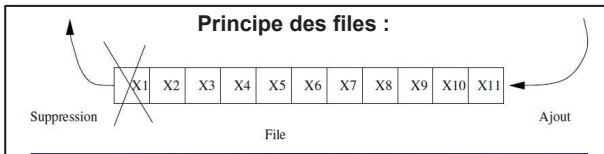
(pile) dépiler(mod pile p)

```
{ si pile_vide?(p)
  retourner (Erreur pile vide);
  sinon
  { p.sommet ← p.sommet.suivant;
    retourner p; }
```

Modèle de données "File"

Une **file** (FIFO, *First in First Out*) est une liste particulière dans laquelle on ajoute (enfile) toujours en fin et on supprime (défile) toujours en tête. Ces opérations sont les seules autorisées.

Dans ce contexte, on distingue et on a un accès direct aux éléments **tête de file** et **dernier de file**.



Licence3 Informatique CDA

79

Modèle de données "File" Définition

- **Éléments** : un objet de type file[elt_file] est un objet de type liste[elt_file] dans lequel on a un accès direct aux éléments tête et dernier, et pour lequel seules les opérations d'ajout en fin et de suppression en tête sont autorisées.
- **Propriétés** :
 - Tous les éléments de la file sont de même type (elt_file)
 - Accès uniquement au premier (tête) et dernier éléments de file
- **Opérations** :
 - Créer_file : \rightarrow file
 - File_vide? : file \rightarrow booléen
 - Enfiler : file, elt_file \rightarrow file
 - Défiler : file \rightarrow elt_file

Licence3 Informatique CDA

80

Opérations primitives sur les files

(file) créer_file

ACT : retourne une file vide.

(bool) file_vide?(file f)

ACT : retourne vrai si la file f est vide et faux sinon.

(file) enfiler (file f, elt_file e)

ACT : ajoute l'élément e à la fin de la file f et retourne la file modifiée.

(elt_file) défiler (file f)

PRE : la file f ne doit pas être vide.

ACT : supprime et retourne l'élément en tête de la file f, la file f est modifiée.

Licence3 Informatique CDA

81

Implantation du modèle de données File

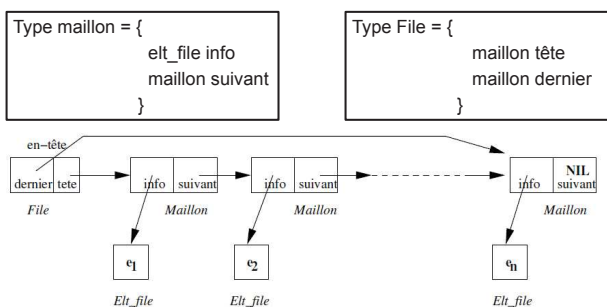
On peut implanter les files :

- en utilisant le TAD Liste, par spécialisation. C'est élégant d'un point de vue conceptuel mais pas très efficace.
- par des tableaux (avec une gestion circulaire). C'est efficace et très simple à mettre en oeuvre. L'inconvénient est de limiter l'aspect dynamique des files.
- en utilisant le *simple chaînage avant* (le double chaînage est inutile car les ajouts/suppressions ne se font qu'en début/fin de file, et il n'existe pas de primitives de parcours de files). Efficace et dynamique, c'est la *solution la mieux adaptée pour implanter les files*.

Licence3 Informatique CDA

82

Implantation des files en chaînage avant



Licence3 Informatique CDA

83

Implantation des files en chaînage avant

(file) créer_file

```
{ file nouvelle;
  nouvelle.tête ← nil;
  nouvelle.dernier ← nil;
  retourner nouvelle; }
```

(file) enfiler(mod file f, elt_file e)

```
{ maillon nouveau;
  nouveau.info ← e;
  nouveau.suivant ← nil;
  si file_vide?(f) f.tête ← nouveau;
  sinon f.dernier.suivant ← nouveau;
  f.dernier ← nouveau;
  retourner f; }
```

(bool) file_vide?(file f)

```
{ retourner (f.tête = nil) et
  (f.dernier = nil); }
```

