Fascicule du Cours :
Algorithmique et Structures de Données

Fiche module

Niveau ciblé	1ère année du cycle ingénieur
Nature de l'enseignement	Cours intégré
Volume horaire	90 heures
Charge d'enseignement hebdomadaire	6 heures/semaine

Description du module

Ce module est constitué de deux volets : un premier volet qui s'intéresse à l'algorithmique et un deuxième qui se focalise sur la programmation. Le premier volet est lui-même constitué de deux parties. La première couvre l'algorithmique de base et est enseignée en parallèle avec la partie programmation. La deuxième quant à elle s'intéresse à des notions d'algorithmique plus avancées englobant les structures de données séquentielles et arborescentes.

Objectifs du module

Les objectifs du volet Algorithmique de ce module sont essentiellement les suivants :

- Expliquer aux étudiants l'importance d'avoir des connaissances solides en algorithmique, le lien entre l'algorithmique et la programmation ainsi que les règles d'écriture d'un algorithme ;
- Etudier, assimiler et maîtriser les notions de base nécessaires pour la résolution de tout problème d'algorithmique ;
- Introduire et expliquer la notion de Types Abstraits de Données en insistant sur leur intérêt ainsi que la manière de les définir ;
- Utiliser la démarche de définition de TAD pour introduire les types : « Liste », « Pile » et « File » permettant l'organisation séquentielle de données aussi bien que les « Arbres Binaires » et les « Arbres Binaires de Recherche » permettant l'organisation arborescente de données ;
- Proposer, comparer et mettre en œuvre les différentes représentations permettant d'implémenter les TAD déjà introduits.

Les objectifs du volet Programmation de ce module sont essentiellement les suivants :

- Expliquer brièvement l'histoire du langage C et souligner ses caractéristiques principales en tant que langage compilé ;
- Introduire de façon progressive les notions de base de la programmation C ;
- Insister sur les notions délicates du langage telles que les pointeurs et le passage par adresse ;
- Apprendre aux étudiants à programmer en C en organisant des séances de TP et en leur proposant des travaux à rendre ;
- Créer des bibliothèques implémentant les TAD vus dans le volet Algorithmique.

Evaluation du module

L’évaluation est constituée d’un ou de plusieurs tests, de travaux pratiques à rendre, d’un devoir surveillé et d’un examen.

Sommaire

Références bibliographiques du cours ASD

MALGOUYRES, Rémy, ZROUR, Rita, et FESCHET, Fabien. *Initiation à l'algorithmique et à la programmation en C-3e éd. : Cours avec 129 exercices corrigés*. Dunod, 2014.

CORMEN, Thomas H., LEISERSON, Charles Eric, RIVEST, Ronald L., et al. *Introduction à l'algorithmique*. Dunod, 1997.

COURTIN, Jacques, KOWARSKI, Irène. *Initiation à l'algorithmique et aux structures de données Volume 2*. Dunod, 1997.

COURTIN, Jacques, KOWARSKI, Irène. *Initiation à l'algorithmique et aux structures de données Volume 1*. Dunod, 1994.

FROIDEVAUX, Christine, GAUDEL, Marie-Claude, et SORIA, Michele. *Types de données et algorithmes*. Ediscience International, 1993.

BEN RHOUMA, Kamel. *Cours Algorithmique et Structures de Données*. ENSI, 2007.

ALGORITHMIQUE DE BASE

CHAPITRE I - LES OBJETS SIMPLES ET LES OPERATIONS DE BASE

1. Définition

L'origine du mot *algorithme* découle du nom latinisé du mathématicien perse du 9ème siècle « Abu Ja'far Mohammed Ibn Mûsâ Al-Khowâ-rismî ».

Le terme algorithme admet, depuis longtemps, plusieurs définitions qui convergent. Froidevaux et al. définissent ce concept comme étant **[Froidevaux et al., 1993]** :

« La composition d'un **ensemble fini d'étapes**, chaque étape étant formée d'un nombre **fini d'opérations** dont chacune est :

- Définie de façon **rigoureuse et non ambigüe ;**
- **Effective** c'est-à-dire pouvant être réalisée par une machine. »

Tout algorithme est donc caractérisé par **[Courtin et al., 1994]** :

- Un ensemble d'opérations à exécuter.
- Un ordre d'exécution de ces différentes opérations dicté par la logique d'enchaînement et conditionné par les structures mises en œuvre.
- Un début et une fin.

Le domaine qui étudie les *algorithmes* est désigné par : « algorithmique ».

2. Structure d'un algorithme

Algorithme <nom de l'algorithme>

CONST

<nom_de_la_constante = valeur>

TYPE

<liste des structures>

VAR

<nom-variable> : <type-variable>

FONC

<liste des fonctions >

PROC

<liste des procédures>

DEBUT

<corps de l'algorithme : ensemble des instructions – une instruction par ligne>

FIN

Dans ce qui suit, nous présentons un exemple d'algorithme :

Supposons qu'on veut calculer la moyenne des notes d'un élève dans une matière donnée et qu'on dispose de 3 notes : N1, N2 et N3.

```
Algorithme Moyenne
CONST
c1=2
c2=3
c3=4
VAR
        Moyenne, N1, N2, N3 : réel
DEBUT
        Ecrire(″ Donnez les trois notes dans l'ordre″)
Lire (N1,N2,N3)
        Moyenne ← (N1*c1+N2*c2+N3*c3)/3
        Ecrire (″ La moyenne est : ″, Moyenne)
FIN
```

3. Types de données

Dès que l'on a besoin de stocker une information au cours d'un algorithme, on utilise une **variable**.

a. Types de base

i. Booléen

Valeurs : Vrai, Faux.

Opérations

* Opérations relationnelles (Faux < Vrai)
* Opérations logiques : négation (non), conjonction (et), disjonction (ou)

A	B	Non A	A et B	A ou B
V	V	F	V	V
V	F	F	F	V
F	V	V	F	V
F	F	V	F	F

ii. Caractère

Valeurs : c'est un ensemble fini de symboles totalement ordonnés. Un code ASCII est associé à chaque caractère.

Opérations

- Opérations relationnelles par exemple : 'a'<'b' ; 'a'>'A'
- Concaténation

iii. Entier

Valeurs : *c*'est un sous ensemble de Z. Il s'agit d'une représentation en machine des entiers relatifs.

Opérations

- Opérations relationnelles : = ; < ; > ; ≤ ; ≥ ; < >
- Opérations arithmétiques : +, -, *, /
- DIV : division entière
- MOD : reste de la division entière
- PUISSANCE

iv. Réel

Valeurs : un sous ensemble de R. Il s'agit d'une représentation en machine des réels.

Opérations

- Opérations relationnelles : = ; < ; > ; ≤ ; ≥ ; < >
- Opérations arithmétiques : +, -, *, /
- Puissance
- Autres opérations comme cos, sin…

Outre les types définis ci-dessus ; il est possible de définir de nouveaux types comme suit.

b. Type scalaire énuméré

Le type scalaire énuméré permet de représenter un ensemble ordonné et fini de valeurs désignées par des identificateurs.

Définition et déclaration

Type nom_type = (val_1,…, val_k)

Une variable de type énuméré prendra une des valeurs figurant entre parenthèses.

Exemple : Type t_couleur = (blanc, bleu, rouge)

Var

cl : t_couleur

cl ← blanc

Opérations

- Opérations relationnelles
- Ord( ) : rang dans le domaine.
- Succ( ) : successeur dans le domaine
- Pred( ) : prédécesseur dans le domaine
- Inc (v,n) : incrémenter v de n

Exemples : ord(bleu)=1, succ(bleu)=rouge, pred(bleu)= blanc.

c. Type intervalle

Le type intervalle permet de restreindre le groupe de valeurs d'un type scalaire (entier, réel, booléen, caractère ou type énuméré).

Définition du type :

Type nom_intervalle = borne_inf…borne_sup

Avec borne_inf < borne_sup.

Les opérations que nous pouvons effectuer sont celles du type des variables contenues dans l'intervalle.

Exp : Type T_chiffre=0..9

T_majuscule='A'..'Z'

4. Opérations de base

Dans un schéma séquentiel, on distingue deux types d'instructions :

- Des instructions simples, élémentaires permettant de faire un traitement ;
- Des instructions permettant de commander le déroulement de l'exécution, appelées structures de contrôle.

a. Affectation

Le rôle de cette instruction est de placer une valeur dans une variable (associer une valeur à une variable).

<nom variable> ← <expression>

<expression> : constante | variable | expression

La variable et l'expression doivent être de même type ou de types compatibles (Exp : il est possible d'affecter un entier à une variable réelle).

Exemple

Test()

var

i,j : entier

Debut

i ← 10

j ← 20

i ← 2*j

Fin

b. Lecture

L'utilisateur introduit des valeurs qui seront affectées à des variables réservées par le programmeur. L'utilisateur doit respecter le type des éléments à introduire.

Lire (< liste variables >)

< liste variables > : <variable1>, … <variable n>

Exemple

Var

i : entier

r : réel

Debut

Lire(i,j)

Fin

c. Ecriture

Cette instruction permet aussi bien l'affichage d'un texte que l'affichage des valeurs d'une liste de variables (une ou plusieurs).

Ecrire (<texte>)

Exemple

Var

i, j : entier

Debut

Ecrire ("Bonjour")

Lire (i, j)

i ← i*j

Ecrire (i)

Fin

Remarque

On peut regrouper dans la syntaxe de l'instruction Ecrire, la liste des variables et le texte. Par exemple : Ecrire ("La valeur de i est", i, " la valeur de j est ", j)

5. Algorithmique et programmation

La mise en œuvre d'un algorithme consiste en l'écriture des opérations qui le composent dans un langage de programmation et constitue alors la brique de base d'un programme informatique. Un algorithme exprime les instructions permettant de résoudre un problème donné indépendamment des langages de programmation. Lors du passage de l'algorithmique à la programmation, le travail revient à résoudre des problèmes de syntaxe, ou de types d'instructions propres à ce langage.

Afin d'écrire un algorithme, on se base généralement sur une série de conventions désignée par « **pseudo-code** », qui est comparable à un langage de programmation dont on aurait évacué la plupart des problèmes de syntaxe. Puisqu'il est purement conventionnel, ce pseudo-code peut varier légèrement d'une référence à une autre.

Pour être compréhensible, un algorithme doit être présenté de manière lisible ; il ne doit pas être trop long, ce qui nécessite parfois de le décomposer en modules. Certaines formes de programmation telles que les branchements sont à éviter ; par contre la récursivité permet une expression concise des algorithmes et facilite leur analyse.

CHAPITRE II - LES STRUCTURES DE CONTROLE

1. Structures conditionnelles

a. Schéma conditionnel à simple choix (Si – Alors – Fsi)

i. Syntaxe

Si <condition> alors

<Traitement>

Fsi

<condition> : expression à résultat booléen

<traitement> : suite d'instructions

Fsi : Fin si

ii. Sémantique

Si <**condition**> est vraie alors on exécute le bloc <**traitement**> et on continue en séquence à l'instruction qui suit <**Fsi**>.

Si <**condition**> est fausse, on continue en séquence avec l'instruction qui suit immédiatement <**Fsi**> .

b. Schéma conditionnel à double choix (Si – Alors – Sinon - Fsi)

i. Syntaxe

Si <condition> alors

<Traitement 1>

Sinon

<Traitement 2>

Fsi

ii. Sémantique

Si <**condition**> est vraie alors on exécute le bloc <**traitement 1**> et on continue en séquence à l'instruction qui suit <**Fsi**>.

