Plan du Cours

Bibliographie

- « Structures de données et algorithmes » - A.Aho, J. Hopcroft et J. Ullman - InterEditions - 1987.
- « Types de données et algorithmes » - M.C. Gaudel, M. Soria et C. Froidevaux - vol 1 - INRIA.
- « Eléments d'algorithmique » - D. Beauquier, J. Berstel et P. Chrétienne - Masson - 1992.
- « Programmation - cours et exercices » - G. Chaty et V. Vicard - Ellipse - 1992.

Chapitre 1 - Introduction

L'écriture d'un programme informatique pour la résolution d'un problème particulier comprend plusieurs étapes : formuler le problème, spécifier les données, construire une solution, mettre en oeuvre l'algorithme, tester le programme, le documenter et enfin évaluer la complexité de la solution.

Connaître le problème à traiter, c'est déjà l'avoir résolu à moitié. Au premier abord, la plupart des problèmes n'ont pas de spécification simple ni précise. En fait, certains problèmes ne sont pas formulables en termes acceptables pour une solution informatique (même si on soupçonne qu'ils peuvent être résolus par une machine).

1. Algorithme, type abstrait de données et structure de données

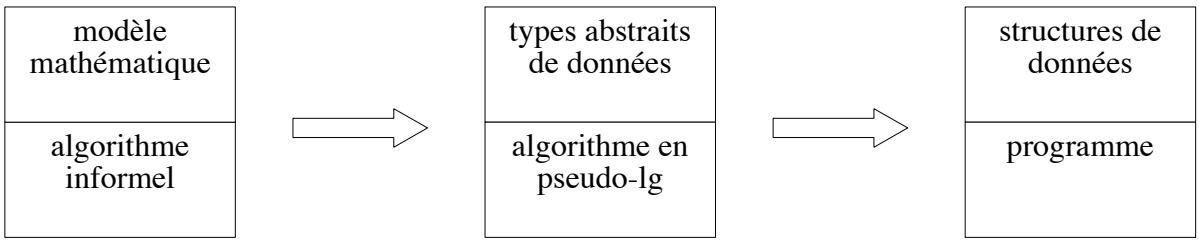
Construire une solution passe par l'écriture d'un algorithme, c'est-à-dire une série d'instructions dont chacune peut-être réalisée en un nombre fini d'étapes et en un temps fini. Il est impératif que, indépendamment des données initiales, un algorithme se termine après un nombre fini d'instructions.

→ Problèmes de calculabilité, de terminaison et de preuve de programme.

La conception d'un algorithme se fait toujours en appliquant une démarche descendante. On part de l'analyse exprimée en langage naturel jusqu'à l'algorithme qui s'exprime dans un langage plus restreint et plus précis que le langage naturel appelé pseudo-langage (destiné à une machine virtuelle capable de comprendre certaines instructions).

Au départ les données sont considérées de manière abstraite : on se donne une notation pour les décrire ainsi que l'ensemble des opérations qu'on peut leur appliquer et les propriétés de ces opérations → type abstrait de données.
La conception de l'algorithme se fait en utilisant les opérations du type abstrait indépendamment du choix du langage de programmation. On donne ensuite une représentation concrète des types et des opérations pour obtenir un programme exécutable écrit dans un langage spécifique. Les différentes représentations du type abstrait permettent d'obtenir différentes versions de l'algorithme si le type abstrait n'est pas un type du langage de programmation qu'on veut utiliser.

Un type abstrait de données est la description d'un ensemble organisé d'objets et des opérations de manipulation sur cet ensemble. Ces opérations comprennent les moyens d'accès aux éléments de l'ensemble et, lorsque l'objet est dynamique, les possibilités de le modifier.
Une structure de données est l'implémentation en machine d'un ensemble organisé d'objets avec la réalisation des opérations d'accès, de construction et de modification.



Définition d'un type abstrait de données

- *Signature* : syntaxe du type
  nom des opérations, type de leurs arguments
- *Propriétés des opérations* :
  axiomes donnant la sémantique des opérations (sous forme logique)

Signature

sorte : nom du type
utilise : types utilisés
opérations : listes des opérations avec le domaine des paramètres

Propriétés des opérations

liste de propriétés (axiomes) devant être satisfaites par toute implémentation du type abstrait
N'y a-t-il pas d'axiome contradictoire ? (pb de consistance)
A-t-on donné un nb suffisant d'axiomes pour décrire toutes les propriétés du type abstrait (pb de complétude)
Opération interne (ou constructeur) : si elle rend un résultat du type abstrait (modifie physiquement l'objet)
Observateur : si elle a au moins un argument du type abstrait et si elle rend un résultat d'un autre type (ne modifie pas physiquement l'objet)

Pour éviter les problèmes de complétude, il faut écrire des axiomes qui définissent le résultat de la composition des observateurs avec les opérations internes.

Importation de types

Lorsqu'un type de données est implémenté, on peut se servir de ses opérations comme opérations élémentaires et en particulier les utiliser dans la définition de types de données plus complexes.

Hiérarchie de types, avec en bas les types élémentaires (entier, réel, booléen, ...).

Exemple : coordonnées d'un point

Sorte : Coord
Utilise : Entier, Booléen
Opérations :
Créer(Entier, Entier) → Coord *[opération interne]*
X(Coord) → Entier
Y(Coord) → Entier
Eq(Coord, Coord) → Booléen
Axiomes :
X(Créer(x,y)) = x
Y(Créer(x,y)) = y
Eq(Créer(x1,y1), Créer(x2,y2)) = ((x1=x2) et (y1=y2))

2. Programme

C'est la suite des instructions résolvant le problème avec les déclarations des objets manipulés (constantes, types, variables, procédures et fonctions).

```
Programme bidule
Const constantes {Déclaration des constantes}
Type types {Déclaration des types}
Var variables {Déclarations des variables globales}
      Procédures et fonctions
Début
Programme principal
Fin
```

3. Instruction

Une instruction est un ordre donné à la machine.
L'exécution est l'action effectuée par la machine en réponse à cette instruction.
Il faut bien distinguer les objets (variables) manipulés par l'algorithme et les actions qui seront effectuées par la machine (correspondant aux instructions).

Affectation
L'affectation (symbolisée par ←) est l'action élémentaire dont l'effet est de donner une valeur à une variable, c'est-à-dire ranger une valeur à une place (en mémoire).
`X←4` mettre la valeur 4 dans la case identifiée par X
la valeur initiale de X est « écrasée » par la nouvelle valeur
La valeur correspond au contenu, la variable au contenant.

Exemple : échanger deux variables
Soit 2 variables entières X et Y ayant respectivement les valeurs x et y ; quelles sont les affectations qui donneront à X la valeur y et à Y la valeur x ?
Il faut utiliser une variable intermédiaire Z.

```
Z←X
X←Y
Y←Z
```

Lecture et écriture
Pour pouvoir effectuer un calcul sur une variable, la machine doit connaître la valeur de cette variable. Si cette valeur n'a pas été déterminée par des initialisations ou des calculs précédents, il faut que l'utilisateur lui fournisse ; notion de données.

- instruction `lire`

`lire(X)` mettre dans la case X, la valeur présente sur l'organe d'entrée de la machine

De la même façon, la machine doit pouvoir fournir un résultat.

- instruction `écrire`

`écrire(X)` mettre sur l'organe de sortie de la machine, le contenu de la case X

Alternatives
Les alternatives permettent à la machine de choisir les exécutions suivant les valeurs des données.

A. Si alors

```
Si cond
      Alors inst
FinSi
```

La suite d'instructions `inst` est exécutée si et seulement si la condition `cond` est vérifiée.

B. Si alors sinon

```
Si cond
      Alors inst1
      Sinon inst2
FinSi
```

La suite d'instructions `inst1` est exécutée si et seulement si la condition `cond` est vérifiée, `inst2` est exécutée sinon.

C. Selon

Cette instruction permet à la machine de faire un choix parmi n possibles, n étant plus grand que deux.

```
Selon exp
      val1 : inst1
      val2 : inst2
      ...
      valn : instn
FinSelon
```

Selon que la valeur de `exp` est `val1`, `val2` ou `valn`, `inst1`, `inst2` ou `instn` est exécutée.

Exemples : programme résolvant l'équation $ax^2 + bx + c = 0$ dans l'ensemble des réels

```
Programme équation
Var a, b, c : réel
Début
      lire(a)
      lire(b)
      lire(c)
      Si a=0
            Alors Si b=0
                  Alors Si c=0
                              Alors écrire('Tout réel est solution')
                              Sinon écrire('Pas de solution')
                        FinSi
                  Sinon écrire(–c/b)
            FinSi
      Sinon Si (b*b-4*a*c)>=0
                  Alors écrire('Il existe deux solutions : ',
                                    -b+sqrt((b*b-4*a*c)/(2*a)),' et ',
                                    -b-sqrt((b*b-4*a*c)/(2*a)))
                  Sinon écrire('Pas de solution')
            FinSi
      FinSi
Fin
```

Chapitre 2 – Les Itérations

Elles servent lorsqu'on doit effectuer le même traitement plusieurs fois sur un même objet ou plusieurs objets de même nature.

1. Répéter

```
Répéter
      inst
Jusqu'à cond
```

La suite d'instructions `inst` est exécutée au moins une fois et son exécution est répétée jusqu'à ce que la condition `cond` soit satisfaite.

Exemple : lecture d'un chiffre rentré par l'utilisateur

```
Répéter
      lire(c)
Jusqu'à (c0 and c≤9)
```

2. Tantque

```
TantQue cond Faire
      inst
FinTantQue
```

La suite d'instructions `inst` est exécutée tant que la condition `cond` est satisfaite. Si d'emblée `cond` n'est pas remplie, il n'y a pas d'action.

Exemple : calcul du PGCD
PGCD(x,0) = x
PGCD(x,y) = PGCD(y, x mod y) si y ≠ 0

```
lire(A)
lire(B)
X ← A
Y ← B
TantQue Y ≠ 0 Faire
      Z ← X
      X ← Y
      Y ← Z mod Y
FinTantQue
écrire(X)
```

3. Pour

Utilisé lorsqu'on sait combien de fois on doit répéter une suite d'instructions.

```
Pour var←valinit à valfin Incr op Faire
      inst
FinPour
```

Au départ, la variable `var` est initialisée à la valeur `valinit` La suite d'instructions `inst` est exécutée tant que la condition `var` est différente de `valfin`. A la fin de chaque exécution de `inst`, l'opération `op` (incrémentation ou décrémentation) est effectuée sur `var`.

