Université A-MIRA

ALGORITHMES

Cours

TINSALHI Faouzi

Sommaire :

INTRODUCTION GENERALE……………………………………………………………………………………..1

ELEMENTS DE BASE DE L'ALGORITHME……………………………………………………………………..4

LES ACTIONS……………………………………………………………………………………………………….14

TYPES ET STRUCTURES DE DONNEES……………………………………………………………………20

COMPLEXITE DES ALGORITHMES……………………………………………………………………………30

LES TYPES ABSTRAITS DE DONNEES…………………………………………………………………….38

INTRODUCTION GENERALE

I : ANALYSE, ALGORITHME, PROGRAMMATION :

But : acquérir une méthode, des outils : démarche a suivre d'un problème a résoudre à un programme informatique.

Domaines d'applications :
- ⇨ Gestion (facturation, paye,…)
- ⇨ Informatique scientifique (météorologie, astronomie,…)
- ⇨ Systèmes industriels (commandes numériques, robotique,…)
- ⇨ Informatique ludique (informatique personnelle, jeux,…)
- ⇨ Etc

Quelque soit le domaine, la démarche de conception du programme reste identique.
Démarche
Problème a résoudre

Etude Préalable	Compréhension du problème, modélisation du problème
Spécification des données et des résultats	Recenser les informations et préciser leur nature
Spécification de fonctionnalités	Recenser et préciser
Solution en langage naturel	Savoir résoudre le problème avant d'automatiser la solution
Données structurées / Algorithme	Mise en forme informatique des informations et des traitements à réaliser
Programmation	Choix du langage, traduction de la solution (algorithme) sous forme de programme
Programme exécutable	Compilation du programme en programme exécutable
Test et évaluation du travail réalisé	Test de la cohérence par rapport aux spécifications
Documentation	Manuel d'utilisateur, aide en ligne, manuel de maintenance du logiciel

Eventuellement, si la validation (avant dernière étape) a échoué, il faut remonter jusqu'à trouvé la source de l'échec.

La partie de l'ETUDE PREALABLE à SPECIFICATION DE FONCTIONNALITES s'appelle la partie ANALYSE
La partie de SOLUTION EN LANGAGE NATUREL à DONNEES… s'appelle la partie ALGORITHME.
La partie PROGRAMMATION à TEST est la phase de programmation.
La rédaction du manuel d'utilisateur et du manuel de maintenance s'effectue durant la partie DOC. TECHNIQUE.

II LA NOTION D'ALGORITHME :

Du mathématicien persan Al-Khwa-Rizm (Bagdad, 780 – 850)

Pour les notions de Al-Jabr (Algèbre) théorie du calcul

Plus ancien : Euclide (3eme siècle avant JC)
Babyloniens (1800 avant JC)

Selon le LAROUSSE, la définition d'algorithme est « un ensemble de règles opératoires dont l'enchaînement permet de résoudre un problème au moyen d'un nombre fini d'opérations. »

Quelques points importants :

- ⇨ Un algorithme décrit un traitement sur un ensemble fini de données de nature simple (nombres ou caractères), ou plus complexes (données structurées)
- ⇨ Un algorithme est constitué d'un ensemble fini d'actions composées d'opérations ou actions élémentaires. Ces actions élémentaires doivent être effectives (réalisable par la machine), non ambiguës.
- ⇨ Un algorithme doit toujours se terminer après un nombre fini d'opérations.
- ⇨ L'expression d'un algorithme nécessite un langage clair (compréhension) structuré (enchaînements d'opérations) non ambiguë, universel (indépendants du langage de programmation choisi)

Problème : un tel langage n'existe pas, on définit son propre langage.

III METHODOLOGIE DE CONCEPTION D'UN ALGORITHME :

Analyse descendante : (ou programmation structurées) : on décompose un problème complexe en sous problèmes et ces sous problèmes en d'autres sous problèmes jusqu'à obtenir des problèmes faciles a résoudre c'est-à-dire connus.
On résout les sous problèmes simples sous forme d'algorithme puis on recompose les algorithmes pour obtenir l'algorithme global du problème de départ.

Garder à l'esprit :

- ⇨ La modularité : un module résout un petit problème donné. Un module doit être réutilisable.
- ⇨ Lisibilité de l'algorithme (mise en page, commentaires, spécification : dire quoi mais pas comment)
- ⇨ Attention à la complexité de l'algorithme :
    - *Complexité en temps* : mesure du temps d'exécution en fonction de la taille des données
    - *Complexité en espace* : espace mémoire nécessaire pour effectuer les traitements.
- ⇨ Ne pas réinventer la roue (c'est-à-dire ne pas refaire les programmes standard dont les solutions sont connues) ce qui implique avoir une certaine culture et un outil technique standard

ELEMENTS DE BASE DE L'ALGORITHME

1 NOTION D'OBJET

Un algorithme ou une action manipule des données pour obtenir un résultat.
Pour cela on manipule des objets simples ou structurés.

Exemple : on calcule le quotient q et le reste r de la division entière de a par b par soustraction successives. 25-6=19 donc q=1 puis 19-6=13 donc q=2 puis 13-6=7 donc q=3 puis 7-6=1 donc q=4 puis 1-6 impossible donc q=4 et r=1.

Un objet va être caractérisé par :

- ⇨ un identificateur (son nom) : pour le désigner cet identificateur doit être parlant : q=quotient…
- ⇨ Un type (nature de l'objet : entier, caractère…) simple ou structuré. Un type détermine en particulier les valeurs possibles de l'objet et les opérations primitives applicable à l'objet.
- ⇨ Une valeur (contenu de l'objet) unique. Cette valeur peut varier au cours de l'algorithme ou d'une exécution à l'autre : ces objets sont des variables.

Dans les cas contraires (valeur fixe) ce sont des constantes. Tous les objets manipulé par un algorithme doivent être clairement définis :
Mots clefs => const : PI=3.14
=> var a, b : entiers
x, y : caractères
a, b, x, y sont des identificateurs

2 ACTIONS ELEMENTAIRES :

Actions élémentaires : opérations simple, directement utilisable.