Si <**condition**> est fausse, on exécute les instructions du bloc <**traitement 2**> et on passe à l'instruction qui suit immédiatement <**Fsi**>.

c. Schéma conditionnel à choix multiple

i. Syntaxe

Selon <v>

<v1> : <Traitement 1>

<v2> : < Traitement 2>

<vn> : < Traitement n>

Sinon : <Traitement >

Fin selon

ii. Sémantique

<Traitement > : suite d'instructions

<v> : identificateur d'une variable de type scalaire.

<v_i> : une valeur ou une liste composée d'une suite de valeurs (**Exp** : 2, 5, 8) ou d'une fourchette (**Exp :** 0..9). Les valeurs contenues dans les <v_i> doivent être de même type que <v>.

Lorsque <v> = <v_i> on exécute le traitement correspondant et on passe à l'instruction qui suit immédiatement **Finselon**.

Le bloc <**sinon**> est facultatif.

Remarque

L'instruction selon peut être ramenée à un enchaînement de « Si » imbriquées.

Exemple d'utilisation

```
Exp( )
Var
        k, x : entier
Debut
x ← 0
lire(k)
Selon k
        5 : x ← x+6
        6..10 : x ← x+18
        11, 12, 13 : x ← x-5
Sinon
        x ← 1
Fin selon
Fin
```

2. Structures itératives

Les structures itératives sont des instructions qui permettent de répéter l'exécution d'une même séquence d'instructions un nombre fini de fois.

On distingue deux types de répétitions :

* L'itération : quand on connait le nombre de fois que l'on va répéter une instruction ou une séquence d'instructions.
* La répétition conditionnelle : quand la poursuite d'une répétition dépend d'une condition à résultat booléen.

a. Boucle « Pour »

Pour <compteur> **de** <valeur initiale> **à** <valeur finale> **pas** <incrément> **faire**

    <Traitement>

Fpour

<compteur> : l'identificateur d'une variable de type scalaire.

<valeur initiale>, <valeur finale>, <incrément> : une variable de même type que le compteur ou une constante de même type que le compteur ou une expression dont le résultat est de même type que le compteur.

<Traitement> : une ou plusieurs instructions quelconques.

Exemple :

```
var
    i, j, n : entier
Debut
    n ← 5
    j ← 1
    Pour i de j à n pas (2) faire
        Ecrire(i)
    Fpour
Fin
```

Remarques

Il est interdit de modifier : <compteur>, <valeur initiale>, <valeur finale> et <incrément>.

* <valeur initiale> <= <valeur finale> Ssi <incrément> >0
* <valeur initiale> >= <valeur finale> Ssi <incrément> <0
* <incrément> ne doit jamais être nul.

Lorsque la valeur de <incrément> n'est pas précisée elle est considérée comme étant égale à 1.

L'instruction Pour est utilisée lorsque le nombre d'itérations est connu.

b. Boucle « Tant que »

Tant que <condition> **faire**

    <Traitement>

Fin tant que

<condition> : expression à résultat booléen

<**Traitement**> : un ensemble d'instructions telles qu'il existe au moins une instruction modifiant la condition.

Exercice

Ecrire un algorithme permettant de saisir une suite d'entiers positifs ou nuls se terminant par une valeur négative, de faire la somme de ces entiers et de l'afficher.

c. Boucle « Répéter »

Répéter

<Traitement>

Jusqu'à <condition>

Remarque

Cette instruction ne peut être utilisée que si on est sûrs que le traitement est exécuté **au moins une fois**.

La boucle précédente peut être remplacée par la boucle suivante :

Tant que non <condition> faire

<Traitement>

FTQ

CHAPITRE III – LES OBJETS STRUCTURÉS

Partie I - Les tableaux

Un tableau est une structure de données homogène contenant des éléments de même type.

1. Tableaux à une dimension

a. Définition de type

<Nom type> = tableau [b_i..b_s] de <type de base>

<Nom type> : identificateur associé au tableau
b_i..b_s : intervalle sur N qui fixe les indices des éléments du tableau (avec $b_i <= b_s$).
b_i : borne inférieure, b_s : borne supérieure
<type de base> : le type des éléments du tableau.

Exemple
Vecteur = tableau [1..7] de réel

b. Définition de variables

<identificateur> : <type tableau>
<type tableau> : <Nom type> ou tableau [b_i..b_s] de <type de base>

Exemple
T : Vecteur (on utilise cette déclaration dans le cas où on a plusieurs tableaux de même type)
T : tableau[1..7] de réel

c. Opérations

* Accès aux éléments d'un tableau

<Nom tableau> [<indice>]
Où <Nom tableau> est l'identificateur du tableau et <indice> est la position de l'élément dans le tableau.

Exemple :
T[1] désigne l'élément du tableau à la position 1.

* Les opérations autorisées sont toutes les opérations autorisées sur le type des éléments (type de base).

Exemple :

T : tableau [1..15] de entier

T[5] ← T[20]+3 faux

T[6] ← '4' faux

T[10] ← 2* T[3] correct

d. Exercice

Ecrire un algorithme qui applique le principe de la recherche dichotomique pour chercher un élément x dans un tableau trié de N éléments.

2. Tableaux à deux dimensions

a. Définition de type

<Nom-type>= tableau [<int-ligne>, <int-colonne>] de <type de base>

<Nom-type> : identificateur associé au tableau

<int-ligne> : intervalle de lignes

<int-colonnes> : intervalle de colonnes

<type de base> : le type des éléments du tableau

Exemple :

MatriceEntier = tableau [1..10, 1..15] de entier

b. Définition de variables

<identificateur > : <type -tableau>

<type-tableau> : <Nom-type> ou tableau [<int-ligne>, <int-colonne>] de <type de base>

Exemple :

MatriceEntier=tableau [1..10,1..15] de entier (Nom-type)

T : MatriceEntier

Ou

T : tableau[1..10, 1..15] de entier

c. Opérations

Accès aux éléments de la matrice

<identificateur-tableau> [<indice-ligne>, <indice-colonne>]

Exemple :

T[1,15] : l'élément se trouvant à l'intersection de la ligne 1 et la colonne 15.

T[i,j] : l'élément se trouvant à l'intersection de la ligne i et la colonne j.

Partie II - Les chaînes de caractères

1. Déclaration, initialisation et accès aux éléments

Le type chaîne permet de décrire des objets formés par une succession de N caractères avec N compris entre 0 et 255. Si N=0 on dit que la chaîne est vide.

Dans la plupart des langages de programmation, il existe des "outils" pour manipuler les chaînes de caractères. Au niveau des algorithmes, nous introduisons quelques fonctions prédéfinies qui correspondent aux "outils" classiquement fournis par les langages. Cette liste partielle peut bien sûr être complétée en fonction de vos besoins.

- ***Déclaration d'une chaine de caractères***

Comme indiqué dans la définition précédente, une chaine de caractères est composée d'au plus 255 caractères donc il n'est pas nécessaire (mais il est possible) de faire figurer dans la déclaration d'une chaine le nombre maximum de caractères qu'elle peut contenir.

Exp :

Nom : Chaine [10] ***//la chaine nom peut contenir au maximum 10 caractères***

Prenom : Chaine ***// la chaine nom peut contenir au maximum 255 caractères***

- ***Accès à un élément de la chaine***

Ch ← 'Bonjour'

Ch[1]='B' Ch[4]='j'

2. Lecture et écriture d'une chaîne de caractères

- ***Lecture***

Ch : chaine

Lire (ch) //la chaine est manipulée comme une entité

- ***Ecriture***

Ch : chaine

Ecrire (ch)

3. Fonctions de base sur les chaînes de caractères

- ***Fonction longueur (ch : chaîne) : entier***

Retourne la longueur de la chaîne *ch* c'est-à-dire le nombre de caractères dont elle est constituée.

Exp : ch ← 'Bonjour'

Longueur (ch)=7

- ***Fonction concat (ch1:chaîne, ch2:chaîne) : chaîne***

Retourne une chaîne de caractères formée par la concaténation de *ch1* et de *ch2*. La chaîne résultat est formée de *ch1* suivi de *ch2*.

Exp : ch1 ← 'Bon', ch2 ←'jour', ch3 ←concat(ch1,ch2)

Ch3 contient 'Bonjour'

- ***Fonction sous_chaîne (ch:chaîne, i:entier, l:entier) : chaîne***

Retourne une sous-chaîne de longueur *l* extraite de la chaîne *ch*, à partir de la position *i*.

Exp : sousChaîne("informatique", 6, 2) retourne la chaîne "ma".

- ***Fonction Pos (ch1:chaîne, ch2:chaîne) : entier***

Retourne la première position de la chaîne ch1 dans la chaîne ch2. Si ch1 n'est pas dans ch2, elle retourne 0. **Exp :** Pos ('jour', 'Bonjour') donne 4

- ***Procédure Efface (ch : chaîne, p : entier, n :entier)***

Efface n caractères de ch à partir de la position p.

Exp : Efface ('CD ROM',3 ,4) la chaine devient 'CD'.

- ***Procédure Insère (ch1 : chaîne, ch2 : chaîne, p : entier)***

Insère la chaîne ch1 dans la chaîne ch2 à partir de la position p. Le caractère numéro p et les suivants sont décalés vers la droite.

Exp : ch1←'DE', ch2 ←'ABCFG'

Insère (ch1, ch2, 4) ch2 devient 'ABCDEFG'

- ***Procédure Convch (d : reel, ch : chaîne)***

Convertit un nombre en une chaine de caractères et l'affecte à ch.

Exp : Convch (2007, ch) ch contient la chaîne '2007'

- ***Procédure Valeur (ch : chaîne, d : reel, erreur : entier)***

Convertit une chaîne ch en une valeur numérique décimale et l'affecte à la variable d. Le paramètre erreur contient 0 si la conversion s'est déroulée sans erreur, sinon elle contiendra le numéro du caractère qui a déclenché l'erreur.

Exp :

Ch ← '10.5'

Valeur (ch,d,e) d contient le nombre 10.5 et e contient 0

Ch ← '01/01/2008'

Valeur (ch,d,e) d contient 0 et e contient 3 (le caractère '/' n'est pas un chiffre).

Partie III - Les enregistrements

1. Définition

Un enregistrement ou article est une structure de données composée d'un nombre fixe d'éléments pouvant être de types différents. Il permet de regrouper dans un même type un ensemble d'informations caractérisant un objet donné. Les éléments d'un article sont appelés « champs » et peuvent être à leur tour des structures de données.

2. Syntaxe

a. Déclaration de type

<Nom-type>=**struct**

<champ 1> : <type1>

<champ 2> : <type2>

…

<champ n> : <type n>

Fin struct

<Nom-type> : identificateur associé à l'article

<Champ i> : identificateur du ième champ

<type i> : le type du ième champ

Exemple

Rectangle= **struct**

Largeur, Longueur : réel

Fin struct

Personne = **struct**

Nom : chaîne de caractères

Age : entier

Fin struct

b. Déclaration des variables

<id-var> : <Nom type article>

Exemple : R1,R2 : Rectangle

P1 : Personne

c. Opération d'accès

<id-var>. <nom-champ>

Exemple :

R1.largeur ← 3

R1.longueur ← 5

P1.nom ← « Mohamed »

P1.age ← 10

3. Exercice

Ecrire un algorithme qui lit une suite d'informations concernant 30 étudiants et les stocke dans une structure de votre choix.

Chaque étudiant est caractérisé par un numéro de cin, un nom, une date de naissance (jour, mois, année) et un niveau d'études (1, 2 ou 3).

Ecrire un algorithme permettant de :

- **a.** Calculer le pourcentage des étudiants au niveau 1.
- **b.** Afficher la liste des noms des étudiants dont l'année de naissance >=1990.
- **c.** Afficher toutes les informations concernant un étudiant dont le numéro de cin est donné.