Exemple : somme des n premiers entiers

```
lire(N)
S←0
Pour I←1 à N Incr +1 Faire
      S←S+I
FinPour
```

Chapitre 3 – Les Types scalaires

Le mot type est utilisé ici pour les variables « simples » rencontrées jusque là : le type d'une variable est l'ensemble des valeurs qu'elle peut prendre, cet ensemble étant muni des opérations qu'on peut effectuer sur ces valeurs.

1. Les types de base

Ce sont le type entier et le type réel. Ils sont dits scalaires car constitués d'objets scalaires (scalaris = échelon ; totalement ordonnés).

Le type entier
Il est muni des opérations suivantes :
+ - * div mod

Le type réel
Il est muni des opérations suivantes :
+ - * /

En mathématique, l'ensemble des entiers est inclus dans l'ensemble des réels. En programmation, les entiers et les réels ont des représentations différentes.

```
Var   i : entier
      b : réel
i ← 5
b ← i
```

L'affectation respecte l'inclusion de N dans R, mais s'accompagne d'un changement de représentation interne. Le contenu de b est 5, mais la représentation interne est de la forme $0{,}5*10^{1}$.
En revanche, l'affectation d'une variable réelle à une entière n'est pas possible directement même si cette variable contient une valeur entière.

```
b ← 3.0
i ← trunc(b) {partie entière}
```

2. Le type booléen

Une *proposition* a l'une des deux valeurs : vrai, faux.
Exemple : La proposition 4>10 a la valeur faux.

Une *fonction propositionnelle* ou prédicat renferme des variables et devient une proposition pour chaque valeur de ces variables.
Exemple : I>10, où I est une variable entière, est une fonction propositionnelle p de valeur p(I). p(I) est parfois considérée comme une variable logique (ou variable binaire) dont les valeurs sont V et F (pour vrai et faux)

Opérations sur les propositions

Dans l'ensemble des deux propositions V et F, on définit des opérations dites opérations logiques dont les plus importantes sont :

conjonction	∧
disjonction inclusive	∨
négation	¬
implication	⇒
équivalence	⇔
disjonction exclusive	w

p	q	p∧q	p∨q	p⇒q	p⇔q	pwq
F	F	F	F	V	V	F
F	V	F	V	V	F	V
V	F	F	V	F	F	V
V	V	V	V	V	V	F

Les opérateurs logiques sont appelés connecteurs.
Chacune de ces opérations peut s'exprimer à l'aide des trois connecteurs : ∧, ∨, ¬.

p⇒q = ¬p∨q
p⇔q = (¬p∨q) ∧ (¬q∨p)

¬(p∨q) = ¬p∧¬q
¬(p∧q) = ¬p∨¬q
pwq = ¬(p⇔q) = ¬ [(¬p∨q) ∧ (¬q∨p)] = ¬(¬p∨q) ∨ ¬(¬q∨p) = (p∧¬q) ∨ (q∧¬p)

Le type booléen

Il est constitué de deux éléments : vrai, faux notés V et F.
Il est muni des opérations : ∧, ∨, ¬
Il est totalement ordonné par la relation : F<V

3. Le type caractère

Le type caractère est l'ensemble des caractères d'imprimerie habituelle. Il est noté `car`.
Une variable de type caractère peut prendre toute valeur de ce type. Une constante de type `car` se distingue par son écriture 'O', 'A', 'o', 'a', '+'.
Un ordinateur ne pouvant manipuler que des éléments binaires, il est nécessaire de codifier les caractères, cad de faire correspondre à chacun d'eux une configuration binaire. Un des codes les plus utilisés est le code ASCII.
Les fonctions `ord` (pour obtenir le code ASCII du caractère donné), `chr` (pour obtenir le caractère du code ASCII donné), `prec` (caractère précédent) et `succ` (caractère suivant) sont définies.

4. Les autres types scalaires

Type défini par énumération

```
Type saison : (prin, été, aut, hiver)
```
On a la relation prin < été < aut < hiver

Type intervalle

`0 .. 10`	pour	{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
`'a' .. 'f'`	pour	{'a', 'b', 'c', 'd', 'e', 'f'}
`été .. hiv`	pour	{été, aut, hiver}

Pour ce dernier exemple, il faut évidemment que le type `saison` ait été défini.

5. Ordres de priorité

1 - `Non`
2 - Opérateurs multiplicatifs : `*` `/` `div` `mod` `et`
3 - Opérateurs additifs : `+` `-` `ou`
4 - Opérateurs relationnels : `=` `≠` `<` `≤` `>` `dans`

Chapitre 4 – Les Séquences

Les tableaux sont une structure de données permettant de manipuler des séquences (ou suites).

1. Définition d'une séquence

Etant donné
- un ensemble de valeurs E
- et un ensemble fini totalement ordonné I d'éléments appelés indices,

une séquence sur E est une application de I dans E

$s : I \rightarrow E$

A chaque i de I correspond un élément de E : $i \rightarrow s_i$
I est souvent N ou une partie de N, mais ce peut être un type scalaire différent de réel car un tel type a toutes les propriétés de I.
L'énumération des éléments de la séquence induit un ordre total appelé ordre induit par la séquence, qui n'est autre que l'ordre des rangs, à ne pas confondre avec l'ordre éventuel sur E.

Séquence triée

Etant donné un ensemble E totalement ordonné par la relation $\leq$, on dit qu'on a une séquence triée lorsque l'ordre induit par la séquence est compatible avec l'ordre sur E :
$\forall (i, j) \in I, i \leq j$ ssi $s_i \leq s_j$

Trier une séquence s, c'est trouver une séquence triée s' formée des mêmes éléments que s. Trier une séquence n'est possible que s'il existe une relation d'ordre sur E.

2. Type abstrait de données d'une séquence

Sorte : Seq
Utilise : Entier, Elément
Opérations :

CréerSeq(Entier, Entier) → Seq *[opération interne]*
ChangerIème(Seq, Entier, Elément) → Seq *[opération interne]*
BorneInf(Seq) → Entier
BorneSup(Seq) → Entier
Ième(Seq, Entier) → Elément

Axiomes :

BorneInf (CréerSeq(x,y)) = x
BorneInf (ChangerIème (s,n,v)) = BorneInf (s)
BorneSup (CréerSeq (x,y)) = y
BorneSup (ChangerIème (s,n,v)) = BorneSup (s)
Ième(CréerSeq (x,y),n) = Indéfini
Ième(ChangerIème (s,n,v),m) = **si** m < BorneInf (s) **ou** m > BorneSup (s)
alors Indefini
sinon si m=n **alors** v **sinon** Ieme(s,m)

Pour supprimer un élément, on met la valeur vide ε à la place de l'élément.

TAD d'une séquence triée SeqTriée :

Même signature et mêmes opérations.
Axiomes à ajouter :
∀ i, j ∈ [BorneInf(s), BorneSup(s)], i ≤ j ⇔ Ième(s,i) ≤ Ième(s,j)
ChangerIème(s,i,e) = erreur **si** e < Ième(s,i-1) et e > Ième(s,i+1)

TAD d'un séquence de coordonnées SeqCoord, dont on veut connaître le nombre effectif d'éléments :

Sorte : SeqCoord
Utilise : Entier, Seq, Coord
Opérations :
CréerSeq (Entier, Entier) → SeqCoord *[opération interne]*
ChangerIème(SeqCoord, Entier, Coord) → SeqCoord *[opération interne]*
BorneInf(SeqCoord) → Entier
BorneSup(SeqCoord) → Entier
Ième(SeqCoord, Entier) → Coord
Compte(SeqCoord) → Entier
Axiomes :
Compte(CréerSeq(i,j)) = 0 (toutes les cases sont à ε)
Compte(ChangerIème(s,i,c)) = **si** (Ième(s,i) = ε) et (c ≠ ε)
alors Compte(s) + 1
sinon si Ième(s,i) ≠ ε et c = ε
alors Compte(s) - 1
sinon Compte(s)
où ε symbolise la valeur vide

3. Implémentation d'une séquence

Un tableau permet l'implantation d'une séquence. Le tableau est le contenant de la séquence, comme une variable élémentaire est le contenant d'une valeur scalaire.
Un tableau à une dimension (appelé aussi vecteur) est une structure de données formées de cellules contiguës et d'accès direct. L'accès direct signifie que l'on peut obtenir le contenu d'une cellule sans qu'il soit nécessaire de connaître le contenu des cellules précédentes du tableau.

Pour définir un tableau, il faut préciser l'identificateur du tableau, le type de I et le type de E.

```
Const max = 30
Type tab-entier : tableau [1..max] d'entier
Var t : tab-entier
```

CréerSeq(i,j) : fait par la déclaration du type et de la variable ;
par contre les initialisations restent à faire
BorneInf(t) : 1
BorneSup(t) : 30
Ième(t,i) : t[i]
ChangerIème(t,i,e) : t[i] ← e

Exemples :
Lecture d'une séquence
```
lire(n)
Pour i←1 à n inc +1 Faire
      lire(t[i])
FinPour
```

Ecriture d'une séquence
```
Pour i←1 à n inc +1 Faire
      écrire(t[i])
FinPour
```

4. Séquence doublement indicée

- un ensemble de valeurs E
- et deux ensembles finis I et J totalement ordonnés,

une séquence sur E est une application de I×J dans E

s : I×J → E

A chaque couple (i, j) dans lequel i est élément de I et j est élément de J, correspond un élément de E : (i, j) → $s_{i,j}$
Une séquence doublement indicée sert à modéliser ce qu'on appelle une table à double entrée. Elle est représentée souvent par une matrice.

Type abstrait de données d'une séquence doublement indicée

Sorte : SeqDouble
Utilise : Entier, Elément
Opérations :

CréerSeq(Entier, Entier, Entier, Entier) → SeqDouble
ChangerIème(SeqDouble, Entier, Entier, Elément) → SeqDouble
BorneInf(SeqDouble, Entier) → Entier
BorneSup(SeqDouble, Entier) → Entier
Ième(SeqDouble, Entier, Entier) → Elément

5. Implémentation d'une séquence doublement indicée par tableau à deux dimensions

Un tableau à deux dimensions est une structure de données permettant l'implantation d'une séquence doublement indicée. Pour définir un tel tableau, il faut préciser l'identificateur du tableau, le type de I et de J et le type de E.

```
Const maxi = 30
      maxj = 50
Type mat-entier : tableau [1..maxi, 1..maxj] d'entier
Var m : mat-entier
```

CréerSeq(i,j,k,l) : fait par la déclaration du type et de la variable ;
par contre les initialisations restent à faire
BorneInf(m, 1) : 1
BorneSup(m, 1) : 30
BorneInf(m, 2) : 1
BorneSup(m, 2) : 50
Ième(m,i,j) : m[i,j]
ChangerIème(m,i,j,e) : m[i,j] ← e

Exemples

Lecture d'une séquence doublement indicée

```
lire(n)
lire(m)
Pour i←1 à n inc +1 Faire
      Pour j←1 à m inc +1 Faire
      lire(m[i,j])
      FinPour
FinPour
```

Ecriture d'une séquence doublement indicée

```
Pour i←1 à n inc +1 Faire
      Pour j←1 à m inc +1 Faire
            écrire(m[i,j])
      FinPour
FinPour
```

Chapitre 5 – Les procédures et les fonctions

Une tâche est une action plus ou moins complexe qui s'exerce à un instant donné sur un ou plusieurs objets (variables). Les procédures et les fonctions permettent de réaliser des tâches.