2.1 Affectation

Permet de donner une valeur à une variable :
A<=28 « reçoit » Si A avait une valeur auparavant, cette valeur disparaît : elle est écrasé par 28

Format général :
<id_variable> <= <expression> A<=28+13
A « l'exécution » : l'expression est évaluée (calculée) et sa valeur est rangée dans la variable. Donc les types <id_variable> et <expression> doivent être compatibles.

Attention : A<=B+3
B doit avoir une valeur. Or au début d'une action, les variables ont une valeur indéterminée/ B doit avoir été initialisé.

2.2 Les opérations en entrée sortie

Elles permettent de récupérer une valeur venant de l'extérieur (lecture) ou de transmettre une valeur à l'extérieur (écriture) : var A, B, C : entiers plus rapide : lire (A, B)

Lire (A, B) écrire (A+B)

C<=A+B

Ecrire C

Remarques :

La notion de littéral : A<=28

28 est un objet caractérisé par son type (entier [numérique]), son nom (sa valeur), et sa valeur (28)

Le littéral est l'objet « constante », le nom est sa valeur

On note les littéraux de type caractère entre quotte 'A'.

On note les littéraux de type chaîne de caractères entre double quotte : ''bonjour''

Autre remarque : Lors de l'écriture d'un algorithme, éviter les détails de programmation au niveau des entrées sorties : par exemple on écrira : écrire (A, B, C)

Et non écrire (''Le produit de'', A, ''par'', B, ''vaut'', C)

3 LES STRUCTURES DE CONTROLE CONDITIONNELLES :

Une action décrit un enchaînement d'actions élémentaires. L'enchaînement est décrit par les structures de contrôle.

Une structure de contrôle déjà vue : l'enchaînement séquentiel lire (A, B)

C <= 2*A + B

La plupart des autres structures de contrôle utilise la notion de condition (expression booléenne) :

Une condition a une valeur qui est, soit vrai, soit fausse.

Pour déterminer la réalité de cette valeur on utilise :

- les opérateurs de comparaisons =, ≤, ≥, ≠
- les opérateurs booléens (logique) : ET, OU, NON

3.1 L'alternative SI-ALORS-SINON

Elle permet d'effectuer tel ou tel traitement en fonction de la valeur d'une condition.

Syntaxe :

Si <condition>

Alors < action _alors>

Sinon < action _sinon>

Exemple : Lire (note)

Si note ≥ 10

Alors écrire ''Bravo''

Sinon écrire ''Désolé''

Remarque, la ligne *Sinon <action_sinon>* est facultative.

Principe de fonctionnement :

1 : la condition est évaluée

2 : Si la condition a la valeur vrai on exécute <action_alors>

Si la condition a la valeur fausse on exécute <action_sinon>

Remarque :

Les <action_alors> ou <action_sinon> peuvent être soit :

- des actions élémentaires
- des composées (bloc)

Dans ce cas on utilise les structures imbriquées.
Exemple de structure imbriquée:
Si A≥10
Alors début
Si A≤10
Alors écrire (''oui'')
Fin
Sinon écrire (''non'')

3.2 Structure à choix multiples SELON-QUE :

Exemple :
Selon que
Note ≥ 16 : écrire (''TB'')
Note ≥ 14 : écrire (''B'')
Note ≥ 12 : écrire (''AB'')
Note ≥ 16 : écrire (''Passable'')
Sinon : écrire (''ajourné'')
Fin selon

Fonctionnement :
1 : la condition 1 est évaluée :
Si la condition 1 est vraie, alors on exécute l'action correspondante et on quitte la structure selon-que
Si la condition 1 est fausse , on évalue la condition 2…et ainsi de suite.
Si aucune n'est vraie on effectue l'action sinon.

Syntaxe :
Selon que
<condition 1> : <action 1>
<condition 2> : <action 2>
…
<condition n> : <action n>
sinon : <action_sinon>
fin selon

Remarque : en programmation, cette structure peut exister mais avec une forme ou un fonctionnement éventuellement différent. Si elle n'existe pas, il faut se souvenir que, en fait, SELON QUE est un raccourci d'écriture pour des SI imbriqués.

4 ACTIONS ET FONCTIONS :

Un algorithme est composé d'un ensemble fini d'actions. En fait, on distingue :
- les actions qui réalisent un traitement (lecture d'un complexe, tri du fichier étudiant)
- les fonctions qui effectuent un calcul et retournent un résultat

En programmation, on trouve parfois cette distinction de façon explicite. Comme en Pascal, ou l'on a procedure et functions. En revanche, en C ou C++ cette distinction n'existe pas, il n'y a que des fonctions.

En algorithme, on s'avisera de faire cette distinction entre fonction et action, pour la simple raison que chacune ne fait pas intervenir le même type de variable ou de paramètres :

4.1 Syntaxe d'une fonction :

```
Fonction <nom_fonction> ( <liste des paramètres> ) : <type de résultat>
< déclaration des objets locaux à la fonction>
début
{ corps de la fonction}
retourner résultat (en général vers une fonction ou action principale)
fin
```

4.2 Syntaxe d'une action :

```
Action <nom_action>
< déclaration des objets locaux à l'action>
début
{corps de l'action}
fin
```

4.3 Exemple de fonction :

```
Fonction périmètre rectangle (largeur, longueur : entiers) : entier
Début
Retourner [ 2*(largeur+longueur)]
Fin
```

4.4 Un exemple de fonction et d'action qui fait appel à une fonction :

Soit la fonction max3 qui donne le maximum de trois entiers :

```
Fontion max3 (x, y, z : entiers) : entier
{cette fonction détermine le max de trois entier}     <= Remarque : on peut insérer des
commentaires entre { }
var : max :entier
début
si x < y
alors   début
        si z<y alors max <= y
        sinon max <=z
        fin
sinon   début
        si x < z alors max <= z
        sinon max <= x
        fin
retourner (max)
```

fin

Maintenant, on peut créer l'action qui détermine le maximum et le minimum de trois entiers, en faisant appel à la fonction max3 :

Action max&min
Var : a, b, c, min, max : entiers
Début
Lire (a, b, c)
Max <= max3 (a, b, c)
Min <= -max3 (-a, -b, -c)
Ecrire (min, max)
Fin

Remarques :

X, y, z sont les paramètres formels de la fonction max3.
Ce sont des paramètres d'entrés : lors de l'appel de la fonction max3, les valeurs des arguments d'appel (ici : a, b, c) ou (-a, -b, -c)) sont transmises aux paramètres x, y, z en fonction de leur ordre.