(elt_file) défiler(mod file f)

```
{ elt_file e;
  si file_vide?(f)
    retourner (Erreur file vide);
  sinon
    { e ← f.tête.info;
      f.tête ← f.tête.suivant;
      si (f.tête = nil) f.dernier ← nil;
      retourner e; } }
```

Licence3 Informatique CDA

84

Les algorithmes de tri

- Les algorithmes de tri interne
 - toutes les données à trier sont présentes en mémoire centrale
- Les algorithmes de tri externe
 - les données à trier sont situées en mémoire secondaire (par exemple le disque), car elles sont trop volumineuses pour tenir en mémoire centrale
 - coût de l'accès aux données très pénalisant
 - le cas du tri fusion
- Complexité (pire des cas) :
 - en nombre de transferts d'éléments à trier
 - en nombre de comparaisons d'éléments à trier

Les algorithmes de tri interne

- Tris élémentaires
 - Tris par sélection
 - Tri à bulles (et tri shaker)
 - Tri par sélection/échange
 - Tris par insertion
 - Tri par insertion séquentielle
 - Tri par insertion dichotomique
 - Tri shell
- Tris par sélection plus sophistiqués
 - Tri rapide (Quick sort ou tri de Hoare ou tri par partitionnement)
 - Tri par tas (Heap sort)

Les tris par sélection

But : trier un tableau dans l'ordre croissant.

Principe des algorithmes de tri par sélection :

- sélectionner un élément (souvent le plus petit),
- le mettre à sa place définitive,
- puis trier le reste des éléments.

Le tri à bulles

Principe : On fait remonter les plus petits éléments vers le haut (début) du tableau, comme des bulles.

Méthode : On parcourt le tableau en partant de la fin (le bas), en effectuant un échange à chaque fois que l'on rencontre deux éléments successifs qui ne sont pas dans le bon ordre. On réitère le parcours jusqu'à ce que tous les éléments soient triés.

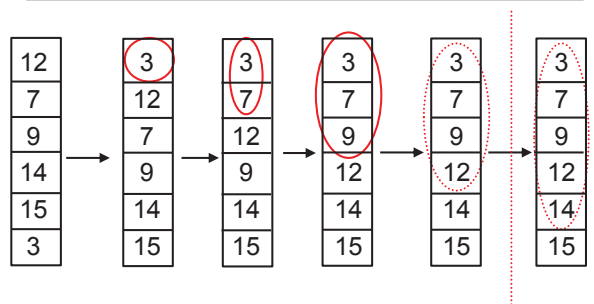
Ainsi, à la fin du premier parcours, le premier élément du tableau est à sa place. À la fin du second parcours, le second élément du tableau est à sa place. Etc...

L'algorithme itératif du tri à bulles

```
TriBulles (t_elt T[n])
{ entier i, j;
  t_elt x;
  pour i = 0 à n-2 pas 1
  { pour j=n-1 à i+1 pas -1
    { si T[j] < T[j-1] alors
      { x ← T[j];
        T[j] ← T[j-1];
        T[j-1] ← x; }
    }
  }
}
```

Complexité : $O(n^2)$
en nombre de transferts
et de comparaisons

Tri à bulles : exemple



L'algorithme récursif du tri à bulles

```
TriBulles_rec(t_elt T[], entier g, entier d)
{ entier j;
  si g<d alors
    { pour j=d à g+1 pas -1
      {
        si T[j]<T[j-1] alors permuter(T[j],T[j-1])
        fsi
      };
      TriBulles_rec(T,g+1,d)
    }
  fsi
}
```

Appel avec : TriBulles_rec(T,0,n-1)

L'algorithme récursif du tri à bulles optimisé

```
TriBulles2_rec(t_elt T[], entier g, entier d)
{ entier j;
  booléen trié;
  si g<d alors
    { trié ← vrai;
      pour j = d à g+1 pas -1
        { si T[j] < T[j-1] alors { permuter(T[j],T[j-1]);
          trié ← faux; }
        fsi
      }
      si non(trié) alors TriBulles2_rec(T,g+1,d) fsi
    }
  fsi
}
```