1. Les procédures

Une procédure permet la réalisation de n'importe quelle tâche. C'est une nouvelle instruction crée à l'initiative du programmeur.
Puisqu'une procédure sert à réaliser une action, elle sera identifiée par un verbe à l'infinitif symbolisant l'action. Chaque donnée et chaque résultat attendu doivent être également identifiés.
Spécifier une procédure,

- c'est identifier
  - l'action réalisée par cette procédure,
  - les *paramètres formels (servant à écrire la procédure)*
    + l'ensemble des données nécessaires, ainsi que leur type,
    + l'ensemble des résultats produits, ainsi que leur type.
- c'est décrire l'état des données et des résultats avant et après exécution de la procédure.

```
Procédure nom-proc (E variables avec types ; E/S variables avec types ;
                    S variables avec types)
```

- E : paramètre de donnée, ne subit aucune altération au cours de l'exécution de la procédure (pas d'affectation de ce paramètre dans la procédure).
- S : paramètre résultat de l'action, sa valeur n'est pas significative avant l'exécution de la procédure (initialisation obligatoire de ce paramètre dans la procédure).
- E/S : paramètre de donnée-résultat, sert de donnée et de résultat.

Exemple : échange de deux variables.

```
Procédure échanger (E/S A, B : entier)
Var C : entier
Début
      C←A
      A←B
      B←C
Fin
```

Les procédures doivent être placées avant le début des instructions du programme et après la déclaration des variables.

Pour appeler une procédure, il suffit de donner son identificateur avec, entre parenthèses, la liste des variables du programme sur lesquelles l'action va s'exercer. Ces variables sont appelées paramètres effectifs.

Contraintes à suivre :

- même nombre de paramètres effectifs que de paramètres formels,
- les para mètres effectifs sont associés aux paramètres formels dans leur ordre,
- les para mètres effectifs doivent être du même type que les paramètres formels associés,

Exemple : appel de la procédure échanger

```
Var C, D, E : entier
........
C←2
D←3
échanger(C,D)
E←5
échanger (E, C)
```

2. Les fonctions

Même sens qu'en mathématique.
On utilise une fonction f en programmation pour obtenir la valeur f(x) de f pour une valeur x de la variable.
On crée une fonction chaque fois qu'on met en évidence au moment de l'analyse, une tache de calcul (au sens large) ne s'exprimant pas directement à l'aide des opérateurs prédéfinis.

- procédure : instruction
- fonction : expression

Dans une fonction, les paramètres formels sont par définition des paramètres d'entrée.

```
Fonction nom-fonc (paramètres entrées avec types) : type du param. résultat
```

Exemple : pour un tableau t, donne la position du minimum pos-min entre deux indices i et j.

```
Fonction pos-min (t : tableau ; i, j : entier) : entier
Var k, l : entier
Début
k←i
Pour l←i+1 à j inc +1 Faire
      Si t[l]<t[k]
            Alors k←l
      FinSi
FinPour
Retourner(k)
Fin
```

Visibilité des objets (constantes, types, variables, paramètres formels)

Pour un module (procédure ou fonction)

- ses *objets locaux* sont définis dans le module, ils ne peuvent pas être utilisés en dehors,
- ses *objets globaux* sont définis en dehors du module.

Des objets globaux ne peuvent pas porter le même nom. Les objets locaux à un module ne peuvent pas porter le même nom. Par contre des objets locaux à des modules différents peuvent porter le même nom. Enfin, un objet local à un module peut porter le même nom qu'un objet global. Dans le module, ce nom désigne alors l'objet local tandis qu'à l'extérieur du module, ce nom désigne l'objet global.

Chapitre 6 – Les chaînes de caractères

Une chaîne de caractères est un type abstrait défini comme une séquence de caractères.
La longueur d'une chaîne est le nombre de caractères qui la composent. La chaîne vide est une chaîne de longueur 0. Les chaînes sont des types scalaires sur lesquels est appliqué l'ordre lexicographique :
Soient A et B deux chaînes : on dit que $A \leq B$ ssi
soit $l(A) \leq l(B)$ et $\forall i\ 1 \leq i \leq l(A)\ Ai = Bi$
soit $\exists i\ 1 \leq i \leq \min(l(A), l(B))$ tq $Ai < Bi$ et $\forall j\ 1 \leq j < i\ Aj = Bj$
La relation d'ordre $Ai \leq Bi$ dans l'ensemble des lettres est celle de l'ordre alphabétique habituel.
L'ordre lexicographique étendu s'étend à l'ensemble des chaînes construites sur l'ensemble des caractères du code ASCII. Il suffit de prendre pour définition de la relation $Ai \leq Bi$
$Ai \leq Bi \Leftrightarrow ORD(Ai) \leq ORD(Bi)$

Le TAD chaîne

Sorte : Chaîne
Utilise : Entier, Car, Booléen
Opérations :

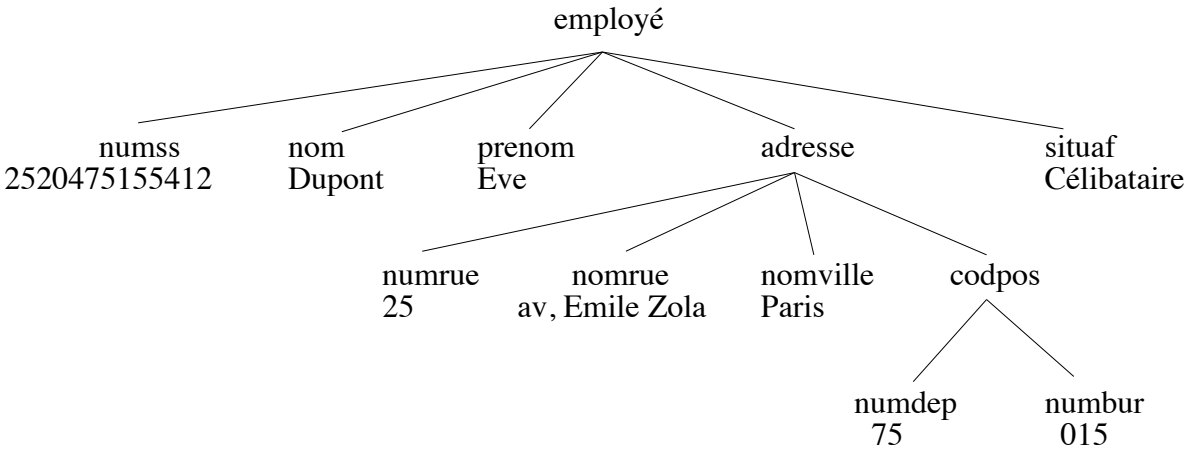
- CréerChaîne() → Chaîne
- ChangerIème(Chaîne, Entier, Car) → Chaîne
- Concaténer(Chaîne, Chaîne) → Chaîne
- Copier(Chaîne, Entier, Entier) → Chaîne
- Effacer(Chaîne, Entier, Entier) → Chaîne
- Insérer(Chaîne, Chaîne, Entier) → Chaîne
- Ième(Chaîne, Entier) → Car
- Egal(Chaîne, Chaîne) → Booléen
- Lg(Chaîne) → Entier
- Pos(Chaîne, Chaîne) → Entier

Exemples
s ← «Il était une fois une vilaine sorcière»
insère(efface(s, 22, 8), «gentille », 22)

Chapitre 7 – Les enregistrements

Un enregistrement est une structure de donnée hiérarchisée. On peut la voir comme une cellule composée d'éléments qui peuvent être de types différents.

Exemple
L'information « employé » peut être représentée par un enregistrement composé de 5 éléments : NUMSS, NOM, PRENOM, ADRESSE, SITUAF ; le 4ème ADRESSE se décomposant en NUMRUE, NOMRUE, NOMVILLE, CODPOS et ce dernier encore en NUMDEP et NUMBUR.



La variable enregistrement `employé` est représentée par une arborescence de racine employé. A chaque niveau de l'arborescence, on trouve les champs. Un champ correspond à un sommet distinct de la racine ; on l'identifie à l'aide des sommets du chemin unique qui lie la racine au champ. Les champs feuilles de l'arborescence sont dits champs élémentaires.

Un employé particulier est une valeur de la variable enregistrement `employé`. Chacune des informations placées aux feuilles est la valeur d'un champ élémentaire. L'ensemble de toutes ces valeurs constitue la valeur de la variable `employé`.

Un type enregistrement est décrit par les types des champs structurés hiérarchiquement.

```
Type   code-postal = Enregistrement
                         numdep, numbur : chaine de caract.
                     FinEnregistrement
       adresse-pos = Enregistrement
                         numrue, nomrue, nomville : chaine de caract
                         codpos : code-postal
                     FinEnregistrement
       employé = Enregistrement
                   numss, nom, prénom, situaf : chaine de caract
                   adresse : adresse-pos
                 FinEnregistrement

Var e : employe
```

Numéro du département où habite l'employé e `: e.adresse.codpos.numdep`

Les seules opérations globales possibles sur les enregistrements de mêmes types sont l'affectation et la comparaison (= ou ≠). Les opérations possibles sur les champs sont celles réalisables sur les variables du type de ces champs.

Exemple : tableau de points

```
Const max = 30
Type coord = Enregistrement
                 x,y : entier
             FinEnregistrement
     tab-coord = tableau[1..max] de coord
Var t : tab-coord
    i : entier
    p : coord
Début
p.x←0
p.y←0
Pour i←1 à max Incr +1 Faire
  lire(t[i].x,t[i].y)
  p.x ← p.x + t[i].x
  p.y ← p.y + t[i].y
FinPour
p.x ← p.x div max
p.y ← p.y div max
Fin
```

Chapitre 8 – Les fichiers

Un fichier est une structure de données formée d'une séquence de cellules en mémoire secondaire (MS). Il permet l'implantation permanente d'une suite d'éléments ; en tant que support de la suite, il est alors le contenant, la suite le contenu. On lui associe un nom qui le définit de manière unique.

L'utilisation d'un fichier se fait en 3 phases : l'ouverture du fichier, l'accès au fichier (lecture/écriture) et la fermeture du fichier.