Les arguments sont des expressions (par exemple : max <= max3 (2*a+b, c-b, a*c)) qui sont évaluées à l'appel. Leur valeur est transmise aux paramètres.
Naturellement, le type des expressions doit être compatible avec le type des paramètres.

5 STRUCTURES REPETITIVES :

Idée : répéter un ensemble d'opérations, arrêter la répétition en fonction d'une condition

5.1 La structure tant que :

Syntaxe :
Tant que <condition>
Faire <action>
Cette action peut être simple ou composée

Exemple :
I<=5
Tant que i≤5
Faire   début
        Ecrire (i*i)
        I<=i+1
Fin

Fonctionnement :
1 : la condition est évalué
2 : si la condition est fausse : c'est fini, on quitte le tant que
3 : si la condition est vraie, on exécute le contenu du tant que puis on remonte à l'étape 1 tester de nouveau la condition.

Remarque :
Le contenu de la structure tant que peut ne jamais être exécuté. Donc cette structure permet en réalité de répéter un traitement, 0, 1 ou plusieurs fois.
La condition étant évaluée au début, les variables étant utilisée dans la condition doivent avoir été initialisées.
On doit s'assurer de la terminaison (sinon le programme ne se termine jamais)
Pour cela, il faut nécessairement que dans le corps de la structure, la condition soit modifié quelque part.

Faisons l'analyse d'un programme écrit avec une structure tant que :

```
Action Division entière
{détermine et affiche le quotient et le reste de la division de 2 entiers}
Var : a, b, q, r : entiers
Début :         Lire (a, b)
        r<=a
        q<=0
tant que r≥b
faire   début
        q<=q+1
        r<=r-b
        fin
écrire (q, r)
fin
```

Faire tourner à la main pour vérifier si le programme fonctionne. Par exemple vérifions qu'il marche pour la division de 15 par 6 :

Instructions	a	b	q	r
Lire (a,b)	15	6		
r<=a			0	15
r≥b vrai				
q<=q+1			1	
r<=r-b				9
r≥b vrai				
q<=q+1			2	
r<=r-b				3
r≥b faux				
écrire			2	3

On peut se poser la question de la terminaison :
En effet si b<0 alors problème : il faut rajouter des conditions.
De plus, si b=0 la division n'est pas définie.
Donc cet algorithme ne marche que pour 2 entiers a et b lus au clavier tels que $a \geq 0$ et $b > 0$

5.2 Structure répéter :

Syntaxe :
Répéter
<actions simples>
jusqu'à <condition>

Fonctionnement :
1 : on exécute le corps
2 : on évalue la condition
3 : si Vraie : on quitte le répéter
4 : si Fausse on recommence

Remarques :
Il y a toujours au moins une exécution du corps. La structure répéter permet de répéter un traitement 1 ou plusieurs fois.
Pour choisir entre répéter et tant que il faut se poser la question : faut-il éventuellement ne jamais faire le traitement ? Si oui : il faut utiliser tant que, sinon utiliser la structure répéter qui exécute au moins une fois l'action.

Attention, en C++ :
La structure est do…while : c'est à dire Faire…tant que.
Alors que la structure algorithmique est répéter…jusqu'à.
C'est à dire qu'en C++ on exécute l'action tant que une condition est vrai alors qu'en algorithme on exécute une action tant que le condition est fausse, c'est à dire jusqu'à ce que la condition inverse soit vraie.

5.3 Structure pour :

Elle permet de parcourir un intervalle en répétant un traitement pour chacune des valeurs de cet intervalle.
Exemples :
1) Pour I de 1 à 5 faire Ecrire (I*I)
2) Pour I de 1 à 11 par pas de 3 faire Ecrire (I*I)

Syntaxe :
Pour <id_variable> DE <val_inférieure> A <val_supérieure>
    [par pas de <val_pas>]   <= facultatif
    Faire <actions>

Les actions peuvent être simples ou composées.

Fonctionnement :
1 : Automatiquement, on a id_variable <= val_inférieure
Donc, on a pas besoin d'initialiser, la structure se charge de la faire
2 : id_variable > val_supérieure ?
    Si oui, alors STOP, on quitte la structure
    Sinon :
- on exécute le programme
- automatiquement, l'incrémentation se fait (+1 ou +pas si l'on a définit un pas particulier, par défaut, le pas est 1)

- on remonte au début du 2 tester la condition id_variable > val_supérieure ?

Remarques :
Il est possible que l'action ne soit jamais exécutée.
Il est cependant possible d'avoir un intervalle inversé à condition d'avoir un pas négatif.
IMPORTANT : Il est absolument interdit de modifier <id_variable>, <val_inférieure>, <val_supérieure>, <val_pas> dans le corps de boucle.
Parfois cette structure n'est pas présente dans un langage de programmation, il faut donc retenir que ce n'est qu'un raccourci pour écrire des tant que.

Utilisation du POUR :
On s'en sert dès que l'on connaît au début de la boucle le nombre de répétitions à effectuer.
Dans les cas contraire, on utilisera des TANT QUE ou des REPETER

6 STRUCTURES DE DONNEES : LES TABLEAUX :

Exemple de tableau de 5 entiers :

	0	1	2	3	4
T	7	32	-8	19	-3

T signifie que c'est un objet de type tableau.
Les numéros en indices 0, 1, 2, 3, 4 correspondent aux valeurs colonnes.
Le contenu de T : les 5 entiers (dans un certain ordre)
La première valeur est T[0] ou 0 correspond donc à l'indice de la première colonne.