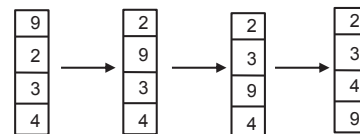
L'algorithme itératif du tri à bulles optimisé

```
TriBulles2 (t_elt T[n])
{ entier i ← 0, j;
  booléen b ← faux;
  tantque ((i<n-1)et(b=faux)) faire
    { b ← vrai;
      pour j=n-1 à i+1 pas -1
        { si T[j] < T[j-1] alors
          { permuter(T[j],T[j-1]);
            b ← faux; }
          fsi; }
      i ← i+1;
    }
}
```

Le tri shaker

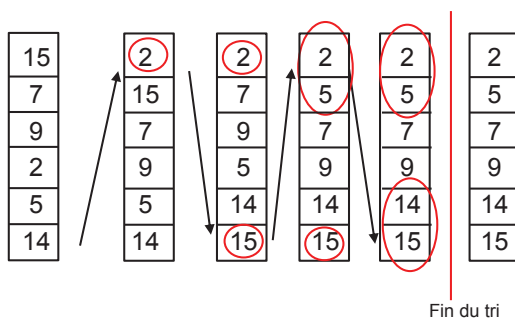
C'est une autre version optimisée du tri à bulles (pénalisé par la présence d'une grande valeur en début de tableau).

Cas typique très coûteux par le tri à bulles simple :



Principe du tri shaker : On alterne les parcours du tableau de bas en haut et de haut en bas.

Tri shaker : exemple



L'algorithme itératif du tri shaker

```
TriShaker (t_elt T[n])
{ entier i ← 0, j;
  booléen b ← faux;
  tantque ((i<n-1)et(b=faux)) faire
    { b ← vrai;
      si (i mod 2 = 0) alors
        { pour j=(n-1-(i/2)) à (i/2)+1 pas -1
          { si T[j] < T[j-1] alors
            { permuter(T[j],T[j-1]); b ← faux; } fsi
          }
        sinon
          { pour j=i/2+1 à (n-1-i/2)-1 pas 1
            { si T[j] > T[j+1] alors
              { permuter(T[j],T[j+1]); b ← faux; } fsi
            }
          }
      fsi;
      i ← i + 1;
    }
}
```

Complexité : $O(n^2)$
en nombre de transferts
et de comparaisons

L'algorithme récursif du tri shaker

```
TriShaker_rec(t_elt T[], entier g, entier d)
{ entier j;
  booléen trié;
  si g<d alors
  { trié ← vrai;
    si (g mod 2) = 0 alors
    { pour j=d à g+1 pas -1
      { si T[j] < T[j-1] alors
        { permuter(T[j],T[j-1]); trié ← faux;} fsi}
    si non(trié) alors TriShaker_rec(T,g+1,d) fsi}
  sinon { pour j=g+1 à d pas 1
    { si T[j] < T[j-1] alors
      { permuter(T[j],T[j-1]); trié ← faux;} fsi}
    si non(trié) alors TriShaker_rec(T,g,d-1) fsi}
  fsi } fsi }
```

Le tri par sélection/échange

Méthode : On recherche l'élément minimum du tableau et on l'échange avec le premier élément du tableau. Ensuite, on recherche l'élément minimum du reste du tableau et on l'échange avec le second élément. Etc...

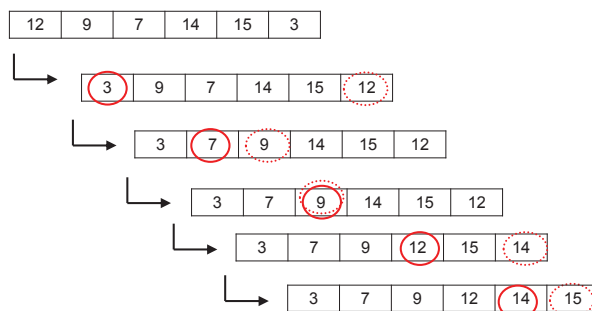
Cette méthode vise à réduire le nombre de transferts par rapport au tri à bulles.