L'accès est la manière d'accéder à un élément du fichier. Il est séquentiel si pour accéder à un élément, il faut visiter les éléments précédents. Il est direct si on peut accéder aux éléments dans un ordre quelconque.

Sur tout type de fichier, il est difficile d'insérer un enregistrement en milieu de fichier existant.

1. Les fichiers à accès séquentiel (FAS)

Les fichiers séquentiels sont des fichiers dont seul l'accès séquentiel est possible. Il est donc impossible de modifier le contenu d'un enregistrement (quand l'élément est lu, on est trop loin pour pouvoir le réécrire).
Les opérations de lecture et d'écriture sont exclusives les unes des autres. Il en résulte que les seules opérations globales possibles sont : la consultation, la création ou l'ajout en fin de fichier.

Type abstrait de données FAS

Sorte : FAS
Utilise : Chaîne, Elément
Opérations :
CréerFichier(Chaîne) → FAS
OuvrirEnEcriture(Chaîne) → FAS
OuvrirEnLecture(Chaîne) → FAS
Fermer(FAS) → FAS
Lire(FAS) → FAS, Elément
Ecrire(FAS, Elément) → FAS
FinFichier(FAS) → Booléen {observateur}

Les fichiers textes (FT)

Les fichiers textes sont des fichiers séquentiels dont les éléments sont des caractères. Pour plus de souplesse, il est possible de manipuler directement des chaînes de caractères. Elles sont écrites dans le fichier comme des séquences de caractères terminées par un marqueur de fin de ligne.

Type abstrait de données FT

Sorte: FT sous-type de FAS[Chaine]
Utilise : Chaîne
Opérations :
LireChaîne(FT) → FT, Chaîne
EcrireChaîne(FT, Chaîne) → FT

2. Les fichiers à accès direct (FAD)

Les fichiers à accès direct (ou fichiers indexés) sont des fichiers qui peuvent être adressés séquentiellement mais qui offrent en plus la possibilité d'un accès direct à un élément donné par son index (comme pour le TAD Séquence, par contre les algorithmes sont différents car le coût des accès disque est important ; on limitera donc les lectures de fichiers, quitte à utiliser des structures de données annexes en mémoire principale).

Contrairement aux FAS, les FAD permettent la modification d'éléments en milieu de fichier. Il est donc possible d'ouvrir un fichier en lecture/écriture.

Type abstrait de données FAD

Sorte: FAD sous-type de FAS
Utilise : Chaîne, Entier
Opérations :
OuvrirEnLectureEcriture(Chaîne) → FAD
AllerEn(FAD, Entier) → FAD

3. Autres opérations sur les fichiers

D'autres opérations sont nécessaires pour la maintenance d'un système de fichiers. Ces opérations sont notamment la destruction et le renommage qui sont assurées par le système d'exploitation.

Chapitre 9 – La récursivité

C'est un concept fondamental en mathématiques et en informatique : c'est une manière particulière d'appréhender certains problèmes.

Un module (procédure ou fonction) récursif est un module qui s'appelle lui-même : il est *défini en référence à lui-même*. Pourtant un module ne peut pas être défini uniquement en référence à lui-même sans qu'on introduise de définition circulaire. La récursion doit donc contenir une *condition de terminaison* qui autorise le module à ne plus faire appel à lui-même. Pour une récursivité qui boucle indéfiniment, l'arrêt se fait quand toute la mémoire disponible est utilisée car chaque appel récursif utilise de la mémoire.

Le calcul effectif d'un module récursif s'effectue à partir du moment où on a atteint le point d'appui.
La suite engendrée par les différents appels d'un module récursif a les propriétés d'une pile : à chaque appel, le contexte d'exécution (les paramètres effectifs, les variables locales ainsi que l'adresse de retour) est empilé sur la pile (zone de la mémoire prévue à cet effet).
Après l'exécution du module, on retourne à l'adresse qui est au sommet de la pile et on désempile.

1. Récursivité fondée sur une relation de récurrence

Quand la solution d'un problème est donnée par l'évaluation d'une suite récurrente, on a le choix entre un algorithme itératif et un algorithme récursif.

Exemple 1 : la fonction factorielle.
$u_o = 1$
$u_n = n * u_{n-1}$ si $n>0$

```
Fonction fac (n : entier) : entier
Var f,i : entier
Début
      f←1
      Pour i←1 à n Inc +1 Faire
            f←i*fac
      FinPour
Retourner(f)
Fin

Fonction fac (n : entier) : entier
Var f : entier
Début
      Si n=0
            Alors f←1
            Sinon f←n*fac(n-1){@2}
      FinSi
Retourner(f)
Fin
```

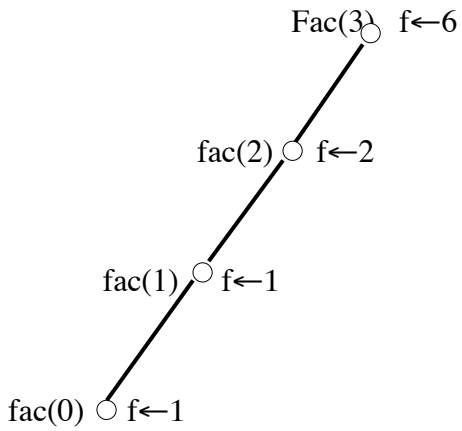
Appel de fac(3) :

```
...
écrire(fac(3)){@1}
...
```

~~@2 ; n=0~~
~~@2 ; n=1~~
~~@2 ; n=2~~
~~@1 ; n=3~~

fac = 3*(2*(1*(1))) = 6

On peut visualiser les différents appels réalisés par un graphe *nommé arborescence des appels récursifs* : le premier appel est extérieur et correspond au seul sommet du graphe qui n'a pas de prédécesseur (racine). Chaque sommet du graphe correspond à un appel de l'algorithme et est identifié par le nom du module exécuté avec le contexte d'exécution. Il existe un arc entre deux sommets X et Y ssi l'exécution qui correspond au sommet X engendre un nouvel appel correspondant au sommet Y.
On peut enrichir les graphes des appels en indiquant les actions réalisées sur chaque version exécutée.



L'intérêt de la récursivité se justifiera surtout par une nouvelle manière d'aborder les problèmes.

Exemple 2 : la suite de Fibonacci
$u_0 = u_1 = 1$
$u_n = u_{n-1} + u_{n-2}$ si $1<n$

```
Fonction fibo (n : entier) : entier
Const max = 10
Type  tab = tableau[1..max] d'entier
Var   i,f : entier
      t : tab
Début
      t[0]←1
      t[1]←1
      Pour i←2 à n Inc +1 Faire
            t[i]←t[i-1]+t[i-2]
      FinPour
      f←t[n]
retourner(f)
Fin
```

```
Fonction fibo (n : entier) : entier
Var f : entier
Début
      Si n<=1
            Alors f←1
            Sinon f←fibo(n-1){@2}+fibo(n-2){@3}
      FinSi
Retourner(f)
Fin
```

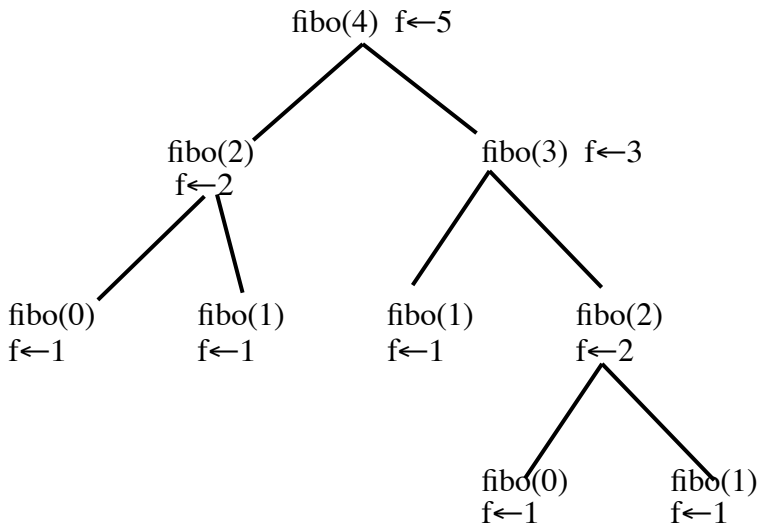
Appel de fibonacci(4) :

```
...
écrire(fibo(4)){@1}
...
```

~~@3 ; n=0~~
~~@2 ; n=1~~
~~@3 ; n=2~~
~~@3 ; n=1~~
~~@3 ; n=0~~
~~@2 ; n=1~~
~~@2 ; n=2~~
~~@2 ; n=3~~
~~@1 ; n=4~~

fibo = 1+1+1+1+1=5

Arborescence des appels récursifs de fibo(4)



Le résultat est obtenu à partir d'une somme de 1. L'algorithme est peu performant car il y a répétition du calcul, la méthode est non optimale.

2. Schéma général d'un algorithme récursif

Il existe des algorithmes récursifs dont la solution ne découle pas d'une suite récurrente.
Dans un algorithme récursif, on distingue deux parties : la condition de terminaison (point d'appui) et l(es)'appel(s) récursif(s).

```
Module A(p1,..,pn)
Début
      Si cond
            Alors I1...
            Sinon I2
                  A(f(p1,..,pn))
                  I3
                  A(g(p1,..,pn))
                  ...
                  Im
      FinSi
Fin
```

Un algorithme est dit *simplement récursif* s'il ne contient qu'un seul appel récursif.
Un appel récursif est *terminal* s'il n'est jamais suivi par l'exécution d'instructions dans le module.
On dit qu'on a une *récursivité indirecte* ou *croisée* quand un module A sans appel récursif appelle un module B qui appelle A.

Chapitre 10 – Notion de complexité

Quand on tente de résoudre un problème, la question du choix d'un algorithme se pose. Un algorithme
- doit être simple à mettre en oeuvre et à comprendre,
- doit utiliser intelligemment les ressources de l'ordinateur (place mémoire et temps d'exécution).

Ces deux aspects sont souvent contradictoires.

Le temps d'exécution d'un programme dépend
- des données entrant dans le programme,
- de la qualité du code généré par le compilateur,
- de la nature et la vitesse d'exécution des instructions du microprocesseur,
- de la complexité algorithmique du programme.

Temps d'exécution d'un programme et complexité algorithmique sont intimement liés.

Assez souvent, le temps d'exécution d'un programme ne dépend pas de la nature précise des données mais de leur « taille ». Dans ces conditions, il est habituel de parler d'un temps d'exécution de *T(n)* pour un programme portant sur des données de taille *n*.
Exemple : $T(n) = cn^2$, où *c* est une constante.
Il est possible de se représenter T(n) comme le nombre d'instructions « élémentaires » exécutées par une machine formelle.