Solution

```
Date=struct
        Jour : 1..31
        Mois : 1..12
        An : entier
    Fin struct
Etudiant =struct
        Num : entier
        Nom : chaine
        D : Date
        Niveau : entier

    Fin struct
```

```
Algo statistiques ( )
Ens-Etd=tableau[1..100] de Etudiant

Var
    T : Ens-Etd
    n : entier
```

```
Algorithme Lire-données ()
Cst
N=30
Var
    num ,i, nb , k : entier
```

```
Debut
        nb ← 0
    Pour i de 1 à N faire
        Lire(num)
        Lire(T[i].nom)
        Lire(T[i].d.jour)
        Lire(T[i].d.mois)
        Lire(T[i].d.an)
        Lire(T[i].niv)
         Si (T[i].niv=1) alors
            nb←nb +1
         Fsi
    Fpour
    Ecrire (« le pourcentage des étudiants au niveau 1 est : », nb*100/N)

    Pour i de 1 à N faire
        Si T[i]>=1990 alors
            Ecrire(T[i].nom)
        Fsi
    Fpour
    k ← 0
    Ecrire (« donner le numero de l'etudiant cherché »)
    Lire(num)
    Tr ← faux
    i ← 1
    Tant que i<=N et non Tr Faire
        Si T[i].num=num alors
            Tr ← vrai
            Ecrire (« nom : », T[i].nom)
            Ecrire (« date de naissance »,T[i].d.jour,"/ ",T[i].d.mois,"/ ", T[i].d.an)
            Ecrire (« niveau : », T[i].niv)
        Sinon
            i ← i +1
        Fsi
    FTQ
Fin
```

CHAPITRE IV – LA CONSTRUCTION MODULAIRE D'ALGORITHMES

1. Introduction

Lors de la conception d'un programme résolvant un problème général, il est nécessaire de décomposer le problème en différents sous-problèmes moins complexes à résoudre.

Ces différents sous-problèmes peuvent être résolus grâce à des modules ou sous-programmes.

L'utilisation des sous-programmes présente de nombreux avantages. Nous citons :

- La possibilité de réutilisation ;
- La facilité de la résolution de problèmes en ne s'intéressant qu'à l'écriture d'un module à la fois ;
- La factorisation du code : les sous-programmes permettent de ne pas répéter plusieurs fois une même séquence d'instructions au sein de l'algorithme.

En algorithmique, il existe deux types de sous-programmes :

- Les procédures
- Les fonctions

Lors de la conception d'un programme deux aspects apparaissent :

- **La définition de la procédure ou de la fonction ;**
- **L'appel de la procédure ou de la fonction dans le programme.**

Une fois explicité (l'algorithme défini), un sous programme peut être utilisé dans d'autres programmes ou sous-programmes. Le (sous-) programme qui utilise un sous-programme est désigné par (sous-) programme **appelant**.

2. Procédures

a. Déclaration

Procédure <Nom procédure> (<Liste-paramètres>)

var

<variables-locales>

Début

<Traitement>

Fin

<Nom procédure> : identificateur

<Liste-paramètres> : <par1> : <Type1>, <par2> :<Type2>,……,<par n> : <Type n>

<par i> : identificateurs de variables

<variables-locales> : <var1> : <Type1>

<var2> : <Type2>

Une variable définie dans un sous-programme est appelée **variable locale.** La portée d'une variable locale est uniquement le sous-programme qui la déclare.

<Traitement> : liste des instructions de la procédure

b. Appel de procédure

Un appel de procédure se fait de la façon suivante :

<Nom procédure> (<Liste-paramètres>)

<Liste-paramètres> : <par1>, <par2>,……,<par n>

<par i> =identificateurs de variables ou constante ou expression.

c. Exemple de procédure

Algo Exp1

Var

A, B : entier

Procédure **AfficheMax**(X :entier, Y :entier)

Début

si (X>Y)

alors ecrire(X)

sinon

ecrire (Y)

fsi

Fin

Début (Algorithme principal : algorithme appelant)

Ecrire(« Donnez deux entiers »)

Lire(A,B)

AfficheMax(A,B)

Fin

3. Fonctions

a. Déclaration

Fonction <Nom fonction> (<Liste-paramètres>) : <type>

var

<variables-locales>

Début

<Traitement>

<Nom fonction> ⟵ **<variable>**

Fin

<Nom fonction> : identificateur

<Liste-paramètres> : <par1> : <Type1>, <par2> :<Type2>,……,<par n> : <Type n>

<par i> : identificateurs de variables

<type> : type de retour de la fonction

<variables-locales> : <vari> : <Typei>

<Traitement> : liste des instructions de la fonction

<variable> : variable de même type que la fonction ou de type compatible.

b. Appel de fonction

Une fonction est utilisée dans une instruction :

- Dans la partie droite d'une affectation (y ⟵ f(x))
- Dans une expression (arithmétique ou logique selon le type de la fonction) (y ⟵f(x)+2)
- Comme paramètre d'une procédure ou d'une fonction P(f(x)), f(f(x)).

c. Exemple de fonction

Algo Max

Var

A, B, C : entier

Fonction **Max(X, Y : entier) : entier**

Var

W : entier

Début

Si (X>Y) alors

W ⟵ X

Sinon

W ⟵ Y

Fsi

Max ⟵ W

Fin

Début (Algorithme principal : algorithme appelant)

Ecrire (« Donner deux entiers »)

Lire (A, B)

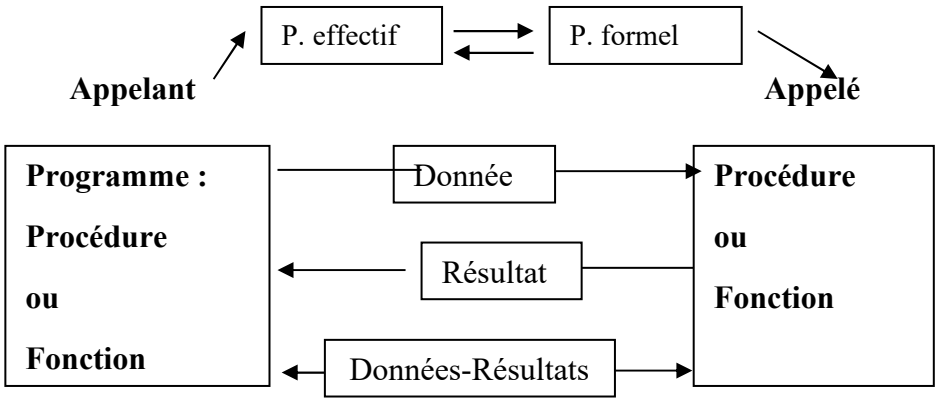
C ⟵ Max(A, B)

Ecrire (« Le maximum est : », C)

Fin

4. Transmission des données

a. Les types de paramètres



Lors de l'utilisation des sous-programmes, une transmission de données et de résultats est mise en jeu. On distingue :

- **Les paramètres formels :** utilisés dans la **définition** de la fonction ou procédure. Ce sont des variables qui peuvent prendre plusieurs valeurs.
- **Les paramètres effectifs :** ce sont les paramètres utilisés de façon effective dans un appel de fonction ou de procédure (*ce sont les valeurs qu'on utilisera pour effectuer les calculs dans la fonction ou procédure*)

Le nombre de paramètres **effectifs** doit être égal au nombre de paramètres **formels**.

Le paramètre **effectif** doit avoir **le même type ou un type compatible** avec le paramètre **formel** correspondant.

La correspondance entre paramètres **effectifs** et paramètres **formels** se fait dans l'ordre de leur apparition dans les deux listes.

b. **Les modes de passage de paramètres**

Il existe 3 modes de passage de paramètres :

- D**onnée** : dans ce mode de passage, le paramètre doit porter une valeur significative avant l'exécution, il est nécessaire aux calculs effectués dans le sous-programme mais il ne sera pas modifié par celui-ci. C'est le seul mode de passage de paramètre qui admet l'utilisation d'une constante. Ce mode de passage est exprimé par le mot clé : **Don**
- **Résultat** : dans ce mode de passage, le paramètre est destiné à recevoir un résultat. Les instructions du sous-programme affectent obligatoirement une valeur à ce paramètre qui n'a pas nécessairement une valeur significative avant l'exécution. Par contre, après l'exécution du sous-programme, il doit comporter une valeur significative. Ce mode de passage de paramètre est exprimé par le mot clé : **Res**
- D**onnée-résultat** : dans ce mode de passage, le paramètre est utilisé dans les calculs faits dans le sous-programme. Il est en même temps utilisé pour recevoir un résultat. Il doit donc avoir une valeur significative avant et après exécution. Ce mode de passage de paramètre est exprimé par le mot clé : **DonRes**

Exemples :

1/

Procédure **AfficheMax** (X :entier, Y :entier)

Utilisation : AfficheMax (A,B)

Les paramètres **formels** sont : X et Y.

Les paramètres **effectifs** sont : A et B.

X et Y sont des paramètres données. Donc on déclarera la procédure AfficheMax comme suit :

Procédure **AfficheMax** (**Don** X , Y :entier)

2/

Fonction **Max(X, Y : entier) : entier**

Utilisation : C ⟵ Max(A, B)

Les paramètres **formels** sont : X et Y.

Les paramètres **effectifs** sont : A et B.

X et Y sont des paramètres données. Donc on déclarera la fonction **Max comme suit :**

Fonction Max(Don X, Y : entier) : entier

3/

Soit la procédure **Permute** qui permute deux entiers X et Y.

Procédure **Permute** (DonRes X, Y :entier)

5. Récursivité

a. Définition et exemples

La récursivité est la possibilité de faire figurer dans la définition d'un objet une référence à l'objet lui-même. Une notation est récursive si elle est partiellement définie à partir d'elle-même.

Il existe deux types de récursivités :

- **La récursivité des objets :** la définition d'un objet est récursive si l'objet apparaît dans cette définition.
  Exemple : une chaîne de caractères est soit la chaîne vide soit un caractère suivi d'une chaîne de caractères.
- **La récursivité des traitements :** un sous-programme est dit récursif s'il fait appel à lui-même pour traiter un problème. Dans ce cas, le problème en question est réduit à un problème plus simple mais de même nature. Il existe deux types de récursivité des traitements : la récursivité **directe** et la récursivité **indirecte**.

Un problème se prête bien à la récursivité lorsqu'il peut se décomposer en sous problèmes de même type.

Récursivité directe : A s'appelle elle-même

```
Module A( )
Debut
        …
        A( )
        …
Fin
```

Récursivité indirecte : A appelle B qui appelle A

```
Module A( )
Debut
        …
        B( )
        …
Fin
Module B( )
Debut
        …
        A( )
        …
Fin
```

Exemple : La fonction Factorielle

n != n*(n-1)

0 !=1

b. Propriétés des algorithmes récursifs

Tout module récursif doit contenir une clause conditionnelle telle que l'évaluation puisse dans certains cas se faire sans récursivité.

Module A( )

Debut

Si B alors C

Sinon f(A( ))

Fsi

Fin

B : est la condition d'arrêt.

C : cas particulier de la récursivité

F(A( )) : une fonction de A( ) qui doit permettre le changement de la valeur de la condition B à un moment donné.

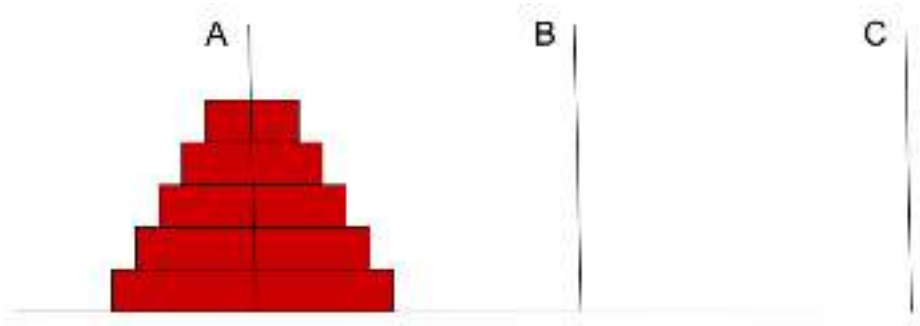
c. Exercices et réflexion

Ecrire la version récursive de la fonction Factorielle. Donner la trace d'exécution de cette fonction pour le paramètre 5.