Déclaration d'un tableau dans un algorithme :
Ctaille est la constante qui indique le nombre de case du tableau.

Const Ctaille=25
Var Tab : tableau [Ctaille] d'entiers
Pour I de 0 à Ctaille – 1
Faire Lire tab [I]
{Cet algorithme permettra notamment de rentrer les variables dans le tableau.}

Pour créer une fonction tableau :

Fonction Mintableau (T :tableau d'entiers, N entier) : entier
{pour créer la fonction Mintableau qui retournera par exemple le minimum d'un tableau, on désigne le tableau par T et N par le nombre de colonnes, ainsi, même si l'on a déclaré auparavant un tableau à 50 colonnes et que l'on n'utilise en fait que 30 colonnes, N=30 permet de définir à l'ordinateur le nombre de colonnes réellement utilisées et limiter la durée du traitement. N est donc indépendant de Ctaille}

Exemple : déterminer si un tableau est trié :

5	8	8	12	15	3	2	1

Le nombre n'est pas toujours fixé donc pas de structure POUR.

6.1 Première proposition

```
Fonction : est trié (T : Tableau [ ] d'entiers, N : entier) : booléen
Var : I : entier
Début :
I <= 0
Tant que (I < N-1) et (T[I] ≤ T[I+1]) faire I<=I+1
Retourner (I=N-1)
Fin
```

Remarque :
Il y a un problème qaudn I atteint le niveau N-1 : on évalue la condition I<N-1 et (T[I] ≤ T[I+1]) or la case T[I+1] n'existe pas. Or la première partie (I<N-1) renvoie faux : mais certains langages évalue quand même la deuxième condition (ce n'est pas le cas du C++) : ce langage n'est pas universel, donc on ne peut pas l'appliquer en algorithmique.

6.2 Deuxième proposition :

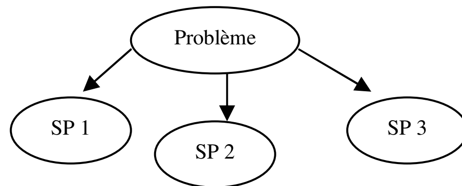
On utilise une variable booléenne :

```
Fonction : Est trié
Var : I entier
Trié : booléen
Début
I<=0   trié<=vrai
Tant que (I<N-1) et trié
Faire
        Si T[I]≤T[I+1] alors I<=I+1
        Sinon trié<=faux
Retourner trié
Fin
```

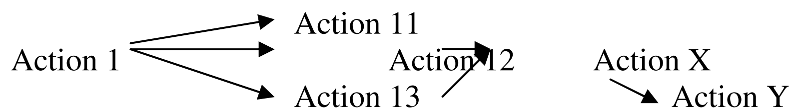
LES ACTIONS

1 La notion d'actions

Idée : un algorithme est un ensemble d'actions et de fonctions.
Méthode : on utilise la méthode d'analyse descendante.



L'algorithme : ensemble d'actions, certaines en appelant d'autres
Schéma :



Interêt :

- Simplification
- Possibilité de se répartir le travail à condition de :
  - Bien spécifier le rôle de chaque action
  - Bien spécifier les communications entre action (échange de valeur des paramètres)

2 Notion de paramètres

Il existe trois catégories de paramètres :

- Les paramètres d'entrée : valeur fournie par l'action appelante et utilisée par l'action appelée.
- Les paramètres de sorties : valeurs calculées par l'appelée et transmises vers l'appelante.
- Paramètres d'entrée/sorties : valeurs fournies par l'appelante, modifiées par l'appelée et renvoyés à l'appelée

Ces paramètres doivent être clairement spécifiés

Syntaxe :
Action <nom_action> ({E ou S ou ES} <nom_paramètre> : typepar ; …)

Remarque :

- Les paramètres permettent la communication 'interne' ente les actions.
- Pour la communication externe (lecture écriture) on utilise des actions particulières dédiées à la lecture ou à l'écriture :
  - Action de lecture : valeurs lues sont renvoyées comme paramètres de sorties (on peut avoir des paramètres d'entrée : par exemple : le nombre de case d'un tableau)
  - Action d'écriture (que des paramètres d'entrés) : écriture des valeurs fournies aux paramètres d'entrée.

IMPORTANT : ne jamais mélanger lecture et traitement ou traitement et écriture.

ATTENTION : en C++ on trouve les notions de paramètres d'entrées et de paramètres d'entrée/sorties mais pas les paramètres de sorties (un paramètre de sortie peut donc être vu comme un paramètre d'entrée/sortie dont on n'utilise pas la valeur d'entrée.)

3 Appel Actions – passages de paramètres

Exemple :
DivisionEntière (X, Y, A, B)
Un appel d'action est une action élémentaire (instruction)
Remarque :

- Le nombre d'arguments doit correspondre au nombre de paramètres.
- Lien argument paramètre : se fait selon l'ordre de la liste.

3.1 *Que ce passe-t-il lors du passage de paramètres :*

- En ASD :
  1. Au moment de l'appel, les valeurs des arguments d'entrée ou d'entrée/sortie sont recopiées dans les paramètres.
  2. L'action appelée s'exécute.
  3. A la fin de cette exécution, les valeurs des paramètres S ou ES sont recopiées dans les arguments.
- En C++ :
  1. Pour les paramètres d'entrée : idem ASD (passage par valeur)
  2. Pour les paramètres d'entrée/sortie : passage par référence :
     Chaque paramètre d'entrée/sortie possède une zone mémoire permettant de stocker une adresse mémoire.
     a) Au moment de l'appel :
     - Recopie les valeurs des arguments d'entrée
     - Recopie l'adresse des arguments d'entrée/sortie dans la zone des paramètres

     b) La fonction appelée s'exécute mais en travaillant directement sur la zone mémoire contenant la valeur des arguments ES.
     c) A la fin de l'exécution de l'appelée : rien ! (les arguments possèdent déjà les valeurs.)