Pour un certain nombre de programmes, la complexité est une fonction de la nature des données et non de leur taille seulement. Dans ce cas, on définit T(n) comme la complexité dans le pire des cas = mesure de la complexité maximum sur tous les ensembles de données possibles de taille n pour un programme donné.
Il est aussi possible de définir la *complexité en moyenne* $T_{moy}(n)$, sur tous les ensembles de données de taille n (souvent difficile à calculer).

Exemple de formulation : « la complexité d'un tel algorithme est proportionnelle à n^2 ».
La constante de proportionnalité n'est pas précisée car elle est déterminée par les performances du compilateur et de la machine.

1. Notation *O* et *Ω*

Pour une complexité algorithmique en *O(f(n))*, il existe deux constantes positives c et n_0 :
$\forall\ n \geq n_0, T(n) \leq cf(n)$
Dire que T(n) est en O(f(n)), c'est garantir que f(n) est un majorant de la fonction de complexité T(n).
g(n) est un minorant de T(n) : T(n) est en Ω(g(n)), $\exists$ c (constante positive) t.q. $T(n) \geq cg(n)$.

Les programmes peuvent être comparés sur la base de leur fonction de complexité, à la constante de proportionnalité près.

Règle de la somme :
Supposons que deux modules P_1 et P_2 aient une complexité $T_1(n)$ en O(f(n)) et $T_2(n)$ en O(g(n)). Alors la complexité de P_1 suivi de P_2 est $T_1(n)+T_2(n) = O(\max(f(n), g(n)))$.

2. Analyse des programmes

Le seul paramètre acceptable pour l'évaluation de la complexité d'un programme est *n*, la taille des données.

- La complexité de toute affectation, opération de lecture ou d'écriture peut en général se mesurer par O(1).
- La complexité d'une suite d'instructions est déterminée par la règle de la somme.
- La complexité conditionnelle (si) est celle des instructions exécutées (règle de la somme pour un si-alors-sinon), plus celle de l'évaluation de la condition (généralement égale à O(1)).
- La complexité d'une boucle est la somme cumulée, sur toutes les itérations, de la complexité des instructions exécutées dans le corps de la boucle plus le temps consacré à l'évaluation de la condition de sortie de la boucle (généralement égale à O(1)). Assez souvent, cette complexité est le produit du nombre d'itérations par la plus grande complexité rencontrée dans l'exécution d'une boucle (règle du produit).
- Appels de procédures non récursives : on commence par celles qui n'en appellent aucune autre, puis celles qui appellent uniquement celles qu'on vient de traiter en incluant les complexités qu'on vient de calculer, et ainsi de suite.
- Appels de procédures récursives : il faut construire une relation de récurrence pour T(n).

3. Exemples

- *Exemple 1 :* tri à bulles

```
Const max = 100
Type tab = tableau [1..max] d'entiers
Procedure Tribulle (E/S t : tab)
Var i,j, temp : entier
Début
      Pour i←1 à n-1 Inc +1 Faire
            Pour j←n à i+1 Inc -1 Faire
                  Si t[j-1]>t[j]
                        Alors temp←t[j-1]
                              t[j-1]←t[j]
                              t[j]←temp
                  FinSi
            FinPour
      FinPour
Fin
```

Analyse dans le pire des cas :
L'évaluation de la condition du `si` est en O(1). Dans le pire des cas, elle sera toujours vraie.
Les instructions faites dans la partie `Alors`, sont en O(1).
Nombre d'itérations du `Pour j` : n-i
Nombre d'itérations du `Pour i` : n-1

$T(n) = \sum_{i=1}^{n-1} (n-i) = n(n-1)/2.$ T(n) est en $O(n^2)$.

- *Exemple 2 :* fonction factorielle

```
Fonction fac (n : entier) : entier
Var f : entier
Début
      Si n≤1
            Alors f←1
            Sinon f←n*fac(n-1)
      FinSi
Retourner(f)
Fin
```

T(n) = c + T(n-1)
T(n-1) = c+ T(n-2)
T(n) = c+ [c + T(n-2)] = 2*c + T(n-2)
T(n) = i*c + T(n-i) si n>i.
Pour i=n-1, T(n) = c*(n-1)+T(1) = c*n + d, où d est une constante positive. T(n) est en O(n).

Chapitres 11 - Algorithmes de tri

Un algorithme de tri permet d'organiser une collection d'objets selon un ordre déterminé. Cette collection doit être munie d'une relation d'ordre (en générale un ordre total, par exemple l'ordre numérique et l'ordre lexicographique). Il est possible de définir un algorithme de tri indépendamment de la fonction d'ordre utilisée (qui permet de comparer tout couple d'éléments de la collection).

1. Classification des algorithmes de tri

Cette classification permet de choisir l'algorithme le plus adapté au problème traité, tout en tenant compte des contraintes imposées par celui-ci. On entend habituellement par « algorithme de tri » un algorithme général de tri par comparaison.
Les principales caractéristiques qui permettent de différencier ces algorithmes sont :

- la complexité algorithmique,
- les ressources nécessaires (notamment en termes d'espace mémoire utilisé)
- et le caractère stable.

Complexité algorithmique :
Pour certains des algorithmes de tri les plus simples, $T(n) = O(n^2)$, pour les tris plus élaborés, $T(n) = O(n \cdot \log(n))$.
On peut montrer que la complexité temporelle en moyenne et dans le pire des cas d'un algorithme basé sur une fonction de comparaison ne peut pas être meilleure que $\boldsymbol{n \cdot \log(n)}$. Les tris qui ne demandent que $n \cdot \log(n)$ comparaisons en moyenne sont alors dits optimaux.

Pour certains types de données (entiers, chaînes de caractères de taille bornée), il existe cependant des algorithmes plus efficaces comme le tri comptage ou le tri par base. Ces algorithmes n'utilisent pas la comparaison entre éléments mais nécessitent des hypothèses sur les objets à trier. Par exemple, le tri comptage et le tri par base s'appliquent à des entiers que l'on sait appartenir à l'ensemble $[1, m]$ avec comme hypothèse supplémentaire pour le tri par base que m soit une puissance de 2.

Ressources nécessaires
Un algorithme est dit en place s'il n'utilise qu'un nombre très limité de variables et qu'il modifie directement la structure qu'il est en train de trier. Ceci nécessite l'utilisation d'une structure de donnée adaptée (un tableau par exemple). Ce caractère peut être très important si on ne dispose pas d'une grande quantité de mémoire utilisable.

Caractère stable
Un algorithme est dit stable s'il garde l'ordre relatif des quantités égales pour la relation d'ordre : lorsque deux éléments sont égaux (même clé), l'algorithme de tri conserve l'ordre dans lequel ces deux éléments se trouvaient avant son exécution.
Exemple, soit la suite d'éléments `[(4, 1) (3, 2) (3, 3) (5, 4)]` qu'on souhaite trier sur leur première coordonnée (la clé), la seconde étant l'indice initial dans la suite, deux cas sont possibles :

```
[(3, 2) (3, 3) (4, 1) (5, 4)]   (ordre relatif maintenu)
[(3, 3) (3, 2) (4, 1) (5, 4)]   (ordre relatif changé)
```

Les algorithmes de tri instables peuvent être retravaillés spécifiquement afin de les rendre stables, cependant cela peut être aux dépens de la rapidité et/ou peut nécessiter un espace mémoire supplémentaire. Parmi les algorithmes listés plus bas, les tris étant stables sont : le tri à bulles, le tri par insertion et le tri fusion. On peut facilement obtenir la stabilité d'un tri si l'on associe à chaque élément sa position initiale. Pour cela, on peut créer un deuxième

tableau de même taille pour stocker l'ordre initial (on renonce alors au caractère *en place* du tri).

Un *tri interne* s'effectue entièrement en mémoire centrale alors qu'un *tri externe* utilise des fichiers sur une mémoire de masse pour trier des volumes trop importants pour pouvoir tenir en mémoire.

Certains *algorithmes parallèles* permettent d'exploiter les capacités multitâches de la machine.

Tris par comparaison
Algorithmes lents
Ces algorithmes sont lents pour plus de 20 éléments parce qu'ils sont en $O(n^2)$.

- Tri à bulles : Algorithme quadratique, $T(n) = O(n^2)$, en moyenne et dans le pire des cas, stable et en place ; amusant mais pas efficace.
- Tri par sélection : Algorithme quadratique, $T(n) = O(n^2)$, en moyenne et dans le pire des cas, pas stable si tri sur place ; rapide lorsque l'on a moins de 7 éléments.
- Tri par insertion : Algorithme quadratique, $T(n) = O(n^2)$, en moyenne et dans le pire des cas, stable et en place. C'est le plus rapide et le plus utilisé pour des listes de moins de 15 éléments.

Algorithmes plus rapides (liste non exhaustive)

- Tri fusion (*merge sort*) : O(n log n) en moyenne et dans le pire des cas, stable mais pas en place.
- Tri rapide (*quick sort*) : O(n log n) en moyenne, mais en $O(n^2)$(quadratique) au pire cas, en place mais pas stable.
- Tri par tas (*heap sort*) : O(n log n) en moyenne et dans le pire des cas, en place mais pas stable. Toujours environ deux fois plus lent que le tri rapide, c'est-à-dire aux alentours de O(n log n), il est donc intéressant de l'utiliser si l'on soupçonne que les données à trier seront souvent des cas quadratiques pour le tri rapide.

Le tri en place le plus rapide est le tri rapide.
Les tris lents (à bulles, insertion, sélection) sont beaucoup trop lents pour être utilisés dans le cas de collections importantes (> 30 000 éléments).

Tris utilisant des structures de données

- Tri comptage ou tri par dénombrement (*counting sort*) : algorithme linéaire en O(n), stable mais nécessite l'utilisation d'une seconde liste de même longueur que la liste à trier. Son utilisation relève de la condition que les valeurs à trier sont des entiers naturels dont on connaît les extrema.
- Tri par base (*radix sort*) : c'est aussi un tri linéaire dans certaines conditions (moins restrictives que pour le tri par comptage), $T(n) = O(n)$, stable mais nécessite aussi l'utilisation d'une seconde liste de même longueur que la liste à trier ;
- Tri par paquets (*bucket sort*) : stable et en complexité linéaire O(n), part de l'hypothèse que les données à trier sont réparties de manière uniforme sur un intervalle réel [a, b[.