Ecrire une fonction récursive qui, étant donné un entier X positif et un entier >=0, permet de calculer X^Y.

Problème des tours de Hanoi :

Etant données 3 tours A, B et C ; le problème consiste à transférer n disques de A à C, en utilisant B, de sorte que jamais un disque ne repose sur un disque de plus petit diamètre.



Afin de déplacer n disques de A vers C :

Si n vaut 1, on déplace le disque

Sinon

On déplace d'abord n-1 disques de A vers B

On déplace le disque numéro n de A à C

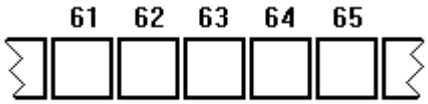
On déplace n-1 disques de B à C

CHAPITRE V – LES POINTEURS

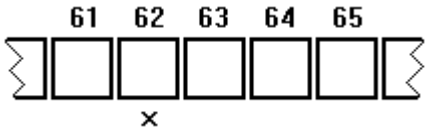
1. Présentation du concept

Les variables sont stockées dans une mémoire statique : on peut **représenter cette mémoire par des cases numérotées dans l'ordre croissant**. Le numéro de la case est appelé « adresse ». Dans ce qui suit, et pour faciliter les choses, on considérera que toute variable utilise une seule case, même si ce n'est pas vraiment conforme à la réalité. En effet, la quantité de mémoire prise par une variable dépend de son type. Il est utile de rappeler, dans ce contexte, que dans une machine **8 bits**, les adresses sont codées sur un octet, ce qui nous fait **256 cases numérotées de 0 à 255**.

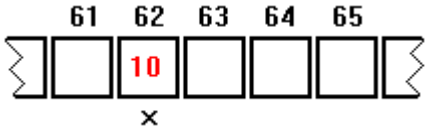
soit le schéma suivant :



Quand on écrit : **x : entier**



On réserve une case pour la variable x dans la mémoire, case numéro 62 dans le cas du schéma.
Quand on écrit : **x <- 10**



On écrit la valeur 10 dans l'emplacement réservé pour x. Dans le schéma ci-dessus, cette valeur est située à la case 62.

Quand on exécute un programme P, le système d'exploitation se charge d'allouer l'espace mémoire nécessaire pour le stockage de toutes les variables déclarées dans P. Dans le cas où une variable déclarée ne sera pas utilisée ou qu'elle ne sera pas utile pendant toute la période d'exécution du programme, il y a un **gaspillage** de la mémoire. En effet, il y a deux types d'allocation mémoire : allocation statique (gérée automatiquement par le système) et dynamique (gérée manuellement par le programmeur).

L'allocation dynamique nous permet d'éviter ce type de gaspillage de mémoire et aussi de gérer les ensembles dynamiques (on verra cela dans le cours listes chaînées) et ce grâce aux pointeurs.

2. Les pointeurs

Un pointeur est une variable destinée à contenir une adresse mémoire, c'est à dire une valeur identifiant un emplacement en mémoire.

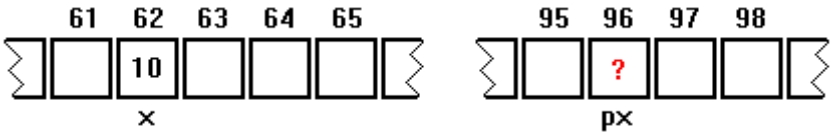
La déclaration d'une variable pointeur réserve uniquement le nombre d'octets nécessaires au codage de l'adresse mémoire de la donnée pointée mais ne réserve pas de la mémoire pour celle-ci (la donnée pointée). Quel que soit le type de la donnée pointée, la taille mémoire du pointeur est toujours la même.

a. Déclaration d'un pointeur

id_var_pointeur : ^ type de base

Exemple :

px : ^ entier



On réserve un emplacement en mémoire pour le pointeur px (case numéro 96 dans le cas du schéma).

b. Réservation de la mémoire

Allouer (id_var_pointeur)

Exemple :

px : ^entier

Allouer (px) ⇔ z : entier et px <- adresse(z) // px pointe vers z et z est la variable pointée par px.

A la création deux choses se produisent :

- De la mémoire est réservée pour stocker une donnée du type spécifié (entier pour px).
- La variable concernée (px) contient l'adresse mémoire de cet espace réservé (représenté par une flèche).
- px contient l'adresse de l'espace réservé, nous pourrons alors utiliser cette variable.

c. Accès à la variable pointée

On accède à la valeur pointée comme suit :

id_var_pointeur ^

Exemple :

px : ^entier

px^ : variable de type entier

px^ <- 20

d. Libération de la mémoire

Dès que le ou les pointeurs ne sont plus utiles, l'espace mémoire qui leur est associé doit être libéré. La libération de l'espace mémoire réservé est la procédure "réciproque" de **allouer.** Elle indique au système d'exploitation que cette partie de la mémoire dynamique n'est plus utilisée (*il peut désormais en faire ce qu'il veut*).

Exemple :

Libérer (px) // la zone pointée par px est libre de nouveau

Remarques

* Pour indiquer l'absence d'adresse dans un pointeur, on utilise le pointeur particulier Nil.
* L'affectation p <- Nil veut dire que p ne pointe vers aucune variable.
* Il ne faut jamais utiliser l'indirection (^) lorsque le pointeur n'a pas été créé, a été déjà libéré, ou contient Nil.

Exp

Var

p : ^entier

Debut

/* le contenu de p est indéterminé : une adresse quelconque de la mémoire

P^ <- 10 /* Erreur

Fin

* Il est tout de même interdit de référencer une variable dynamique après sa destruction.

Var

p : ^entier

Debut

Allouer(p)

P^<-10

Libérer(p)

P^ <-20 /* Erreur : affectation d'un entier (20) à une zone indéterminée

Fin

3. Les pointeurs et les tableaux

Soit Tab un type tableau prédéfini ;

T1 : Tab // réservation d'une zone mémoire appelée T1 qui est une suite de cases successives. La variable T1 est réservée même si elle n'est pas utilisée.

P : ^Tab // P destiné à contenir une adresse seulement = une seule case mémoire.

Au lancement du programme, l'espace mémoire permettant de stocker ce pointeur (cette adresse mémoire) est réservé.

Allouer (P) ⇔ t : Tab et P := adresse(t) // l'adresse de la première case du tableau est affectée à P

Accès aux cases du tableau :

P^[i] : ième case

STRUCTURES DE DONNEES

CHAPITRE VI - LES TYPES ABSTRAITS DE DONNEES

1. Présentation du concept de TAD

La conception d'un algorithme de complexité moyenne ou élevée se fait en plusieurs étapes pendant lesquelles on effectue des raffinements successifs de la solution proposée.

La première version d'un tel algorithme est en général indépendante de son implémentation. En particulier, la représentation des données n'est pas fixée. A ce premier niveau, les données sont donc décrites par une notation particulière, un ensemble d'opérations ainsi que leurs propriétés et sont manipulées de façon abstraite c'est-à-dire non liée à la représentation. C'est exactement ça le concept de Type Abstrait de Données que nous désignerons dans la suite par TAD.

Dans les étapes ultérieures, les différentes représentations du TAD donneront plusieurs implémentations de l'algorithme en question. Si les types de données qu'on connaît ne représentent pas d'une manière cohérente la réalité, on peut définir des types adaptés à notre contexte, on les appelle TAD.

Pour expliquer davantage ce que c'est qu'un Type Abstrait de Données, nous pouvons considérer par exemple le type « **Int** » dans le langage C. Pour utiliser des données de ce type, le langage C offre des conventions de manipulation couvrant l'écriture des constantes, la liste des opérations permises (+,-,*,…) ainsi que leurs propriétés. Il est donc tout à fait possible de manipuler des données de ce type sans savoir comment ils sont représentés en interne (nombre de bits, bit de signe…). C'est ça le concept de TAD qu'on utilise couramment sans le savoir **[Froidevaux et al., 1993]**.

Avantages du concept

Les avantages de ce concept consistent en le fait qu'il facilite la conception, puisqu'on n'a pas à prendre en compte la programmation ; elle est faite une fois pour toutes quelle que soit la représentation choisie ultérieurement pour le type abstrait. Dans les chapitres suivants nous allons appliquer cette démarche.

2. Signature d'un TAD

Un Type Abstrait de Données est défini par une signature représentant sa syntaxe et les propriétés représentant la sémantique. La syntaxe d'un type abstrait de données est décrite par le concept de signature. La signature englobe les noms des opérations ainsi que les types des arguments mais ne définit pas les propriétés de celles-ci.

La signature d'un TAD est la donnée de :

* Noms de certains ensembles utilisés par le type (booléen, entier,…) : ces noms sont appelés sortes (c'est l'équivalent de types en programmation)

* Noms d'un certain nombre d'opérations et de leurs profils. Le profil précise l'ensemble de valeurs des arguments et des résultats.

Exemple de signature

Sortes *Vecteur, Element, Entier*

Opérations

Ième : Vecteur x Entier → Element

Change-ième : Vecteur x Entier x Element → Vecteur

Bornesup : Vecteur → Entier

Borneinf : Vecteur → Entier

Dans le premier exemple, nous utilisons les entiers pour numéroter les éléments du vecteur mais il serait peu pratique de donner la signature et les propriétés des entiers quand on définit les vecteurs. D'ailleurs la signature donnée précédemment est incomplète car rien n'est dit sur les calculs permis sur les entiers et les éléments. On se donne donc la possibilité de réutiliser des types déjà définis dans la définition d'un nouveau type. Donc l'exemple précédent aurait dû être écrit comme suit :

Sorte *Vecteur*

Utilise *Element, Entier*

Opérations

Ième : Vecteur x Entier → Element

Change–ième : Vecteur x Entier x Element → Vecteur

Bornesup : Vecteur → Entier

Borneinf : Vecteur → Entier

La signature du type vecteur est l'union des signatures des types utilisés, enrichie des nouveaux noms de sortes et d'opérations. Dans ce cas, on peut utiliser par exemple l'addition sur les entiers…
On appelle **sorte définie** la ou les sortes correspondant aux nouveaux noms de sorte (tel que vecteur) et sorte **prédéfinie** la ou les sortes provenant des types utilisés.
Une opération est **interne** si elle rend un résultat d'une sorte définie (exp : change-ième). Toute valeur d'une sorte définie est le résultat d'une opération interne.
On dira qu'une opération est un **observateur** si elle a au moins un argument d'une sorte définie et si elle rend un résultat d'une sorte prédéfinie.

La signature définie ci-dessus n'est pas suffisante pour définir le TAD « Vecteur » car elle ne donne qu'une simple syntaxe. L'exemple ci-dessous montre plus l'insuffisance de la signature pour la définition du TAD.

Sorte *R, S, T*

Opérations

O : R x T → S

P : R x T x S → R

Q : R → T

V : R → T

Pour compléter la définition du TAD, il est impératif d'ajouter la sémantique des opérations définies dans sa signature.

3. Description des propriétés d'un TAD

Les propriétés des opérations d'un TAD sont en général décrites sous forme d'axiomes.

Par exemple : BorneInf (v) <= i <= BorneSup (v) => Ième(changer_ième(v,i,e),i)=e

Où v, i, e sont des variables respectivement de sortes Vecteur, entier et element.