3.2 *Remarque :*

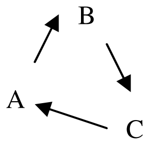
- Les arguments d'entrée peuvent être des variables ou des expressions (car la valeur est transmise).
- Les arguments S ou ES doivent être des variables.
- Une action ne doit jamais modifier ses paramètres d'entrée. Pourquoi ? Car suivant le langage de programmation, ils seront ou non modifiés.

4 La notion de fonction :

Certaines actions n'ont que des paramètres d'entrée et un seul paramètre de sortie. Dans ce cas, l'utilisation est plus souple sous forme de fonction.

5 La récursivité

Définition : un algorithme est récursif dès qu'il y a un circuit dans le graphe des appels.



récursif≠itératif

5.1 *Exemple de fonction avec récursivité : la fonction factorielle :*

n!=1*2*…*n
0!=1
n!=n*(n-1)! : cette dernière écriture est une définition récursive : on utilise (n-1)! Pour définir n!
Elle permet une traduction immédiate :

Fonction FACT (n :entier) : entier
Début
    Si n=0 alors retourner (1)
    Sinon retourner (n*FACT(n-1))
Fin

Si l'on fait le graphe des appels on se rend donc bien compte que la fonction FACT appelle la fonction FACT : donc elle est récursive.
Mais faisons tourner cet algorithme sur un exemple : n=3
Calcul de FACT(3) : met en suspens FACT(3) et va chercher FACT(2)
    Calcul de FACT(2) : met en suspens FACT(2) et va chercher FACT(1)
        Calcul de FACT(1) : met en suspens FACT(1) et va chercher FACT (0)
        FACT(0) connu (=1) : il remonte a FACT(1)
    FACT(1) connu (=1) : il remonte a FACT(2)
FACT(2) connu (=2) il remonte a FACT(3)
FACT(3) connu (=6)
Fin du programme

Remarque : toute fonction récursive doit donc nécessairement contenir un cas ou il n'y a pas d'appel récursif et ce cas doit toujours être atteint.
Autre remarque importante : la fonction FACT est plus performante dans sa fonction itérative (avec une structure Pour) car même si le nombre d'opérations est identique, choisir d'utiliser la fonction récursive demande une mémoire et temps plus important pour les traitements.

5.2 *Autre mauvais exemple*

Prenons un autre exemple pour montrer que l'option récursivité n'est pas a utiliser n'importe comment car parfois plus facile elle rend les calculs très longs.
Faisons un programme qui calcule le nombre de Fibonacci : 0 1 1 2 3 5 8 13 21
C'est en réalité la suite définie par :

```
f(0)=0
f(1)=1
f(n)=f(n-1)+f(n-2)
```

```
fonction FIB (n : entier) : entier
début :
        selon que
        n=0 : retourner (0)
        n=1 : retourner(1)
        sinon retourner (FIB(n-1)-FIB(n-2))
        fin selon
fin
```

Si ce n'est pas un cas simple on remarque donc qu'il y a deux appels de fonctions a chaque fois. Le nombre d'appel est de l'ordre de 2^n . De plus, il y a beaucoup de re-calcul (FIB(a) peut être calculé plusieurs fois pour déterminer FIB(n))

Voyons la version itérative :

```
Fonction FIB (n : entier) : entier
Var : A, B, I : entiers
Début
        A<=0
        B<=1
        Pour I de 1 à n faire
                B<=A+B
                A<=B-A
        Retourner(A)
Fin
```

Cette méthode est plus efficace : pour n=100, il y a 200 calcul à faire (et pas 2^{100} comme dans la méthode récursive !!).

5.3 Un bon exemple ??

Les deux exemples précédents sont de complexités linéaires : c'est à dire que pour connaître la réponse pour 2n, il faut deux fois plus de temps que pour connaître la réponse pour n. (le temps de calcul est proportionnel à la taille de n).
La méthode par récursivité est parfaite en revanche pour les problèmes exponentiels (c'est à dire ou le temps de calcul pour n est k fois plus important que pour n-1. Bien quelle ne soit pas parfaite pour les durée de calcul, elle sera relativement simple à écrire.

Prenons par exemple le problème des tours de Hanoï. On veut calculer le nombre de déplacement nécessaire (on peut démontrer que ce nombre est 2^n-1 ou n est le nombre de disques.)
Le but du programme : on a les tours TA, TB et TC, un compteur de disques :
On fera les déplacements : TA=>TC, TA=>TB et TC=>TB. Exemple pour 18disques : en utilisant le fait qu'on sait le faire pour 17, qui lui même sait le faire pour 16 et ainsi de suite.
On obtient un programme relativement court (qui sera traité en exercice). La version itérative est bien plus longue !!

Remarque : pour un même problème, la version récursive n'est jamais plus performante que l'itérative, mais elle est souvent plus facile à écrire et il existe des techniques (algorithmes) de « dérécursification ».

TYPES ET STRUCTURES DE DONNEES

1 Introduction

Un type de donnée détermine :

- un domaine : ensemble des valeurs possibles pour les objets de ce type.
- primitives : ensembles des opérations permettant de « manipuler » les objets de ce type.