Tris volumineux
Les algorithmes de tri doivent aussi être adaptés en fonction des configurations informatiques sur lesquels ils sont utilisés. Dans les exemples cités plus haut, on suppose que toutes les données sont présentes en mémoire centrale (ou accessibles en mémoire virtuelle). La situation se complexifie si l'on veut trier des volumes de données supérieurs à la mémoire centrale disponible (ou si l'on cherche à améliorer le tri en optimisant l'utilisation de la hiérarchie de mémoire). Ces algorithmes sont souvent basés sur une approche assez voisine de celle du tri fusion : on découpe le volume de données à trier en sous-ensembles de taille inférieure à la mémoire rapide disponible puis on trie chaque sous-ensemble en mémoire centrale pour former des *monotonies* (sous-ensembles triés) et on interclasse ces monotonies.

2. Tri fusion (mergesort)

C'est un tri récursif qui est une application de « diviser pour régner ».
Le principe récursif est le suivant :

1. on divise le tableau à trier en deux sous-tableaux de même longueur (±1),
2. on trie les deux sous-tableaux indépendamment,
3. et on fusionne les deux sous-tableaux triés.

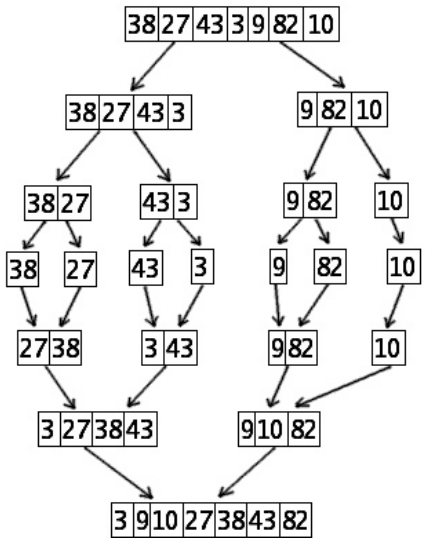
```
Const inf = 1
      sup = 100
Type tab = tableau [inf,sup] d'entier

Procédure trier-fusion (E/S t : tab ; E inf,sup : entier) {inf ≤ sup}
Var m : entier
Début
  si inf=sup
    alors écrire(''le tableau est trié'')
    sinon
      m←(inf+sup) div 2
      trier-fusion(t,inf,m)
      trier-fusion(t,m+1,sup)
      fusionner(t,inf,m,sup)
  finSi
Fin
```

La procédure `fusionner` permet de fusionner deux parties chacune triée du tableau `t` dans ce même tableau en utilisant un tableau temporaire `temp`. L'appel `fusionner(t,inf,m,sup)` fusionne les deux parties de `t` qui vont de `inf` à `m` et de `m+1` à `sup` (`inf≤m≤sup`).

```
Procédure fusionner (E/S t : tab, E d,m,f : entier )
Var i,j,k : entier
    temp : tab
Début
  i←d
  j←m+1
  Pour k←1 à f-d+1 inc +1 Faire
    Si i≤m et j≤f
       Alors
         Si t[i]≤t[j]
           Alors
             temp[k]←t[i]
             i←i+1
           Sinon
             temp[k]←t[j]
             j←j+1
         FinSi
       Sinon
         Si i≤m
           Alors
             temp[k]←t[i]
             i←i+1
           Sinon
             temp[k] ←t[j]
             j←j+1
         FinSi
    FinSi
  FinPour
  Pour k←1 à f-d+1 inc +1 Faire
     t[d+k-1]←temp[k]
  FinPour
Fin
```

Exemple :



3. Tri rapide (quicksort)

C'est aussi un tri récursif qui est une application de « diviser pour régner ».
Le principe récursif est le suivant :

1. on partitionne le tableau en deux sous-tableaux afin que les éléments du sous-tableau gauche (resp. droit) soient plus petits (resp. grands) ou égaux à un élément appelé pivot,
2. et on trie les deux sous-tableaux indépendamment.

```
Procédure tri-rapide(E/S t : tab, E inf,sup : entier)
Var p : entier
Début
  si inf<sup
    Alors
      partionner(t,inf,sup,p)
      tri-rapide(t,inf,p-1)
      tri-rapide(t,p+1,sup)
  Finsi
Fin
```

L'algorithme de partitionnement réduit le nombre d'échanges effectués lors du partitionnement. Son principe est le suivant :

1. on cherche le plus petit i tel que T[i] soit supérieur ou égal au pivot,
2. on cherche le plus grand j tel que T[j] soit inférieur ou égal au pivot,
3. si i<j, on échange T[i] et T[j] puis on recommence.

```
Procédure partitionner (E/S t : tab, E d,f : entier, S p : entier )
Var i,j,pivot : entier
Début
   pivot←t[d]
   i←d
   j←f
   TantQue i≤j Faire
      TantQue t[i]≤pivot et i≤j Faire
         i←i+1
      FinTantQue
      TantQue t[j]>pivot et i≤j Faire
         j←j-1
      FinTantQue
      Si i≤j alors
         échanger(t[i],t[j])
      FinSi
   FinTantQue
   p←j
   échanger(t[d],t[j])
Fin
```

Exemple : 6 3 0 9 1 7 8 2 5 4

6 est le pivot !

6 3 0 9 1 7 8 2 5 4
échange de 9 et 4
6 3 0 4 1 7 8 2 5 9
échange de 7 et 5
6 3 0 4 1 5 8 2 7 9
échange de 8 et 2
6 3 0 4 1 5 2 8 7 9
échange de 6 et 2
2 3 0 4 1 5 6 8 7 9

Et on recommence sur les deux sous-tableaux …

Chapitre 12 – Les pointeurs

Notion de variable dynamique : elle peut être créée ou détruite au cours de l'exécution du bloc dans lequel elle est déclarée ; l'espace-mémoire rendu libre peut être récupéré et l'accès à la valeur se fait par un pointeur.

1. Définition

Un pointeur est une variable dont les valeurs sont des adresses. C'est une variable statique P appelé *variable pointeur* ; par contre, dans la plupart des cas, la variable pointée par P est dynamique.
Etant donné un type quelconque T appelé *type de base*, une variable P de type pointeur est une variable scalaire qui peut prendre comme valeurs les adresses des variables de type T. Il existe donc autant de types pointeurs que de types de base.

P = 43
43 | 2345 |
Ici, le type de base est Entier.

2. Déclaration

```
Type pEntier = ^Entier
Var p : pEntier
```

3. Opérations

Sorte : Pointeur
Utilise : TypeDeBase
Operations :

Allouer (TypeDeBase) → Pointeur	{constructeur}
Récupérer(Pointeur) →	{observateur}
^Pointeur → TypeDeBase	{observateur}

« Allouer » réserve en mémoire principale un espace assez grand pour contenir un élément du type de base et renvoie un pointeur sur cet élément. C'est une opération indispensable pour la manipulation des pointeurs. La valeur de l'élément alloué est indéfinie (non nulle).

« Récupérer » rend la mémoire au système d'exploitation. P est alors inutilisable.

P^ permet d'accéder à l'élément pointé par P.

Valeur particulière

Quel que soit son type de base, un pointeur peut prendre une valeur particulière nil : signifie que « le pointeur ne pointe sur rien ». Donc, si p=nil alors p^ renvoie toujours une erreur.

Exemple

```
Type pEntier = ^Entier
Var p,q : pEntier

p←allouer(Entier)
q←allouer(Entier)
p^←3
q^←p^
écrire(p^=q^) (* VRAI *)
écrire(p=q)   (* FAUX *)
q←p
écrire(p^=q^) (* VRAI *)
écrire(p=q)   (* VRAI *)
récupérer(p)
écrire(q^)    (* ERREUR ! ! ! car q est aussi récupéré *)
```

(* ERREUR ! ! ! Plus de moyen de récupérer la case pointée par q à sa création *)

Chapitre 13 – Structures séquentielles : les listes

Une liste sur un ensemble E est une suite finie (e_1, ..., e_n) d'éléments de E. La liste est vide si n=0. On différencie la place d'un élément dans la liste de l'élément lui-même. Les insertions et les suppressions peuvent se faire à toute place de la liste.

1. Type abstrait de données : Liste

TAD Place

Nécessité de commencer par définir le TAD `Place` qui est une case mémoire contenant un élément et un accès à la place suivante.

Sorte : Place
Utilise : Elément, TypeDebase
Opérations :

affecter-elt(Place, Elément) → Place	{constructeur}
affecter-succ(Place, Place) → Place	{constructeur}
succ(Place) → Place	{constructeur}
contenu(Place) → Elément	{observateur}

TAD Liste

Sorte : Liste
Utilise : Elément, Place, Entier, Booléen
Opérations :

créer-liste-vide() → Liste	{constructeur}
insérer(Liste, Entier, Elément) → Liste	{constructeur}
supprimer(Liste, Entier) → Liste	{constructeur}
concaténer(Liste, Liste) → Liste	{constructeur}
accès(Liste, Entier) → Place	{observateur}
longueur(Liste) → Entier	{observateur}
liste-vide(Liste) → Booléen	{observateur}
ième(Liste, Entier) → Elément	{observateur}

ième(l,i) = contenu(accès(l,i))

2. Implémentation du TAD Liste

Représentation contiguë (par tableau)

Les éléments de la liste sont mémorisés dans les cellules contiguës d'un tableau. Les notions de rang et de place sont confondues. Les insertions et suppressions dans la liste impliquent des décalages dans le tableau (i≠n).
Le ième élément de la liste est placé dans la ième cellule du tableau.

```
Type liste = Enregistrement
                  t : tableau [1..n] d'élément
                  dernier : entier
             FinEnregistrement

Var l : liste


Procédure insérer (E i : entier, e : élément, E/S l : liste)
Var j : entier
Début
Si (i>l.dernier+1) ou (i<1)
      Alors erreur
      Sinon Pour j←l.dernier à i inc -1 Faire
                  l.t[j+1]←l.t[j]
            FinPour
            l.dernier←l.dernier+1
            l.t[i]←e
FinSi
Fin

Procédure supprimer (E i : entier, E/S l : liste)
Var j : entier
Début
Si (i>l.dernier) ou (i<1)
      Alors erreur
      Sinon Pour j←i à l.dernier inc +1 Faire
                  l.t[j]←l.t[j+1]
            FinPour
            l.dernier←l.dernier-1
FinSi
Fin

Fonction localiser (E l : liste, e : élément) : entier
Var i, loc : entier
Début
i←0
Répète
      i←i+1
Jusqu'à (i>l.dernier) ou (l.t[i]=e)
Si i>l.dernier
      Alors loc←0
      Sinon loc←i
FinSi
Retourner(loc)
Fin
```