De plus, les opérations énoncées sont parfois des fonctions partielles non définies partout. Le domaine de définition d'une opération partielle est défini par une précondition.

La définition d'un type abstrait de données est donc composée d'une signature et d'un ensemble de propriétés (axiome + précondition).

Terminons l'exemple de sorte Vecteur qui a été proposé précédemment…

Vect : Entier x Entier → Vecteur (retourne un vecteur où aucun élément n'est défini, les entiers c'est les deux bornes)

Init : Vecteur x Entier → Booléen (il faut aussi savoir si une certaine case du vecteur a été initialisée)

Donc la signature devient

Sorte ***Vecteur***

Utilise ***Element, Entier, booléen***

Opérations

Vect : Entier x Entier → Vecteur (op interne) //*donne un vecteur où aucun élément n'est défini.*

Init : Vecteur x Entier → Booléen (observateur) // *permet de savoir si un élément a été associé à un certain indice.*

Ième : Vecteur x Entier → Element (observateur)

Change–ième : Vecteur x Entier x Element → Vecteur (op interne)

BorneSup : Vecteur → Entier (observateur)

BorneInf : Vecteur → Entier (observateur)

Préconditions

Ième(v,i) est défini SSi BorneInf(v)<=i<=BorneSup(v) et Init(v,i)=vrai

Change-ième(v,i,e) est défini SSi BorneInf(v)<=i<=BorneSup(v)

Axiomes

Init(vect(i,j),k)=faux

BorneInf(Vect(i,j))=i

BorneSup(Vect(i,j))=j

Init(v,i)=faux et borneInf(v)<=i<=borneSup(v) => Init(change-ième(v,i,e),i)=vrai

BorneInf(v)<=i<=BorneSup(v) et i ≠ j => Init(change-ième(v,i,e),j)= Init(v,j)

BorneInf(v)<=i<=BorneSup(v) et BorneInf(v)<=j<=BorneSup(v) et i ≠ j => Ième(Change-ième(v,i,e),j)=Ième(v,j)

BorneInf(v)<=i<=BorneSup(v) => Ième(Change-ième(v,i,e),i)=e

BorneInf(Change-ième(v,i,e))=BorneInf(v)

BorneSup(Change-ième(v,i,e))=BorneSup(v) Avec v: Vecteur, i,j: entier, e: Element

La définition des propriétés des opérations implique une certaine rigueur et cohérence. Pour ceci deux propriétés doivent être respectées :

La Consistance : les axiomes ne doivent pas être contradictoires.

La Complétude : nombre suffisant d'axiomes pour décrire toutes les propriétés du type abstrait à spécifier.

CHAPITRE VII - LES LISTES

1. Définition

Les listes linéaires sont la forme la plus courante d'organisation des données. On les utilise pour organiser des données qui doivent être traitées de manière **séquentielle**. Elles doivent être évolutives, c'est à dire que l'on doit pouvoir **ajouter** et **supprimer** des éléments.

Une *liste* est une suite finie (éventuellement vide) d'éléments **de même type** repérés selon leur **rang** dans la liste **[Froidevaux et al., 1993]**.

On remarque que l'ordre des éléments est fondamental. Mais attention, il ne s'agit pas d'un ordre sur les valeurs des éléments mais d'un ordre sur les places des éléments dans la liste. Deux visions des listes sont possibles :

Vision **itérative** qui considère une liste comme étant un ensemble d'éléments où chaque élément pointe vers l'élément suivant. Dans ce cas on se base sur les opérations d'accès et d'insertion.

Vision **récursive** qui définit une liste par son premier élément celui-ci pointe vers la liste des éléments suivants. Dans ce cas on se base sur les opérations Tete qui rend la première place de la liste et Fin qui rend la liste des éléments suivants. Pour éviter les redondances, on s'intéressera, dans ce chapitre, à plusieurs implémentations des « Listes itératives » et on traitera les « Listes récursives » qui sont équivalentes aux « Piles » dans le chapitre suivant.

2. Types abstraits « Liste »

a. Liste itérative

Signature

Sorte Liste, Place

Utilise : Entier (rang), Element (valeur)

Opérations :

Liste-vide : → Liste	*interne* pour *Liste*
Acces : Liste x Entier → Place	*observateur* pour *Liste et interne pour Place*
Contenu : Place → Element	*observateur pour Place*
Longueur : Liste → Entier	*observateur* pour *Liste*
Supprimer : Liste x Entier → Liste	*interne* pour *Liste*
Insérer : Liste x Entier x Element → Liste	*interne* pour *Liste*

Succ : Place → Place *interne pour Place*

Ième : Liste x Entier → Element *observateur* pour *Liste*

on peut s'arrêter là et ne pas ajouter I^{ème} puisque l'opération accès existe et a à peu près le même rôle sauf que le résultat de type place n'est pas très intéressant : ce qui caractérise vraiment une liste c'est le contenu de la ième place donc on ajoutera Ième : ***Ième(l,k)=Contenu(Acces(l,k))***

Préconditions :

Accès (l,k) est défnini SSI 1<=k<=Longueur(l)

Supprimer (l,k) est défini SSI 1<=k<=Longueur(l)

Insérer (l,k,e) est défini SSI 1<=k<=Longueur(l)+1

Ieme (l,k) est défini SSI 1<=k<=Longueur(l)

Axiomes :

Longueur (Liste-vide)=0

Si l ≠ Liste-vide et 1<=k<=Longueur(l) alors Longueur(supprimer(l,k))=Longueur(l)-1

Si 1<=k<=Longueur(l)+1 alors Longueur(inserer(l,k,e))=Longueur(l)+1

l ≠ Liste-vide et 1<=k<=Longueur(l) et 1<=i<k => Ième(Supprimer(l,k),i)=Ième(l,i)

l ≠ Liste-vide et 1<=k<Longueur(l) et k<=i<=Longueur(l)-1 =>

Ième(Supprimer(l,k),i)=Ième(l,i+1)

1<=k<=Longueur(l)+1 et 1<=i<k => ième(insérer(l,k,e),i)=ième(l,i)

1<=k<=Longueur(l)+1 et i=k => ième(insérer(l,k,e),i)=e

1<=k<=Longueur(l)+1 et k<i<=Longueur(l)+1 => ième(insérer(l,k,e),i)=ième(l,i-1)

Pour Place

l≠ liste-vide et l<=k<Longueur(l) => Succ(accès(l,k))=accès(l,k+1)

Avec *l : Liste ; i,k : Entier et e : Element*

3. Représentation des listes

a. Représentation contigüe

Dans ce cas la liste est représentée par un tableau dont la ième case est la ième place. La longueur de la liste ne doit pas dépasser la longueur du tableau qui est surdimensionné. Donc pour manipuler la liste nous avons besoin de connaître sa longueur parce que les cases du tableau ne sont pas toutes significatives. La liste est donc représentée par un couple <tableau, entier>.

La liste est représentée par un enregistrement contenant 2 champs :

```
ListeCont = struct
              T : tableau[1..Max] de Element
              Longueur : 0..Max
            Fin struct
```

1	2			n		Max
e1	e2	e3		En		

Longueur

N

Liste itérative et représentation contigue

Procedure Liste-vide (RES L : ListeCont**)**

Debut

L.longueur = 0

Fin

Procédure Ieme(DON L : ListeCont, k :entier, RES e : Element)

Var

Debut

Si k>=1 et k<=L.Longueur alors

e=L.T[k]

Fsi

Fin

Fonction Longueur(DON L :ListeCont) : entier

Var

Debut

Longueur = L.Longueur

Fin

Procédure Insérer (DONRES L : ListeCont, DON k : entier, e : Element)

Var

i, n : entier

Debut

n=L.Longueur

Si (k>=1 et k<=n+1 et n< Max) alors

Si k<=n alors

```
            Pour i de n+1 à k+1 pas (-1) faire
                L.T[i] = L.T[i-1]
            Fpour
        Fsi

            L.T[k]=e
            L.Longueur=n+1
    Sinon
        Ecrire (« insertion impossible»)
Fsi
Fin
Procédure Supprimer (DONRES L : ListeCont, DON k : entier)
Var
i, n : entier
Debut
N=L.Longueur
Si (k>=1 et k<=n) alors
        Si k<n alors
            Pour i de k à n-1 faire
                L.T[i] = L.T[i+1]
            Fpour
        Fsi
            L.Longueur=n-1
    Sinon
        Ecrire (« suppression impossible »)
Fsi
Fin
```

Avantages

Accès direct aux éléments

Insertion et suppression faciles du dernier élément

Inconvénients

Limitation de la taille de la liste

Insertion ou suppression à la kème place nécessitent un décalage

Conclusion

Cette représentation est relativement bien adaptée aux listes itératives : l'accès et le parcours sont très efficaces, mais les opérations d'insertion/suppression ailleurs qu'en fin de liste sont coûteuses et il faut savoir majorer la taille de la liste. Elle est par contre mal adaptée aux listes récursives : l'accès au premier élément est facile mais il n'est pas commode de représenter les opérations «cons » et « fin ».

b. Représentation chainée

Dans cette représentation les éléments de la liste sont chaînés entre eux à l'aide des pointeurs. La liste est déterminée par l'adresse de son 1er élément. La liste est représentée par un article contenant 2 champs <valeur, ptr>.

```
noeud= struct
            Val : Element
            Suiv : ^noeud
      Fin struct
ListCh=^noeud
```

Remarques :

La liste vide est représentée par le pointeur Nil.

Si l ≠ listevide alors l contient l'@ de la tête de la liste.

Listes itératives et représentation chainée

```
Procedure Liste-Vide (RES L : ListeCh)
Debut
L← Nil
Fin

Fonction Longueur (DON L : ListeCh) : entier
Var
Aux : ListeCh
Long : entier
Debut
Aux ← l
Long← 0
Tant que (Aux<> NIL) faire
      Long ← Long+1
      Aux ← Aux^.suiv
```

```
FTQ
Longueur ← Long
Fin
Fonction Longueur(DON L : ListeCh) : entier     # Récursive
Debut
Si L=Nil alors Longueur ← 0
Sinon
Longueur ← Longueur(L^.suiv)+1
Fsi
Fin
Procedure ieme (DON L : ListeCh, k : entier RES e: element, Erreur : booléen)
Var
Aux : ListeCh
Debut
Si k>=1 et k<=Longueur(L) alors
        aux ← L
        i ← 1
        Tant que  i < k faire
                aux=aux^.suiv
                i ← i+1
        FTQ
        e ← aux^.val
        Erreur ← faux
Sinon
        Erreur ← vrai
Fsi
Fin
Procedure supprimer (DONRES L : ListeCh, DON k :entier, RES Erreur : booléen)
version1
Var
i : entier
aux,p : ListeCh

Debut
Si k>=1 et k<=Longueur(L) alors
```

```
            aux← L
            Erreur ← faux
            Si k=1 alors
                  L← L^.suiv
                  Liberer (aux)
            Sinon
                  i ← 1
                  Tant que  i < k-1 faire
                        Aux=Aux^.suiv
                        i ← i+1
                  FTQ
                  p← aux^.suiv
                  aux^.suiv← p^.suiv
                  liberer(p)
            Fsi
Sinon
            Erreur ← vrai
Fsi
Fin
```

```
Procedure inserer (DONRES L : ListeCh, DON k :entier, e : Element, RES Erreur : booléen)
Var
P,aux : ListeCh
i : entier
Debut
Si k>=1 et k<=Longueur(l)+1 alors
      Créer(p)
      p^.val← e
      Erreur ← faux


      Si k=1 alors
            p^.suiv← l
            l← p
      sinon
```

```
                aux ← l
                i ← 1
                Tant que  i < k-1 faire
                        aux← aux^.suiv
                        i ← i+1
                FTQ
                p^.suiv← aux^.suiv
                aux^.suiv ← p
        Fsi
Sinon
        Erreur ← vrai
Fsi
Fin
```

Avantages

Allocation dynamique

Taille non limitée

Facilité des opérations d'insertion et de suppression à n'importe quelle position de la liste

Inconvénients

Accès séquentiel

Nécessite de la place supplémentaire pour les pointeurs

Conclusion

Cette représentation est adaptée aux cas où la liste est évolutive et on n'a aucune hypothèse sur le nombre maximal d'éléments.