On distingue :

- les types élémentaires : simples (entiers, caractères,…) prédéfinis dans la plupart des langages de programmation
- les types structurés : construits à l'aide de constructeurs de types (exemple : tableaux)
- les types abstraits : réservés à des structures plus évoluées (listes, files, arbres,…)

Remarque : certains types peuvent ne pas exister (ou pas totalement). Dans ce cas, il est possible de les simuler :

- coder les objets à l'aide des constructeurs existants (coder une liste par un tableau)
- écrire les primitives sous forme d'actions ou de fonctions

2 types élémentaires

2.1 LES ENTIERS

domaine [-N ; =N[ou N dépend du nombre d'octets
primitives :

- arithmétique : + - * div mod
- comparaison = ≥ ≠ <…
- fonctions particulières (ValAbs,…)

ATTENTION : problèmes de débordements (sortir du domaine)

2.2 LES REELS (NOMBRES A VIRGULES FLOTTANTES)

Domaine : un sous ensemble fini, d'un intervalle de la forme : [-N, -ε]U[0]U[ε, N]
Primitives : idem que les entiers, plus les fonctions mathématiques.
ATTENTION : les calculs sont toujours faux : ils sont approchés. Exemple : A≠B*(A/B)

En pratique : on utilise des bibliothèques implémentant, par exemple, les décimaux à 1000000 de décimales en assurant, par exemple, une précision exacte jusqu'a la $n^{\text{ème}}$. décimale

2.3 LE TYPE BOOLEEN

Domaine : Vrai, Faux
Primitives : opérateur booléenne, comparaisons
Remarque : se familiariser avec les écritures :
Tant que non fini ⇔ tant que fini = faux
Si correct alors ⇔ si correct = vrai alors
Retourner (trié) ⇔ Si trié = vrai alors retourner (vrai) sinon retourner (faux)

2.4 TYPE CARACTERES

Domaine : codés sur un octet (ASCII) voire deux octets.
Quelque soit le codage :
'a', 'b',…,'z' : sont consécutifs
'A', 'B',…,'Z' : sont consécutifs
'0',…,'9' : sont consécutifs
l'espace '_' précède les caractères alphabétiques.
Primitives :
ORD : 1 entier => le caractère ayant ce code
CHR : 1 caractère => son code

A savoir :

- Si (car≥'0') et (car≤'9') : tester un caractère chiffre
- Car<=chr(ord(car)+ord(A)-ord(a)) : convertir une minuscule en majuscule sans avoir a connaître le codage des lettres car les lettres sont codées consécutivement.
- Val<=ord(car)-ord('0') : récupère la valeur d'un chiffre.

2.5 LES CHAINES DE CARACTERES

En ASD :
Var : ch1 : chaîne [30] : longueur ≤ 30
Ch2 : chaîne : longueur quelconque
Codage en C/C++

B	o	n	j	o	u	r

1 octet = 1 caractère
Primitives :

- accès au i^{ème} car : ch(i-1)
- concaténation : +
- fonction longueur
- fonction spécifiques (cf doc)
- comparaison =, <, ≥

2.6 LES TYPES ELEMENTAIRES EN C++

2.6.1 Entiers :

- int : stockés sur un mot mémoire (dépend donc de la machine)
- short int : codés sur 2 octets (de -2^{15} à 2^{15})
- long int : codés sur 4 octets (de -2^{31} à 2^{31})
- unsigned int/short int/long int : non signés (ce qui libère un bit de plus pour le codage)

2.6.2 Réels

- float : codés sur 4 octets
- double : codés sur 8 octets
- long double : codés sur 10 octets
- unsigned foat/double/long double : non signés

2.6.3 Caractère

- char

2.6.4 Booléens

- bool : avec les valeurs 'true' ou 'false'

2.6.5 Chaînes

- tableaux de caractères :
  déclaration du tableau : char ch[25]
  déclaration dans le programme : ch='Bonjour' : il stocke un caractère par case de tableau.
- bibliothèque string :
  Lors de la programmation, il faut #include <string.h>
  Déclaration d'une chaîne de caractère : string ch ;
  Pour la longueur : ch.length ( )

3 Les constructeurs de types internes

Idée : l'utilisateur définit ses propres types
Les primitives sur ces nouveaux objets dépendent du constructeur, d'ou nécessité de rubriques spéciales de définitions de types.
En C++ on utilisera l'instruction typedef
Exemple :
Typedef int Ttab[100]   {pour définir un tableau d'au maximum 100 entiers}

Puis pour la déclaration de variable de type Ttab :
Ttab : t1, t2 ;

3.1 LE CONSTRUCTEUR INTERVALLE

C'est un constructeur qui porte sur les entiers ou les caractères
Il permet de réduire, préciser le domaine.

Exemple :
Var ch.alpha : 'a..z'
Var age : 0..120

3.2 LE CONSTRUCTEUR DE TYPE ENUMERE

Le domaine correspond à l'ensemble de noms symboliques que l'on définit soit même.

3.2.1 Exemple :

Var direction : (Nord, Sud, Est, Ouest)
…
Si direction=Nord alors…

3.2.2 En C++ :

Enum (NORD, SUD, EST, OUEST) direction ;
Ou bien :
Typedef enum (NORD, SUD, EST, OUEST) Tdirection ;
Puis pour déclarer les variables :
Tdirection direction ;

3.3 LE CONSTRUCTEUR ENTITE

But : Regrouper les éléments d'informations.

3.3.1 *Exemple :*

Type Tdate = entité (Jour : 0..31
Mois : chaîne[ ]
Année : entier)
Type Tpersonne = entité (nom, prénom : chaîne
dateNaissance : Tdate)
var d : Tdate
p : Tpersonne

On a accès au champ avec la notion de pointée :
d.jour <=8
p.dateNaissance.mois <= « Mai »

3.3.2 *Remarque :*

En ASD, on s'autorise les affections et les entrées globales, c'est à dire, dans l'algorithme :
Lire (p) : on va donc lire le nom, le prénom, la date (jour, mois, année)
Attention, ne fonctionne pas en C++. Il faut faire champ par champ !