L'algorithme itératif du tri par sélection/échange

```
TriSelectEchange (t_elt T[n])
{ entier i, k, min;
  pour i=0 à n-2 pas 1
  { min ← i;
    pour k=i+1 à n-1 pas 1
    { si T[k] < T[min] alors min ← k fsi}
    permuter(T[min],T[i]);
  }
}
```

Complexité :
 $O(n)$ en nombre de transferts,
 $O(n^2)$ en nombre de comparaisons

Tri par sélection/échange : exemple



L'algorithme récursif du tri par sélection/échange

```
TriSelectEchange_rec(t_elt T[], entier g, entier d)
{ entier min, k;
  si g<d alors
  { min ← g;
    pour k=g+1 à d pas 1
    { si T[k] < T[min] alors min ← k fsi }
    permuter(T[min],T[g]);
    TriSelectEchange_rec(T,g+1,d)
  }
}
```

Appel avec : TriSelectEchange_rec(T,0,n-1)

Les tris par insertion

Principe des algorithmes de tri par insertion :

- trier le début du tableau,
 - puis y insérer les éléments non encore triés.
- Ainsi, on trie successivement les premiers éléments du tableau.
- À la ième étape, on insère le (i+1)ème élément à sa place parmi les i éléments précédents qui sont déjà triés entre eux.

- C'est la méthode du joueur de cartes.
- Il existe *plusieurs types de tri par insertion* selon la méthode utilisée pour rechercher la position d'insertion : *séquentielle* ou *dichotomique*.

L'algorithme itératif du tri par insertion séquentielle

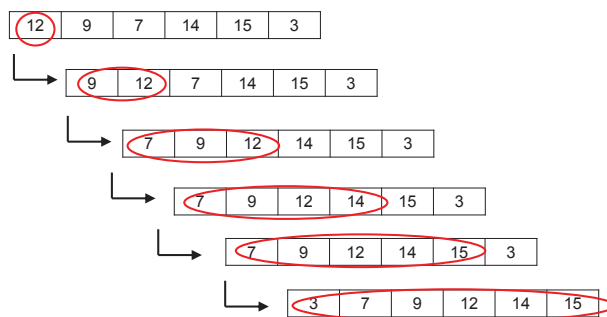
```

TriInsertSeq (t_elt T[n])
{ entier i, k;
  t_elt x;
  pour i=1 à n-1 pas 1
  { si T[i-1] > T[i] alors
    { k ← i-1;
      x ← T[i];
      tantque ((k ≥ 0) et (T[k] > x)) faire
        { T[k+1] ← T[k];
          k ← k-1; };
      T[k+1] ← x;
    }
  }
}

```

Complexité : $O(n^2)$
en nombre de transferts
et de comparaisons

Tri par insertion séquentielle : exemple



L'algorithme récursif du tri par insertion séquentielle

```

TriInsertSeq_rec(t_elt T[], entier i)
{ entier k;
  t_elt x;
  si i > 0 alors
  { TriInsertSeq_rec(T, i-1);
    si T[i-1] > T[i] alors
    { k ← i-1;
      x ← T[i];
      tantque ((k ≥ 0) et (T[k] > x)) faire
        { T[k+1] ← T[k]; k ← k-1; };
      T[k+1] ← x;
    }
  }
}

```

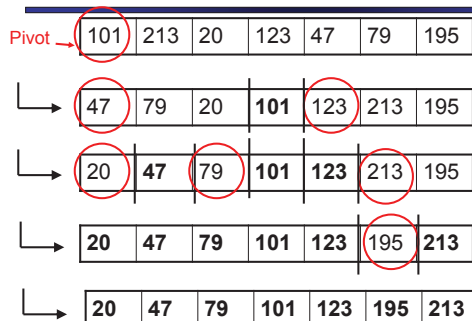
Appel avec : TriInsertSeq_rec(T, n-1)

Le tri rapide (Quick Sort)

Méthode : C'est un tri par sélection généralisé (on met un élément à sa place définitive, mais qui n'est pas forcément la première), par dichotomie, qui consiste à :