4. Autres types de listes

a. Liste circulaire

Dans une liste circulaire on remplace, dans la dernière place de la liste, le pointeur à Nil par un pointeur vers la tête de la liste. Ce type de listes est utile pour représenter les files (que nous verrons dans les chapitres suivants).

Si l'on choisit la dernière place de la liste comme point d'entrée ; il suffit de parcourir un seul lien pour retrouver la tête de la liste.

b. Liste doublement chaînée

Dans la représentation classique, le parcours des listes est orienté dans un seul sens : du premier élément vers le dernier. Mais de nombreuses applications nécessitent de parcourir la liste à la fois

vers l'avant et vers l'arrière, et dans ce cas on peut faciliter le traitement en rajoutant des pointeurs arrière ou chaînage arrière. On obtient une liste doublement chaînée. Chaque place de cette liste contient un pointeur vers la place suivante (successeur) et un pointeur vers la place précédente.

c. Exercice

On considère la liste doublement chaînée représentée comme suit :

```
LD=struct
        First : ^Nœud
        Last : ^Nœud
    Finstruct
Noeud=struct
            Val: Element
            Pred: ^Noeud
            Suiv: Noeud
        Finstruct
```

1. Donner la procédure d'insertion en tête d'un élément dans une liste doublement chaînée L.
2. Donner la fonction qui permet de supprimer le dernier élément d'une liste doublement chaînée L et de retourner sa valeur. On suppose que la liste n'est pas vide.

```
Procedure Insere-tete(DONRES L : LD, DON e : element)
Var
P :^nœud
Debut
Créer(p)
p^.val←e
p^.pred←Nil
p^.suiv←L.First
L.First←p


Si L.Last=Nil alors     # la liste était vide
        L.Last← p
Sinon               #si la liste n'était pas vide
        L.First^.Suiv^.Pred ← p
Fsi
p ← Nil
```

Fin

Procédure Supprime-queue(DONRES L:LD, RES e : Element)

Var

p : ^nœud

x : Element

Debut

p ← L.Last

L.Last ← p^.Pred

Si L.Last=Nil alors # la liste contenait un seul element

L.First ←Nil

sinon

L.Last^.suiv ← Nil

Fsi

x ← p^.val

liberer(p)

p← Nil

Supprimequeue ← x

Fin

CHAPITRE VIII - LES PILES

1. Définition

La pile est une structure de données abstraite linéaire (chaque elt a un successeur et un prédécesseur sauf le premier et le dernier) dont le seul et unique élément accessible est le sommet de la pile. Elle est appelée aussi LIFO (Last in First Out) cad dernier-entré-premier-sorti. Les opérations d'insertion/ suppression ne se font qu'au sommet.

Exemple d'utilisation de la pile

L'appel en cascade de plusieurs sous-programmes est géré par une pile. Si le programme A appelle B qui appelle C qui appelle D, la pile des programmes en attente évoluera comme suit :

				C			
			B	B	B		
Pile		A	A	A	A	A	
Temps	T0	T1	T2	T3	T4	T5	T6

T0 : lancement du programme A

T1 : appel du programme B par A

T2 : appel du programme C par B

T3 : appel du programme D par C

T4 : poursuite du programme C (quand D est terminé)

T5 : poursuite du programme B (quand C est terminé)

T6 : poursuite du programme A (quand B est terminé)

2. Type abstrait « Pile »

Les opérations sur les piles sont :

Créer une pile ; Tester si une pile est vide ; accéder au sommet d'une pile ; empiler un élément ; retirer l'élément qui se trouve au sommet (dépiler).

La signature du type est donc :

Sorte : Pile

Utilise booléen, Elément

Opérations :

Pile-Vide : → Pile

Empiler : Pile x Element → Pile

Dépiler : Pile → Pile

Sommet : Pile → Element

Est-vide : Pile → Booléen

Pré-conditions :

Dépiler (P) **est défini ssi** Est-vide (P) = faux

Sommet (P) **est défini ssi** Est-vide (P) = faux

Axiomes :

P est une Pile, et e est un Element

Dépiler (Empiler (P,e))=P

Sommet (Empiler (P,e)) =e

Est-vide (Pile-Vide)=vrai

Est-vide (Empiler (P,e))=faux

Pour implémenter les piles on peut utiliser les représentations étudiées pour les listes.

3. Représentation des Piles

a. Représentation contigüe

P est représenté par :

Un tableau contenant les éléments de la pile ;

Un indice indiquant le sommet de la pile.

```
Pile= struct
            taille : entier
            Tab : tableau [1..max] de Element
      Fin struct
```

Remarque :

Comme le nombre maximum d'éléments est fixé, on peut ajouter une autre fonction à résultat booleen PilePleine.

Avantages et inconvénients :

Manipulation très facile de la pile et non couteuse en temps ; mais nécessité de majorer le nombre d'éléments de la pile !

b. Représentation chainée

Les éléments de la pile sont chaînés entre eux, et le sommet d'une pile non vide est le premier de la liste ; la pile vide est représentée par Nil.

```
      cellule= struct
            val : Element
            suiv : ^cellule
      Fin struct
      Pile= ^cellule
Procedure pile-vide (RES P : Pile)
```

Debut

P ← Nil

Fin

Fonction Est-vide (DON P : Pile) : Booleen

Debut

Est-vide ← (P=Nil)

Fin

Procedure Empiler (DONRES P : Pile, DON e : Element) *# même chose que cons*

Var

q : Pile

Debut

allouer(q)

q^.val ← x

q^.suiv ← P

P ← q

Fin

Procedure Depiler (DONRES P: Pile, RES e : Element)

 # même chose que la suppression en tête ou fin, ***on suppose que la pile est non vide***

Var

aux : Pile

Debut

aux ← P

e ← aux^.val

P ← P^.suiv

Liberer (aux)

Fin

Procédure Sommet (DON P: Pile, RES e : Element)

*même chose que Premier ;* ***on suppose que la pile est non vide***

Debut

e ← P^.val

Fin

Avantages et inconvénients

Pas de limitation de la taille mais cette représentation est plus couteuse en mémoire.

4. Exercice

Pour passer d'un nombre en base 10 à un nombre en base *N*, on peut appliquer la méthode suivante :

Soit *K* le nombre en base 10 à convertir en base *N*.

1. Effectuer la division entière de *K* par *N*. Soit *D* le résultat de cette division et *R* son reste ;
2. Si *D* > 0 , revenir à l'étape 1 en remplaçant K par D.
3. Sinon, l'écriture en base *N* de *K* est égale à la concaténation de tous les restes obtenus en commençant par le dernier.

Ecrire une procédure qui, étant donnés les deux nombres strictement positifs K et N, permet de convertir K en base N et d'afficher le résultat. Utiliser une pile comme structure intermédiaire.

```
Procédure Conversion (DON K : entier, N : entier)
Var
        E, D : entier
Debut
        Pile-vide(P)
        D ← K
        Tant que D>0 faire
                Empiler(P, D mod N)
                D ← D div N
        FTQ
        Ecrire (« Le nombre », K, « en base »,N, « est = »)
        M ← 0
        Tant que non Est-vide(P) faire
                Depiler (P,e)
                M ← M*10+e
        FTQ
        Ecrire (M)
Fin
```

CHAPITRE IX - LES FILES

1. Définition

Une file est une liste d'éléments caractérisée par deux extrémités (une tête et une queue). Dans le cas d'une file, les insertions se font à la queue et les suppressions et accès se font à la tête. Par analogie avec les files d'attente on dit que l'élément présent depuis le plus longtemps est le premier. Les files d'attente sont aussi appelées FIFO (First In First Out) c a d premier entré premier sorti.

2. Type abstrait de données « File »

Les opérations sur les files sont :
Tester si la file est vide ; accéder au premier élément de la file ; ajouter un élément dans la file; retirer le premier élément de la file.
La signature du type File est donc :

Sorte : File
Utilise booléen, Elément
Opérations :
File-Vide : → File
Enfiler : File x Elément → File #ajouter
Défiler : File → File # retirer
Premier : File → Elément
Est-vide : File → Booléen

Pré-conditions :
Defiler (F) **est défini ssi** est-vide (F) = faux
Premier (F) **est défini ssi** est-vide (F) = faux

Axiomes :
Les axiomes différent de ceux de la pile en ce qui concerne les opérations Premier et Retirer. Pour tout F de sorte File et e de sorte Elément, on a :

Est-vide(F)= vrai => Premier(Enfiler(F,e))=e
Est-vide(F)= faux => Premier(Enfiler(F,e))= Premier(F)

Est-vide(F)= vrai => Defiler(Enfiler(F,e))=File-vide
Est-vide(F)= faux => Defiler(Enfiler(F,e))= Enfiler(Defiler(F),e)

Est-vide(File-vide)=vrai
Est-vide(Enfiler((F,e))=faux

On peut représenter les files de manière contigüe ou de manière chaînée.

3. Représentation des files

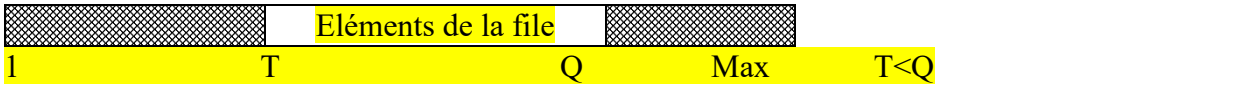
a. Représentation contigüe

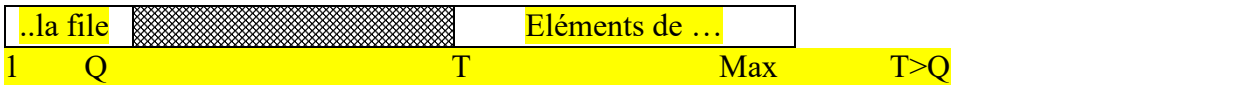
Dans ce cas quelles seront les informations à conserver ?

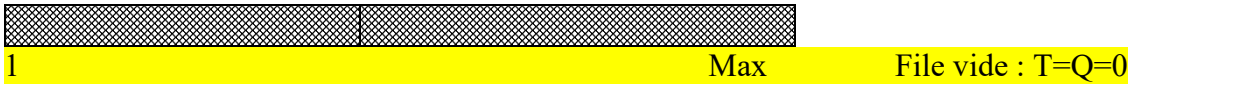
Si on choisit de représenter une file par un tableau et l'indice de la queue. Comme la suppression se fait à la tête donc ceci imposera des décalages : solution trop couteuse et non efficace. Afin d'éviter ceci, la file sera représentée par une structure contenant les informations suivantes : la queue et la tête.

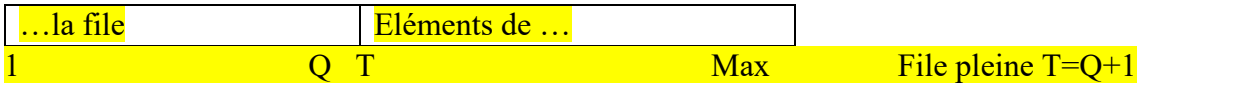
File= struct
    FA : tableau [1..Max] de Element
    t : entier ou 0..Max
    q : entier ou 0..Max
Fin struct
Ou alors
File= struct
    FA : tableau [1..Max] de Element
    T,Q : entier ou 0..Max
Fin struct

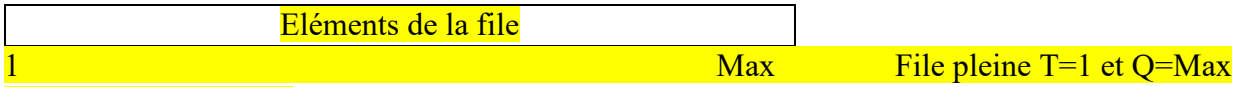
Exemples :











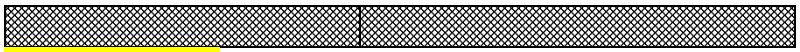
(T=Q+1 modulo Max)

Développer les opérations enfiler et defiler.