3.3.3 *En C++*

```
Typedef struct {     int jour ;
          String mois ;
          Int année ;} Tdate ;
```

Puis déclaration des variable :

```
Tdate d ;
```

3.4 LE CONSTRUCTEUR TABLEAU

3.4.1 *En général*

Type Ttab = tableau de [TAILLE] de <type_objet>

3.4.2 *Exemple : Le tableau multidimensionnel*

Const CNBlignes=8, CNBcollones=8
Type Tpiece = (Roi, Rein »e, …, Pion, Vide)
Type TpieceJoueur = entité (p : Tpiece
C : (BLANC, NOIR))
Type Techiquier=Tableau[CNBlignes] de tablteau[CNBcolonnes] de TpieceJoueur
Var e : Techiquier

3.4.3 *Remarques*

En ASD, on s'autorise un raccourci d'écriture :
Tableau [CNBlignes, CNBcolonnes] de…

On peut définir deux catégories de tableaux :

- tableau statiques : le nombre de case est déterminé au départ (lors de la déclaration).
  Inconvénient : on a une taille maximale. Solution : simuler des tableaux de taille variable, avec une entité :

type TtabVar (t : tableau[TAILLE] de…
nb : 0..CTAILLE)

- tableau dynamique : la taille peut être modifiée en cours d'exécution.

3.4.4 *Les principaux algorithmes à connaître pour les tableaux : (tous ces algorithmes ont été vu en TD)*

3.4.4.1 Algorithme de recherche :

Trouver la position (indice) d'une valeur donnée :

- Par recherche séquentielle : si le tableau n'est pas trié, c'est la seule solution. Si le tableau est trié on peut améliorer cet algorithme car on a une condition d'arrêt supplémentaire : si T.t[i]>valeur-recherchée
  Complexité de cet algorithme : le coût en temps est proportionnel à la taille du tableau. La recherche est donc linéaire.
- Par recherche dichotomique : le tableau doit être trié et il faut aussi connaître le nombre d'éléments. Complexité de cet algorithme : Pour un tableau deux fois plus grand, on ajoute une seule étape. Le temps est donc proportionnel à $\log_2$(taille)

3.4.4.2 Algorithmes de mise à jour

- Pour un tableau non trié :
  - ajout : on peut le faire ici en fin de tableau et incrémenter le nombre de case (si possible)
  - suppression : on fait une recherche séquentielle, puis on met le dernier élément du tableau à la place (on peut, car le tableau n'est pas trié) et on décrémente le nombre de case.
  - Modification : on cherche puis remplace.
- Pour un tableau trié :
  - Ajout : on fait une recherche de la position d'insertion, on décale les valeurs à droite et on insère la valeur voulue.
  - Suppression : recherche dichotomique et si on trouve la valeur, on fait un décalage vers la gauche puis on décrémente le nombre de cases.

Remarque :
Pour l'ajout dans un tableau trié, on peu aussi commencer par les décalages et la recherche de position se fait en même temps, au fur et à mesure des décalages.

Récapitulatif :

	Nombre de comparaisons	nombres d'affectations	complexité
Recherche séquentiel + décalage	I	N-I	O(N)
Recherche dichotomique + décalage	log N	N-I	O(N)
Décalage des plus grands	N-I	N-I	O(N)

4 Les constructeurs de type externes

4.1 LES FICHIERS SEQUENTIELS

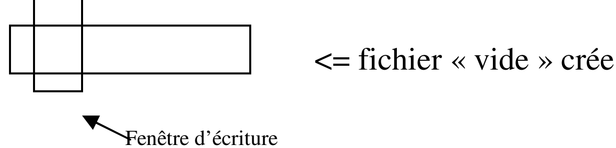
Ce sont des données stockées sur un disque, donc permanentes.
Idée : c'est une succession d'éléments d'un certain type (la structure est homogène)

4.1.1 Les primitives :

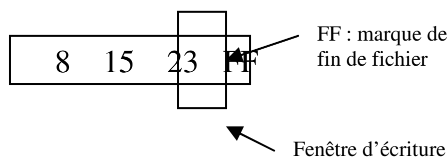
4.1.1.1 ouverture :

Elle est nécessaire avant tout accès aux enregistrements
OuvrirFichier (<identificateur>, <mode d'ouverture>)
Les différents modes d'ouvertures sont :

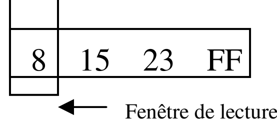
- création : crée un nouveau fichier (et écrase l'ancien si existant)
  fonctionnement :



- rajout : ouvre un fichier existant pour rajouter en fin de fichier
  fonctionnement :



- lecture : ouvre le fichier existant pour lire les enregistrements
  fonctionnement



4.1.1.2 Fermeture

Fermer Fichier (<identificateur>)
Si le fichier a été ouvert en mode création ou rajout, la marque de fin de fichier FF est rajouté automatiquement (donc, dans un algorithme, pas besoin de faire la notification d'une case fin de fichier.)

4.1.1.3 Lecture et écriture

Il faut que la lecture et l'écriture soient compatibles avec le mode d'ouverture.
LireFichier (<identificateur>, <idvariable>)
EcrireFichier (<identificateur>, <expression>)
Dans un cas ou dans l'autre, idvariable ou expression doit être du même type que les enregistrements.

Remarque :
Pour les fichiers entités, les lectures ou écritures doivent se faire enregistrement par enregistrement.

Fonctionnement :

- LireFichier : la valeur « vue » dans la fenêtre de lecture est transférée dans <idvariable> (sauf si la fenêtre voit FF) puis la fenêtre avance.
- EcrireFichier : la valeur de <expression> est inscrite dans la fenêtre d'écriture et la fenêtre avance.