- choisir un élément **pivot**
- **partager le tableau à trier en deux sous-tableaux** T1 et T2 tels que les éléments de T1 soient inférieurs ou égaux au pivot, et les éléments de T2 soient supérieurs au pivot
- **insérer le pivot à sa place définitive** en le plaçant entre les deux sous-tableaux T1 et T2
- **trier les deux sous-tableaux** T1 et T2 selon la même méthode (on recommence jusqu'à ce que la taille du (sous-)tableau traité soit égale à 1)

Le tri rapide : exemple



Algorithme récursif du tri rapide

```

TriRapide(t_elt T[], entier g, entier d)
{ entier k;
  si g < d alors
  { k ← placer(T, g, d);
    TriRapide(T, g, k-1);
    TriRapide(T, k+1, d);
  }
}

```

Appel avec : TriRapide(T, 0, n-1)

Tri rapide : fonction *placer*

```
entier placer(t_elt T[], entier g, entier d)
{
  entier i ← g+1;
  k ← d;
  tantque i ≤ k faire
  {
    tantque T[k] > T[g] faire
    {
      k ← k-1;
    }
    tantque (i ≤ d et T[i] ≤ T[g]) faire
    {
      i ← i+1;
    }
    si i < k alors { permuter(T[i], T[k]);
                  i ← i+1;
                  k ← k-1; }
  }
  permuter(T[g], T[k]);
  retourner(k);
}
```

Complexité : $O(n)$
en nombre de transferts
et de comparaisons

Complexité de l'algorithme de tri rapide

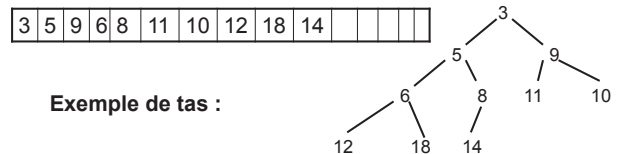
- **Complexité de "tri rapide" en nombre d'appels récursifs** :
les appels à TriRapide représentent un arbre binaire :
 - Si l'arbre est équilibré, le nombre d'appels récursifs est de l'ordre de $\log_2(n)$
 - Si l'arbre est totalement déséquilibré, le nombre d'appels récursifs est de l'ordre de n .
 - D'où l'importance du choix du pivot pour tenter d'obtenir à chaque fois deux sous-tableaux de taille équivalente. Le choix de ce pivot doit cependant rester simple (valeur médiane...).
- **Complexité globale de "tri rapide" en nombre de comparaisons et de transferts** : $O(n \log_2(n))$ si l'arbre est équilibré et $O(n^2)$ au pire.

Le tri par tas (Heap Sort)

- Méthode basée sur la notion de **tas** où on garde, d'une étape à l'autre, le résultat des comparaisons effectuées.
- **Définition : tas (heap)**
C'est un arbre (binaire) parfait *partiellement ordonné* représenté par un tableau $T[1..max]$ tel que :
 - $T[1]$ contient la racine de l'arbre
 - $T[i/2]$ est le père de $T[i]$, $\forall i > 1$
 - $T[2*i]$ et $T[2*i + 1]$ sont les deux fils, s'ils existent, de $T[i]$, $\forall i \geq 1$
 - Si p est la taille de l'arbre ($\leq max$) et si $p = 2*i$, $T[i]$ n'a qu'un seul fils $T[p]$
 - Si $i > p/2$ alors $T[i]$ est une feuille.

Notion de tas

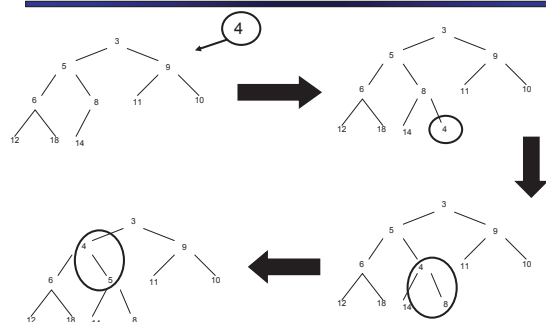
- **Définition : arbre (parfait) partiellement ordonné**
C'est un arbre (parfait) tel que l'élément contenu dans tout noeud est inférieur ou égal aux éléments contenus dans ses fils. Les éléments de l'arbre doivent donc faire partie d'un ensemble muni d'un ordre total.
→ L'élément minimum de l'arbre se trouve toujours à sa racine.