Représentation chaînée

On utilise des pointeurs pour lier entre eux les éléments successifs, et la liste est alors déterminée par l'adresse de son premier élément.

```
Type  liste = ^cellule
      cellule = Enregistrement
                  val : élément
                  succ : liste
                FinEnregistrement

Var l : liste

Procédure insérer (E i : entier, e : élément, E/S l : liste)
Var   j : entier
      p, q : liste
Début
Si i=1
      Alors p←allouer(cellule)
            p^.val←e
            p^.succ←l
            l←p
      Sinon j←2
            q←l
            TantQue (j<i) et (q≠nil) Faire
            j←j+1
            q←q^.succ
            FinTantQue
            Si q≠nil
            Alors p←allouer(cellule)
                  p^.val←e
                  p^.succ←q^.succ
                  q^.succ←p
            FinSi
FinSi
Fin

Procédure supprimer (E i : entier, E/S l : liste)
Var   j : entier
      p, q : liste
Début
Si l≠nil
     Alors Si i=1
               Alors p←l
                     l←l^.succ
                     récupérer(p)
               Sinon j←2
                     q←l
                     TantQue (j<i) et (q^.succ≠nil) Faire
                         j←j+1
                         q←q^.succ
                     FinTantQue
                     Si q^.succ≠nil
                        Alors p←q^.succ
                              q^.succ←q^.succ^.succ
                              récupérer(p)
                     FinSi
          FinSi
FinSi
Fin
```

```
Fonction localiser (l : liste, e : élément) : entier
Var   i, j : entier
      q : liste
Début
j←0
i←0
q←l
TantQue (q≠nil) et (j=0) Faire
      i←i+1
      Si q^.val=e
            Alors j←i
            Sinon q←q^.succ
      FinSi
FinTantQue
retourner(j)
Fin
```

Représentation par faux pointeurs

Certains langages de programmation ne comportent pas de pointeur. On peut alors utiliser un tableau et des entiers pour représenter l'indice des éléments dans le tableau. Il existe un nombre assez grand de cases pour y avoir plusieurs listes à la fois. Pour pouvoir facilement insérer et supprimer des éléments dans les listes, il est important de chaîner entre elles les cases libres du tableau.

```
Type  listab = tableau[1..n] de cellule
      cellule = Enregistrement
                  val : élément
                  succ : entier
                FinEnregistrement

Var t : listab
```

l1 = 4 (a,~~b~~,c) {Suppression de b}
l2 = 6 (d,e)
libre = ~~8~~ 7 {q}

t

1		9
2	c	0
3	e	0
4	a	~~7~~ 2 {p}
5		10
6	d	3
7	~~b~~	~~2~~ 8
8		1
9		5
10		11
11		12
12		0

Pour insérer `e` dans une liste `l`, on utilise la première cellule de `libre` et on la place à la bonne position dans `l`. `e` est ensuite placé dans le champs `val` de cette cellule. Pour supprimer une cellule de `l`, on la retire de `l` et on la met en tête de `libre`.

Nécessité d'une procédure `transfert(p,q)` :

- prendre la cellule `c` sur laquelle pointe `p`
- faire pointer `q` sur `c` en sauvegardant `q`
- faire pointer `p` sur l'élément pointé par `c`
- faire pointer `c` sur l'élément pointé par `q` auparavant.

```
Procédure transfert(E/S t : listab, p,q :entier) {p pointe sur c}
Var temp : entier
Début
temp←q
q←p
p←t[q].succ
t[q].succ←temp
Fin

Procédure supprimer(E/S t : listab, l, libre : entier ; E i :entier)
Var   j,q : entier
Début
j←1
q←l
TantQue (j<i) et (q≠0) Faire
      q←t[q].succ
      j←j+1
FintantQue {q pointe sur le p voulu de transfert}
Si q≠0
      Alors transfert(t,q,libre)
FinSi
Fin

Procédure insérer(E/S t : listab, l, libre : entier ;
                  E i :entier, e : élément)
Var   j,q : entier
Début
j←1
q←l
TantQue (j<i) et (q≠0) Faire
      q←t[q].succ
      j←j+1
FintantQue {q pointe sur le p voulu de transfert}
Si q≠0
      Alors transfert(t,libre,q)
      t[q].val←e
FinSi
Fin
```

Chapitre 14 – Structures séquentielles : les piles

Une pile est un cas particulier de liste : les insertions et les suppressions se font à une seule extrémité, appelé *sommet de la pile*. LIFO : Last In First Out.

1. Type abstrait de données : Pile

TAD Pile

Sorte : Pile
Utilise : Elément, Booléen
Opérations :

créer-pile-vide() → Pile	{constructeur}
empiler(Pile, Elément) → Pile	{constructeur}
dépiler(Pile) → Pile	{constructeur}
sommet(Pile) → Elément	{observateur}
pile-vide(Pile) → Booléen	{observateur}

2. Implémentation du TAD Pile

Représentation contiguë (par tableau)

```
Type pile = Enregistrement
                t : tableau [1..max] d'élément
                sommet : entier
            FinEnregistrement

Var p : pile
```

Plus de décalage par rapport aux listes quand on empile ou dépile. Ces opérations se font toujours sur le 1er élément. `Sommet` indique à tout instant la position du 1er élément de la pile. L'implémentation du type pile est triviale.

Représentation chaînée (par pointeurs)

Les suppressions et les insertions se font toujours en tête de liste.

```
Type  pile = ^cellule
      cellule = Enregistrement
                  val : élément
                  succ : pile
                FinEnregistrement

Var p : pile

Procédure empiler (E e : élément, E/S p : pile)
Var   q : pile
Début
q←allouer(cellule)
q^.val←e
q^.succ←p
p←q
Fin
```

```
Procédure dépiler (E/S p : pile)
Var   q : pile
Début
q←p
p←p^.succ
récupérer(q)
Fin
```

Chapitre 15 – Structures séquentielles : les files

Une file est un cas particulier de liste : on fait les ajouts à une extrémité et les suppressions se font à l'autre extrémité. FIFO : First In First Out.

1. Type abstrait de données : File

TAD File

Sorte : File
Utilise : Elément, Booléen
Opérations :

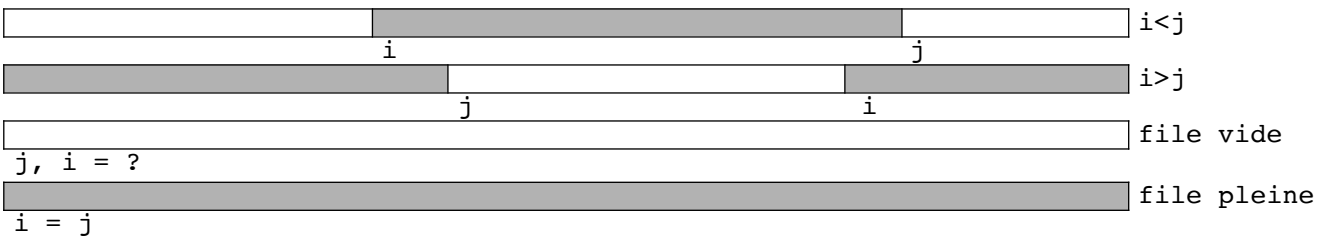
créer-file-vide() → File	{constructeur}
ajouter(File, Elément) → File	{constructeur}
retirer(File) → File	{constructeur}
premier(File) → Elément	{observateur}
file-vide(File) → Booléen	{observateur}

2. Implémentation du TAD File

Représentation contiguë (par tableau)

```
Type file = Enregistrement
                t : tableau [1..max] d'élément
                tête, queue : entier
            FinEnregistrement

Var f : file
```



tête : indice du premier élément de la file.
queue : indice de la première case libre après la file.
On fait progresser ces indices modulo la taille max du tableau.
L'implémentation des opérations du TAD File est triviale.

Représentation chaînée (par pointeurs)

```
Type  file = Enregistrement
                tête, queue : ^cellule
             FinEnregistrement
      cellule = Enregistrement
                  val : élément
                  succ : ^cellule
                FinEnregistrement

Var f : file


Procédure ajouter (E e : élément, E/S f : file)
Var   q : ^cellule
Début
q←allouer(cellule)
q^.val←e
q^.succ←nil
f.queue^.succ←q
f.queue←q
Fin

Procédure retirer (E/S f : file)
Var q : ^cellule
Début
q←f.tête
f.tête←f.tête^.succ
récupérer(q)
Fin
```

Chapitre 16 – Structures arborescentes : les arbres binaires

Un arbre est un ensemble d'éléments appelés nœuds dont un se singularise comme la racine, liés par une relation induisant une structure hiérarchique parmi les nœuds. Un nœud, comme tout élément d'une liste, peut-être de n'importe quel type.

1. Définitions

Arbre
- Un nœud unique, par lui-même, est un arbre. Il est aussi la racine de l'arbre.
- Si n est un noeud et A_1, ..., A_k sont des arbres de racine respective n_1, ..., n_k, on peut construire un nouvel arbre en associant comme parent unique aux noeuds n_1 , ..., n_k le noeud n. Dans cet arbre, n est la racine et A_1, ..., A_k sont les sous-arbres de cette racine. Les noeuds n_1, ..., n_k sont appelés les fils de n.

Utilité d'inclure l'arbre vide (noté Λ) parmi l'ensemble des arbres, c'est un arbre sans noeud.

Arbre binaire
- Un arbre binaire est soit un arbre vide, soit un arbre où chaque noeud a un fils gauche, un fils droit ou les 2 à la fois.
  $B = \Lambda + <o, B_1, B_2>$ où B_1 et B_2 sont des arbres binaires disjoints et 'o' est un noeud appelé racine. B_1 est le sous-arbre gauche et B_2 est le sous-arbre droit.
  Remarque : non symétrie gauche/droite des arbres binaires.
- On dit que C est un sous-arbre de B ssi C=B ou $C=B_1$ ou $C=B_2$ ou C est un sous-arbre de B_1 ou de B_2.
- Si un noeud n_i a pour fils gauche (ou droit) un noeud n_j, on dit que n_i est le père de n_j.
- Deux noeuds qui ont le même père sont dits frères.
- Le noeud n_i est un ascendant du noeud n_j ssi ni est le père de n_j, ou un ascendant du père de n_j ; et n_i est un descendant de n_j ssi n_i est fils de n_j, ou n_i est un descendant d'un fils de n_j.
- Un noeud sans fils est appelé feuille. Les noeuds qui ne sont pas des feuilles sont appelés noeuds internes.
- On appelle branche de l'arbre tout chemin (suite de noeuds consécutifs) de la racine à une feuille. Un arbre a donc autant de branches que de feuilles.
- La hauteur d'un noeud (ou profondeur ou niveau) est définie récursivement par :
  soit x noeud de B,
  $h(x) = 0$ si x est la racine de B
  $h(x) = 1 + h(y)$ si y est le père de x.
- La hauteur d'un arbre B est
  $h(B) = \max \{h(x), x \text{ noeud de } B\}$
- Un arbre binaire est **entier** ssi tous ses nœuds possèdent zéro ou 2 fils.
- Un arbre binaire est **complet** ssi ses feuilles ont pour profondeur *n* ou *n-1*.
- Un arbre binaire est **parfait** ssi il est entier et toutes ses feuilles sont à la même profondeur.