Opérations :

Procédure File-vide (DONRES F : File)
Debut
    F.T ← 0
    F.Q ← 0
Fin
Fonction Est-vide (DON F : File) : booléen
Debut
    Si F.T=0 alors
        Est-vide ← vrai
    Sinon
        Est-vide ← faux
    Fsi
Fin

Procédure Premier (DON F :File RES e : Element, Erreur : booléen)
Début
Erreur ← vrai
Si (non Est-vide(F)) alors
Erreur ← faux
e ← F.FA [F.T]
Fsi
Fin
Procédure Enfiler (DONRES F : File, DON e : Element)
Début
Si (F.T - F.Q=1 ou (F.T=1 et F.Q=Max)) alors ***# file pleine***
Ecrire (« Enfilement impossible : file pleine »)
Sinon
Si F.Q < Max alors
F.Q ← F.Q+1
Sinon ***# F.Q=Max***
F.Q ← 1
Fsi
F.FA[F.Q] ← e
Si (F.T = 0) alors ***# file de départ vide***
F.T ← 1
Fsi
Fsi
Fin
Schéma explicatif :



File vide T=Q=0

Enfilement d'un élément

T=1, Q=1 file contenant un seul élément

Procédure Défiler (DONRES F : File, RES e : Element, Erreur : booléen)
Début
Si (non Est-vide(F)) alors
Erreur ← faux
e ← F.FA[F.T]
Si F.T= F.Q alors ***# la file de départ contient un seul élément***
F.T ← 0
F.Q ← 0
Sinon ***# la file de départ contient plus qu'un élément***
Si F.T=Max alors
F.T ← 1
Sinon
F.T ← F.T+1
Fsi
Fsi

```
Sinon
        Erreur ← vrai
Fsi
Fin
```

b. Représentation chaînée

```
Def Type
cellule= struct
            val : Element
            suiv : ^cellule
        Fin struct
File= struct
            T,Q : ^cellule
        Fin struct
```

Autre possibilité : on peut utiliser une liste circulaire pour représenter une file. Dans ce cas nous utiliserons un seul pointeur.

Développer les opérations enfiler et defiler.

```
Procédure Enfiler (DONRES F :File, DON e : Element)      #insertion en queue
Var
Aux : ^cellule


Debut
        Créer(aux)
        aux^.val ← e
        aux^.suiv ← Nil
        Si F.T=Nil alors          # file de départ vide
            F.T ← aux
            F.Q ← aux
        Sinon
            F.Q^.suiv ← aux
            F.Q ← aux
        Fsi
Fin
```

```
Procédure Defiler (DONRES F :File, RES e : Element, Erreur : booléen)
Var
aux : ^cellule
Debut
        Si Est-vide(F) alors
            Erreur ← vrai
        Sinon
            Erreur ← faux
            e← F.T^.val
            aux ← F.T
```

```
                F.T ← F.T^.suiv
                Liberer(aux)
                Si (F.T=Nil) alors     # la file de départ contient un seul élément
                        F.Q ← Nil
                Fsi
        Fsi
Fin
```

4. Exercice

Ecrire un algorithme qui permet de fusionner deux files F1 et F2 à éléments ordonnés dans une file F3 ordonnée. *Utiliser une représentation chaînée.*

```
Procédure Fusion (DON F1,F2 : File, RES F3 : File)
Var
E : Element
Debut
        File-vide (F3)
        Tant que non(Est-vide(F1) et non Est-vide(F2) ) faire
                Si Premier(F1) <Premier(F2) alors
                        Defiler(F1,e)
                Sinon
                        Defiler(F2,e)
                Fsi
                Enfiler(F3,e)
        FTQ
        Tant que non Est-vide(F1) faire
                Defiler(F1,e)
                Enfiler(F3,e)
        FTQ
        Tant que non Est-vide(F2) faire
                Defiler(F2,e)
                Enfiler(F3,e)
        FTQ
Fin
```

CHAPITRE X - LES ARBRES BINAIRES

1. Définition

Une **arborescence** désigne la représentation d'une structure hiérarchique à plusieurs niveaux, rappelant la structure d'un arbre. Une structure arborescente est un ensemble d'informations homogènes organisées en niveaux : chaque information d'un niveau peut être reliée à plusieurs informations du niveau supérieur.

En informatique, **l'organisation des fichiers sur un support de stockage est structurée en arborescence**, chaque répertoire ou dossier étant une *branche* pouvant comporter des *feuilles* (fichiers) et des nœuds de départ d'autres branches (sous-dossiers). À la base d'une arborescence se trouve un répertoire appelé *racine*. Ce répertoire peut contenir des fichiers et des répertoires, qui eux-mêmes peuvent contenir la même chose. **C'est aussi sous forme d'arbre que sont organisés les fichiers dans des systèmes d'exploitation tels qu'UNIX. Les programmes traités par un compilateur** sont représentés sous forme d'arbre.

Une propriété intrinsèque de la structure d'arbre est la récursivité. Les définitions des caractéristiques des arbres, aussi bien que les algorithmes qui les manipulent s'écrivent très naturellement de manière récursive.

2. Terminologie

Dans ce qui suit nous utiliserons un vocabulaire inspiré des arbres généalogiques.

Racine : nœud de niveau zéro (n'a pas de père).

Nœud : racine (ou sommet) de tout sous-arbre. Chaque nœud a 0 ou plusieurs fils.

Feuille : un nœud qui n'a pas de fils.

Fils : x est le fils de y si x est la racine du sous arbre y.

Père : x est le père de y si y est le fils de x.

Frère : deux nœuds sont frères s'ils ont le même père.

Ascendants : A est ascendant de B si A est le père de B ou un ascendant de B.

Descendants : X est descendant de Y si x est le fils de y ou un descendant d'un fils de y.

Branche : tout chemin de la racine à une feuille de l'arbre.

Hauteur d'un nœud (ou profondeur ou niveau) est définie récursivement comme suit : étant donné x un nœud d'un arbre B,

h(x) =0 si x est la racine de B et, h(x)= 1+h(y) si y est le père de x.

Avantages de l'utilisation des structures arborescentes :

- Adaptation à la représentation naturelle des informations homogènes organisées ;
- Grande commodité et rapidité de manipulation des informations grâce à la notion de récursivité.

3. Définition du TAD

Le type arbre binaire est soit vide (noté ∅), soit de la forme B=<o,B1,B2>, où B1 et B2 sont des arbres binaires disjoints et 'o' est un nœud appelé racine. B1 est le sous-arbre gauche de la racine de B et B2 est sous-arbre droit.

Sorte arbre

Utilise : **Nœud, Element**

Opérations

Arbre-vide : → Arbre

< - , - , - > : Nœud x Arbre x Arbre → Arbre

Racine : Arbre → Nœud

g : Arbre → Arbre

d : Arbre → Arbre

contenu : Nœud → Element

Préconditions

racine(B1) est défini Ssi B1 <> Arbre-vide

g(B1) est défini Ssi B1 <> Arbre-vide

d(B1) est défini Ssi B1 <> Arbre-vide

Axiomes

racine(<o,B1,B2>)=o

g(<o,B1,B2>)=B1

d(<o,B1,B2>)=B2

4. Mesures sur les arbres

- La **taille** d'un arbre est le nombre de ses nœuds ; on définit récursivement l'opération taille par :

  taille(arbre-vide)=0 et

  taille(<o,B1,B2>)=1+taille(B1)+taille(B2).

- La **hauteur d'un nœud** (on dit aussi profondeur ou niveau) est définie récursivement de la façon suivante, étant donné x un nœud de B,

  h(x)=0 si x est racine de B, et h(x)=1+h(y) si y est le père de x.

- La **hauteur ou profondeur d'un arbre B** est :

  h(B)=max {h(x) ; x nœud de B}

5. Arbres binaires particuliers

- Un arbre binaire est **dégénéré** ou **filiforme** si tous ses nœuds ont au maximum un seul fils.
- Un arbre binaire est **complet** s'il contient un nœud au niveau 0, 2 nœuds au niveau 1, 4 nœuds au niveau 2. D'une façon générale **2^h nœuds au niveau h.** *(chaque nœud est complètement rempli)*
- Un **arbre binaire parfait** est tel que tous les niveaux sauf éventuellement le dernier sont remplis, et dans ce cas les feuilles du dernier niveau sont groupées à gauche.

6. Représentation des arbres binaires

a. Représentation chaînée

```
Nœud = struct
        Val : Element
        g,d : ^Nœud
    Fin struct
Type Arbre=^Nœud
```

A=nil ⇔ A= arbre-vide, A^.val ⇔ contenu (racine (A)), A^.g ⇔ g(A) et A^.d ⇔ d(A).

b. Représentation contigüe

Avec cette représentation le nombre de nœuds de l'arbre est limité et ne doit pas dépasser une valeur Max.

i. Représentation indexée

A chaque nœud de l'arbre on associe une valeur ainsi que les indices des deux fils gauche et droit. On prendra la convention suivante : indice =0.

```
Nœud = struct
        Val : Element
        g,d : 0..Max
    fin struct
Type ArbTab= tableau [1..Max] de Nœud
Arbre = struct
        Rac : 0..Max
        T : ArbTab
    Fin struct
```

Un exemple d'arbre binaire serait représenté par une variable A de type Arbre, dont la racine A.Rac contient l'indice 3 et le champ T contient le tableau suivant.

	V	G	D
d		0	10
a		5	6
g		0	0
b		2	0
c		13	11
f		0	0
m		0	0
e		8	4
l		9	0
k		0	0

Traduction des opérations définies dans le TAD.

A.Rac=0 ⇔ A= arbre-vide,

Si A.Rac=r et r>0 alors l'arbre n'est pas vide. A.T[r]=racine(A), A.T[r].val= contenu(racine(A)) et A.T[r].G et A.T[r].D sont les indices de g(A) et d(A).

Remarque

Dans cette représentation indexée, l'indice associé à chaque nœud est arbitraire ; la suppression/ajout d'un nœud ne présente donc pas de difficultés dans la limite des places réservées pour le tableau. On garde donc un des avantages de l'allocation dynamique.

ii. Représentation séquentielle

Cette représentation se base sur la convention suivante : si un nœud est numéroté par i, son fils gauche se trouve à l'indice 2i et son fils droit se trouve à l'indice 2i+1. Dans cette représentation le passage d'un nœud à un autre se traduit par un simple calcul d'indice dans le tableau :

2<=i<=Max => le père du nœud d'indice i est à l'indice i div2

1<=i<=Max div 2=> le fils gauche du nœud d'indice i est en 2i

Le fils droit du nœud d'indice i est en 2i+1

Notons que cette représentation nous impose, dans le cas d'arbres binaires quelconques, de laisser des cases vides dans le tableau pour marquer la place des nœuds non présents. Ceci nous fait perdre l'avantage de compacité du tableau.