Opérations sur les tas : adjonction d'un nouvel élément à un tas

- **Méthode** :
 - ajout d'une feuille au dernier niveau de l'arbre,
 - on met le nouvel élément dans cette feuille (attention, l'arbre n'est plus nécessairement partiellement ordonné),
 - on fait remonter la nouvelle valeur dans l'arbre (par échange avec son père) jusqu'à ce qu'elle soit à sa place définitive.
- **Complexité** : au pire en $O(\log_2(p))$ si p est la taille de l'arbre (du tas).

Adjonction à un tas : exemple



Adjonction d'un élément à un tas

```

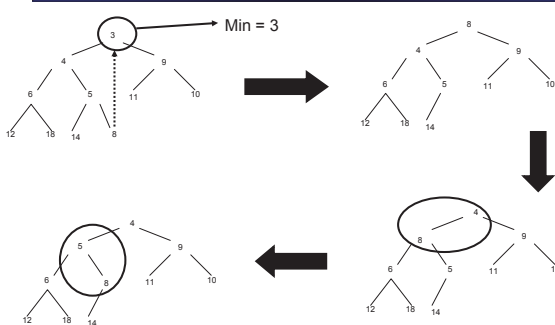
Adjonction(elt_tas T[1..max], entier p, elt_tas x)
{
  entier i;
  p ← p+1;           // p est un argument modifiable
  T[p] ← x;
  i ← p;
  tantque ((i>1) et (T[i] < T[i/2])) faire
    { permuter(T[i], T[i/2]);
      i ← i/2; }
}

```

Opérations sur les tas : suppression de l'élément minimum d'un tas (la racine)

- **Méthode :**
 - on extrait la valeur de la racine
 - on remplace la valeur de la racine par celle de la dernière feuille (attention, l'arbre n'est plus nécessairement partiellement ordonné),
 - on fait descendre cette valeur dans l'arbre (par échange avec son fils ayant la plus petite valeur) jusqu'à ce qu'elle soit à sa place définitive.
- **Complexité :** au pire en $O(\log_2(p))$ si p est la taille du tas.

Suppression dans un tas : exemple



Suppression de l'élément minimum d'un tas

```

elt_tas Suppression(elt_tas T[1..max], entier p)
{
  entier i, j;
  elt_tas min ← T[1];
  T[1] ← T[p];
  p ← p-1;           // p est un argument modifiable
  i ← 1;
  tantque i ≤ p/2 faire
    { si ((2*i = p) ou (T[2*i] < T[2*i + 1])) alors { j ← 2*i; }
      sinon { j ← 2*i + 1; }

      si T[i] > T[j] alors { permuter(T[i], T[j]);
                            i ← j; }
      sinon { i ← p; }

    }
  fin;
  retourner(min); }

```

Principe du tri par tas

- Le tri par tas est réalisé en deux étapes :
 - Construction d'un tas en y ajoutant successivement tous les éléments du tableau à trier.
 - Suppression du minimum du tas (puis réorganisation du tas) jusqu'à ce qu'il soit vide.
- Remarques :
 - Tout se passe dans un seul tableau.
 - À l'issue de l'algorithme, T est trié dans l'ordre décroissant ("à l'envers").
- Complexité : $O(n \log_2(n))$ au pire en nombre de transferts et en nombre de comparaisons.
 - Il est possible d'accélérer la construction du tas (en $O(n)$) mais le tri reste en $O(n \log_2(n))$.

Algorithme du tri par tas

```

TriParTas(t_elt T[1..n])
{
  entier p ← 0, min;
  tantque (p < n) faire
    { adjonction(T, p, T[p+1]); }           p augmente de 1
  tantque (p > 1) faire
    { min ← suppression(T, p);               p diminue de 1
      T[p+1] ← min;
    }
}

```