2. Type abstrait de données : ArbreBinaire

TAD ArbreBinaire
Sorte : ArbreBinaire
Utilise : Noeud, Booléen, Elément

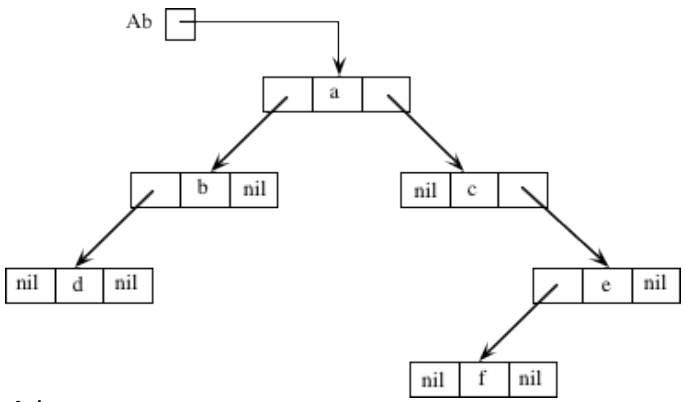
Opérations :

créer-arbre-vide :	→ ArbreBinaire
créer :	Noeud, ArbreBinaire, ArbreBinaire → ArbreBinaire
racine :	ArbreBinaire → Noeud
père :	Noeud, ArbreBinaire → Noeud
fils-gauche :	ArbreBinaire → ArbreBinaire
fils-droit :	ArbreBinaire → ArbreBinaire
arbre-vide :	ArbreBinaire → Booléen
affecter-val :	Noeud, Elément → Noeud
val :	Noeud → Elément

3. Implémentation du TAD ArbreBinaire

Par pointeurs
Chaque noeud contient deux pointeurs, l'un vers le sous-arbre gauche et l'autre vers le sous-arbre droit ; et l'arbre est déterminé par l'adresse de sa racine.

```
Type  arbrebin = ^noeud
      noeud = Enregistrement
                  val : élément
                  gauche, droit : arbrebin
              FinEnregistrement
```




```
Var Ab : arbrebin

fils-gauche(Ab) = Ab^.gauche
fils-droit(Ab) = Ab^.droit
```

Par tableau
Les noeuds de l'arbre sont mis en bijection avec un ensemble d'adresses dans des « cases » gauche et droit d'un tableau correspondant aux racines des sous-arbres gauche et droit. L'adresse de la racine de l'arbre est contenue dans une variable Ab.

```
Type  cellule = Enregistrement
                  val : élément
                  gauche, droit : entier
                FinEnregistrement
      tabarbre = tableau [1..n] de cellule
      arbrebin = entier
```

```
Ab = 4
libre = 1   {Chaînage des cases libres dans un arbre}
```

t	val	gauche	droite
1	-	3	6
2	c	0	8
3	-	9	0
4	a	7	2
5	d	0	0
6	-	0	0
7	b	5	0
8	e	10	0
9	-	0	0
10	f	0	0

```
Var Ab : arbrebin
    t : tabarbre
Ab ← 4
fils-gauche(Ab) = t[Ab].gauche
fils-droit(Ab) = t[Ab].droit
```

4. Parcours

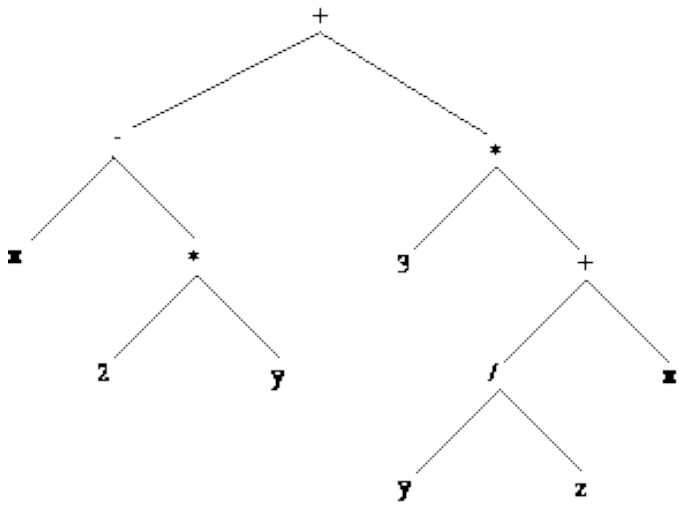
Le plus fréquent est le *parcours en profondeur à gauche* : il suit le chemin qui part à gauche de la racine et va toujours le plus à gauche possible en suivant l'arbre. Dans ce parcours, chaque noeud est rencontré trois fois : à gauche, en dessous, à droite. A chacun de ces passages peut correspondre un certain traitement du noeud rencontré.

```
Procédure profGauche (E A : Arbrebin)
Début
      Si arbre-vide(A)
            Alors terminaison
            Sinon traitement1
            profgauche(fils-gauche(A))
            traitement2
            profgauche(fils-droit(A))
            traitement3
      FinSi
Fin
```

On peut s'arrêter avant l'arbre vide, c'est-à-dire aux feuilles : pas de fils-gauche, ni de fils-droit.

ordre préfixé (préordre) : seul traitement1 est appliqué
ordre infixé (symétrique) : seul traitement2 est appliqué
ordre sufixé (postfixé) : seul traitement3 est appliqué

Exemple : arbre étiqueté représentant l'expression arithmétique x-2*y+3*(y/z+x)



préfixe : traitement1 = affiche(val(racine(A))
+-x*2y*3+/yzx
postfixe : traitement3 = affiche(val(racine(A))
x2y*-3yz/x+*+
infixe : traitement2 = affiche(val(racine(A))
x-2*y+3*y/z+x

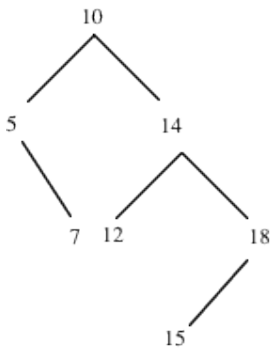
Problème de parenthèsage, d'où l'algorithme suivant :

```
Procédure infixe (E A : arbrebin)
Début
Si arbre-vide(fils-gauche(A)) et arbre-vide(fils-droit(A))  {feuille(A)}
     Alors affiche(val(racine(A)))
     Sinon affiche('(')
           infixe(fils-fauche(A))
           affiche(val(racine(A)))
           infixe(fil-droit(A))
           affiche(')')
FinSi
Fin
```

5. Arbre Binaire de Recherche (ABR)

Propriété fondamentale : dans un ABR, le sous-arbre gauche (resp. droit) du noeud x ne contient que des éléments de valeur inférieure (resp. supérieure) à celle de l'élément contenue en x. Cette propriété est satisfaite par tous les noeuds de l'arbre, y compris la racine.

Exemple :
Le test d'appartenance à l'ensemble des éléments de l'arbre est aisé. Principe récursif simple.



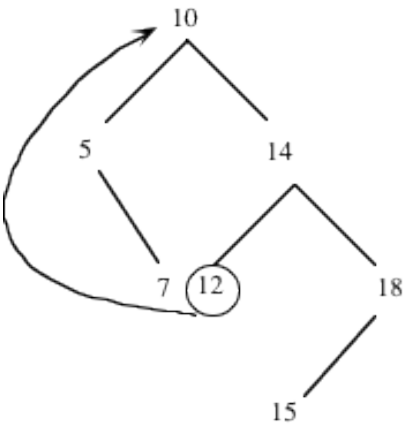
L'insertion est aussi facile à écrire. On cherche à positionner x en attendant de rencontrer un noeud vide dans la descente pour le remplacer par un noeud contenant x.

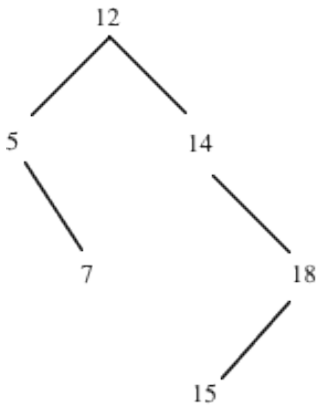
```
Fonction appartient (E x : élément, A : arbrebin) : booléen
Var app : booléen
Début
Si arbre-vide(A)
     Alors app ← faux
     Sinon Si x=val(racine(A))
               Alors app ← vrai
               Sinon Si x<val(racine(A))
                         Alors app ← appartient(x,fils-gauche(A))
                         Sinon app ← appartient(x,fils-droit(A))
                     FinSi
           FinSi
FinSi
Retourner(app)
Fin

Procédure insérer (E x : élément, E/S A : arbrebin)
Début
Si arbre-vide(A)
     Alors affecter-val(racine(A),x)
           fils-gauche(A) ← créer-arbre-vide
           fils-droit(A) ← créer-arbre-vide
     Sinon Si x<val(racine(A))
               Alors insérer(x,fils-gauche(A))
               Sinon Si x>val(racine(A))
                         Alors insérer(x,fils-droit(A))
                     FinSi
           FinSi
FinSi
Fin
```

La suppression pose quelques petits problèmes.

Exemple :





Il faut chercher le plus petit élément de l'arbre plus grand que l'élément à supprimer (ici, on supprime 10 donc la ppe est 12). Ensuite, on remplace l'élément à supprimer par le ppe. Besoin d'une procédure supprimin qui retire le ppe et retourne sa valeur.

```
Procédure supprimin (E/S A : arbrebin, S e : élément)
Début
Si arbre-vide(fils-gauche(A))
     Alors e ← val(racine(A))
           A ← fils-droit(A)
     Sinon supprimin(fils-gauche(A),e)
FinSi
Fin

Procédure supprimer (E x : élément, E/S A : arbrebin)
Début
Si ¬arbre-vide(A)
   Alors Si x<val(racine(A))
            Alors supprimer(x,fils-gauche(A))
            Sinon Si x>val(racine(A))
                     Alors supprimer(x,fils-droit(A))
                     Sinon Si arbre-vide(fils-gauche(A))
                                 et arbre-vide(fils-droit(A))  {feuille(A)}
                              Alors A←créer-arbre-vide
                              Sinon Si arbre-vide(fils-gauche(A))
                                       Alors A←fils-droit(A)
                                       Sinon Si arbre-vide(fils-droit(A))
                                                Alors A←fils-gauche(A)
                                                Sinon supprimin(fils-droit(A),e)
                                                      affecter-val(racine(A,e)
                                             FinSi
                                    FinSi
                           FinSi
                  FinSi
        FinSi
FinSi
Fin
```