7. Parcours d'un arbre binaire

Parcourir un arbre revient à visiter tous ses nœuds. Une des opérations les plus fréquentes mises en œuvre par les algorithmes qui manipulent les arbres consiste à parcourir ceux-ci. Il existe plusieurs ordres dans lesquels les nœuds peuvent être visités, et chacun a des propriétés utiles qui peuvent être exploitées par les algorithmes basés sur les arbres binaires.

a. Parcours en largeur

Ce parcours essaie toujours de visiter le nœud le plus proche de la racine qui n'a pas été visité. En suivant ce parcours, on va d'abord visiter la racine, puis les nœuds à la profondeur 1, puis 2… D'où le nom parcours en largeur.

b. Parcours en profondeur main gauche

Ce parcours consiste à tourner autour de l'arbre en suivant le chemin indiqué sur l'exemple. Ce chemin part à gauche de la racine, et va toujours le plus à gauche possible en suivant l'arbre.

Si on essaie d'écrire un algorithme récursif qui reflète ce parcours, on va visiter 3 fois chaque nœud. Appelons Traitement1 (resp. Traitement 2 et Traitement 3) la suite d'actions exécutées lorsqu'un nœud est rencontré pour la première (resp. deuxième et troisième) fois. L'algorithme est le suivant :

```
Procédure Parcours (DON A : Arbre)
Debut
    Si (A <> Arbre-vide) alors
        Traitement 1
        Parcours (g(A))
        Traitement 2
        Parcours (d(A))
        Traitement 3
    Fsi
Fin
```

On remarque ici que la complexité est assez élevée puisque chaque nœud est visité 3 fois. Souvent on n'a pas 3 traitements différents à exécuter : par exemple si on veut écrire les valeurs des nœuds de l'arbre en appliquant cet algorithme on va les écrire 3 fois et c'est inutile.

D'où l'intérêt de l'existence de trois ordres classiques d'exploration de l'arbre :

- Ordre préfixé : père, fils gauche, fils droit
- Ordre infixé : fils gauche, père, fils droit
- Ordre postfixé : fils gauche, fils droit, père

c. Exercices

1. Ecrire une procédure récursive qui permet d'afficher les valeurs des nœuds d'un arbre binaire selon l'ordre préfixé chaque valeur est affichée une et une seule fois.

```
Procedure P-Prefixé (DON A : Arbre)
Debut
    Si (A<> Arbre-vide) alors
        Ecrire(A^.val)
        P-Prefixé(g(A))
        P-Prefixé(d(A))
    Fsi
Fin
```

2. Ecrire une version itérative de la procédure précédente.

Plusieurs manières de procéder :

On garde trace de tous les nœuds visités, afficher leur contenu, et se déplacer à leur fils gauche. Après remonter et afficher les contenus des fils droits des nœuds déjà visités.

```
cellule= struct
        val : Arbre
        suiv : ^cellule
    Fin struct
Pile= ^cellule

Procedure P-Prefixé (DON A : Arbre)
Var
P: Pile
Debut
Pile-vide(P)
Tant que (A <> nil) ou (non Est-vide(P)) faire     // non (A=Nil et Est-vide(P))
    Tant que (A<> Nil) faire
        Ecrire(A^.val)       //ou tout autre traitement
        Empiler (P,A)
        A ← g(A)
    FTQ

Depiler(P,A)
A← D(A)
FTQ
Fin
```

2eme version :

Si on est sur un nœud, on traite le nœud, on sauvegarde son fils droit s'il existe et on traite le fils gauche s'il existe sinon on dépile le dernier nœud.

```
Procedure P-Prefixé (DON A : Arbre)
Var
P: Pile
q : Arbre
Debut
q ← A
Pile-vide(P)
Tant que (q <> Nil) faire
        Ecrire(q^.val)
        Si (d(q) <> Nil) alors Empiler(P,d(q)) Fsi
        Si (g(q) <> Nil) alors q ← g(q)
        Sinon depiler(P,q) fsi
FTQ
Fin
```

3eme version :

```
Procedure P-Prefixé (DON A : Arbre)
Var
P: Pile
q : Arbre
Debut
Pile-vide(P)
Empiler(P,A)
Tant que (non Est-vide(P)) faire
        Depiler(P,q)
        Si (q <> Nil) alors
                Ecrire(q^.val)
                Empiler (P,d(q))
                Si g(q) <> Nil alors q ← g(q)
                Sinon depiler(P,q) Fsi
        Fsi
FTQ  Fin
```

CHAPITRE XI - LES ARBRES BINAIRES DE RECHERCHE

1. Définition

Un arbre binaire de recherche est un arbre binaire tel que pour tout nœud v de l'arbre :

- Les éléments de tous les nœuds du sous arbre gauche de v sont inférieurs ou égaux à l'élément contenu dans v.
- Les éléments de tous les nœuds du sous arbre droit de v sont supérieurs à l'élément contenu dans v.

Il en résulte de cette définition que le parcours symétrique d'un arbre binaire de recherche produit la suite des éléments triée en ordre croissant.

Avantages : recherche et tri efficaces

2. Recherche d'un élément dans un arbre binaire de recherche

Pour rechercher une occurrence d'un élément dans un arbre binaire de recherche, on compare cet élément au contenu de la racine :

- S'il y a égalité, l'élément est trouvé et la recherche s'arrête ;
- Si l'élément est plus petit (resp. plus grand) que celui de la racine, on poursuit la recherche dans le sous-arbre gauche (resp. droit) ; si ce sous-arbre est vide, il y a échec.

```
Fonction Rechercher (DON X : Element, A : Arbre) : booléen        #version récursive
Début
Si A= Nil alors Rechercher ← faux
Sinon si X= A^.val alors Rechercher ← vrai
        Sinon si X <A^.val alors Rechercher(X,g(A))
                Sinon Rechercher(X,d(A))
                Fsi
        Fsi
Fsi
Fin
```

```
Fonction Rechercher (DON X : Element, A : Arbre) : booléen        #version itérative
Var
trouve : booléen
q : Arbre
Début
trouve ← faux
q ← A
```

```
Tant que (q<>Nil et non trouve) faire
        Si X=q^.val alors
                trouve ← vrai
        sinon si  X< q^.val alors
                        q ← g(q)
                sinon
                        q← d(q)
                Fsi
        Fsi
FTQ
Rechercher ← trouve
Fin
```

3. Adjonction d'un élément

a. Adjonction aux feuilles

Principe

On compare l'élément au contenu de la racine pour savoir si l'ajout doit être fait dans le sous-arbre gauche ou dans le sous-arbre droit. On fait un appel récursif jusqu'à arriver à un arbre vide. On insère à cet emplacement l'élément à ajouter.

Procédures

```
Procédure Ajouter (DON X : Element, DONRES A : Arbre)          # version récursive
Début
Si A= Nil alors
        Créer(A)
        A^.val ← X
        A^.g ← Nil
        A^.d ← Nil
Sinon Si X<=A^.val alors Ajouter(X,A^.g)
        Sinon Ajouter(X, A^.d)
        Fsi
Fsi
Fin
Procédure Ajouter (DON X : Element, DONRES A : Arbre)          # version itérative
Var
p, q : Arbre
trouve : booléen
```

```
Début
Créer(p)
p^.val ← X
p^.g ← Nil
p^.d ← Nil
Si A = Nil alors
        A ← p
Sinon
        q ← A
        trouve ← faux
        Tant que non trouve faire
                Si X <= q^.val alors
                        Si q^.g= Nil alors
                                q^.g ← p
                                trouve ← vrai
                        sinon
                                q ← q^.g
                        Fsi
                Sinon  si q^.d =Nil alors
                                q^.d ← p
                                trouve ← vrai
                        sinon
                                q←q^.d
                        Fsi
                Fsi
        FTQ
Fsi
Fin
```

b. Adjonction à la racine

Principe :

L'adjonction à la racine peut être utile si les recherches portent sur les éléments **récemment ajoutés**.

Pour ajouter un élément X à la racine d'un arbre binaire de recherche, il faut d'abord couper l'arbre binaire de recherche en deux arbres binaires de recherche G et D contenant respectivement tous

les éléments inférieurs ou égaux à X, et tous les éléments supérieurs à X, puis former l'arbre dont la racine contient X et qui a pour sous-arbre gauche (resp. droit) l'arbre G (resp. D). On obtient bien un arbre binaire de recherche.

L'étape de coupure est la plus délicate ; il est important de remarquer qu'il n'est pas nécessaire de parcourir tous les nœuds de l'arbre initial A pour former G et D. Ce sont les nœuds situés sur le chemin C suivi lors de la recherche de X qui déterminent la coupure : en effet si un nœud de C contient un élément plus petit que X, il vient se placer sur le bord droit de G et par la propriété des arbres binaires de recherche, il entraîne avec lui dans G tout son sous arbre gauche. Si un nœud de C contient un élément plus grand que X, il vient se placer sur le bord gauche de D, et par la propriété des arbres binaires de recherche, il entraîne avec lui dans D tout son sous arbre droit

```
Procédure Coupure (DON X : Element, DONRES A, G, D : Arbre)
Début
    Si A=Nil alors
        G ← Nil
        D ← Nil
    Sinon
        Si A^.val> X  alors
            D ← A
            Coupure(X, A^.g, G, D^.g)
        Sinon          # A^.val <=X
            G ← A
            Coupure (X, A^.d, G^.d, D)
        Fsi
    Fsi
Fin
Procédure AjouterRac(DON X : Element, DONRES A : Arbre)
Var
r : Arbre
Debut
Créer(r)
r^.val ← X
couper(X, A, r^.g, r^.d)
A ← r
Fin
```

4. Suppression d'un élément

Pour supprimer un élément, il faut tout d'abord déterminer sa place, puis effectuer la suppression proprement dite, qui s'accompagne éventuellement d'une réorganisation des éléments.

Principe :

- Si le nœud à supprimer est sans fils : la suppression est immédiate ;
- Si le nœud possède un seul fils : il suffit de le remplacer par ce fils ;
- Si le nœud possède deux fils : il y a deux solutions
  - Remplacer l'élément à supprimer par l'élément immédiatement inférieur (le plus grand élément de son sous-arbre gauche)
  - Remplacer l'élément à supprimer par l'élément immédiatement supérieur (le plus petit élément de son sous-arbre droit)

  Ces deux solutions sont équivalentes lorsque tous les éléments de l'arbre binaire de recherche sont distincts.

On utilisera dans ce qui suit la première solution.

La procédure **supprimearb** utilise la procédure **supmax** de suppression du maximum dans un arbre binaire de recherche.

La procédure **supmax** supprime le nœud contenant l'élément maximal dans un arbre binaire de recherche A non vide ; il donne pour résultats l'élément maximal et l'arbre binaire de recherche A privé du nœud contenant cet élément.

```
Procédure Supmax (RES Max : Element, DONRES A : Arbre)    on suppose que A<>NIL
q : Arbre
Debut
Si A^.d=Nil alors
        q← A
        Max← A^.val
        A ← A^.g
        Liberer (q)
Sinon
        Supmax (Max,A^.d)
Fsi
Fin
```

La procédure **supprimearb** supprime le nœud contenant X dans un arbre binaire de recherche A ; il donne pour résultat A si X n'est pas dans A, et A privé de X sinon. Il utilise la procédure de suppression du maximum.

```
Procédure Supprimearb (DON X : Element, DONRES A : Arbre)
Var
Max : Element
Debut
Si A<>Nil alors
        Si X<A^.val alors
                Supprimearb(X, A^.g)
        Sinon si X> A^.val
                        Supprimearb(X, A^.d)
                Sinon                   # X=A^.val
                        Si A^.d=Nil alors
                                A ← A^.g
                        Sinon si A^.g=Nil alors
                                        A ← A^.d
                                Sinon                   # A^.d<>Nil et A^.g <>Nil
                                        Supmax(Max, A^.g)
                                        A^.val ← Max
                                Fsi
                        Fsi
                Fsi
        Fsi
Fsi
Fin
```