4.1.2 Exemples

4.1.2.1 Premier exemple : création d'un fichier d'entiers

```
Var fic: fichier d'entiers
n : entier
Début :
Ouvrir (fic, création)
Lire(n)
Tant que n>0 faire
        Début
        EcrireFichier (fic, n)
        Lire(n)
        Fin
FermerFichier (fic)
```

4.1.2.2 Deuxième exemple : lecture et affichage du contenu

```
Ouvrir (fic, lecture)
Si EtatFichier (fic)=succès alors
        Début
        LireFichier (fic, n)
        Tant que EtatFichier (fic) ≠ FdF faire
                Début
                Ecrire(n)
                Lire(fic, n)
                Fin
        FermerFichier(fic)
        Fin
```

Retenir le schéma du squelette standard pour la lecture et le traitement de fichiers :

```
Ouvrir
Lire
Tant que non FdF faire
        Traiter
        Lire
Fermer
```

4.1.2.3 Dernier exemple : vérifier qu'un fichier est trié

```
Trié<=vrai
OuvrirFichier (fic, lecture)
LireFichier(fic, n1)
Si EtatFichier(fic)≠FdF alors LireFichier(fic, n2)
Tant que EtatFichier(fic)≠FdF et trié faire
```

Si n1>n2 alors trié <= faux
Sinon
n2<=n1
LireFichier(fic, n2)

4.1.3 Principaux algorithmes

- algorithme de sélection : à partir d'un fichier existant on crée un nouveau fichier contenant les enregistrements satisfaisants une certaine condition.
- Mise à jour : à part l'ajout en fin de fichier toute méthode est déraisonnable (car nécessite la recopie du fichier et que les fichiers séquentiels ne sont pas la bonne méthode pour conserver une information qui est sensée évoluer

Remarque :
En C++, les fichiers « typés » n'existent pas, c'est à dire que tous les enregistrements sont du même type.

4.2 LES FICHIERS TEXTE

Ce sont des fichiers de caractères qui fonctionnent comme le clavier en lecture et comme l'écran en écriture

4.2.1 Exemple

Var fict : fichier texte
LireFichier (fict, v) {v: pour tout type}
EcrireFichier (fict, a, « est le produit de », f1, « par », f2)

4.2.2 En C++

Ne pas oublier de mettre la bibliothèque fstream.h

4.2.2.1 Pour déclarer

fstream f ;

4.2.2.2 Pour ouvrir

f.open (nomfichier, ios ::in) ;
ou nomfichier est le nom physique du fichier.
In : pour entrée, out : pour sortie

4.2.2.3 Pour fermer

f.close( ) ;

4.2.2.4 Lecture et écriture

Lecture : f>>a>>b ;
Ecriture : f<<a<< « plus » ;
f.eof : booléen vrai si fin de fichier (End Of File)
f.fail : booléen vrai si échec (fail en anglais)

4.3 FICHIERS RELATIFS

Idée : les enregistrements sont « repérés » par un numéro d'ordre comme dans un tableau. On peut donc accéder directement à une information par son numéro d'enregistrement.

4.4 FICHIERS SEQUENTIELS INDEXES

On a un accès direct grâce à une clé où cette dernière fait partie de l'enregistrement. Donc, deux enregistrements différents font que l'on a deux clés différentes.

COMPLEXITE DES ALGORITHMES

0 INTRODUCTIONS, RAPPELS

Le principe est d'avoir une mesure pour évaluer les performances d'un algorithme (temps d'exécution ou espace mémoire).
Le but est de comparer les algorithmes résolvant un même problème.
La complexité en espace correspond à l'espace mémoire (nombre d'octets) nécessaire à l'exécution sur une donnée de taille N.
La complexité en temps correspond au temps d'exécution (nombre d'opérations élémentaires) de l'algorithme sur une donnée de taille N.

0.1 EXEMPLES

0.1.1 Premier exemple

Somme des entiers de 1 à n :

0.1.1.1 Solution 1

Som←0
Pour i de 1 à n faire som←som+i

Cet algorithme, en terme :

- d'espace, utilise 3 entiers (codés par exemple sur 4 octets) donc 12 octets au total (quelle que soit la valeur de n)
- de temps :
  - Affectations : 2n+1 (initialisation de la somme)
  - Arithmétique : n additions + n affectations i←i+1
  - Comparaisons : n

  Il y a donc 5n+1 opérations. C'est donc proportionnel à n, c'est à dire linéaire.

0.1.1.2 Solution 2

Som←n*(n+1)/2

Cet algorithme, en terme :

- D'espace : 2 entiers, soit 8 octets
- De temps : 4 opérations élémentaires : la complexité est constante, elle est indépendante de n.

0.1.1.3 Conclusion

La deuxième solution est la plus adaptée.

0.1.2 Deuxième exemple

Un tableau de n entiers strictement positifs et trouver une valeur inférieure à $½*\Sigma_{Valeurs}$

0.1.2.1 Solution 1

Faire le calcul de la somme des valeurs, puis rechercher une valeur inférieure à $½*\Sigma_{Valeurs}$
La complexité en temps est alors proportionnelle à n.

0.1.2.2 Solution 2

Le minimum des deux premiers entiers, sauf si les deux entiers sont égaux ou s'il n'y a qu'un seul entier.

0.2 CE QU'IL FAUT RETENIR

A un problème donné, il existe plusieurs solutions algorithmiques. Il faut les comparer par rapport à leurs complexités. Il faut s'assurer que l'algorithme proposé est le meilleur possible pour le problème considéré. En réalité, chaque problème a sa propre complexité (connue ou non). Dès qu'un algorithme à la même complexité, il est optimal.
Plus formellement : soient A un algorithme, d la donnée de cet algorithme, D_n l'ensemble de toutes les données possibles, de taille n :
Coût $_{espace}$ (A, d) (en octets) et coût $_{temps}$ (A, d) (en nombre d'opérations élémentaires) de l'exécution de A sur d.
On s'intéresse à trois types de coût :

- Le coût minimum: il correspond au meilleur des cas : $Min_n(A) = min_{d \in Dn}$ {coût (A, d)}
- Le coût maximum: il correspond au pire des cas : $Max_n(A)= max_{d \in Dn}$ {coût (A, d)}
- Le coût moyen : $Moy_n(A)=\Sigma_{d \in Dn}$ P(d)*{coût (A, d)}
  Ou P(d) est la probabilité d'apparition de la donnée d. Si les données sont équiprobables alors $P(d)=1/(Card[D_n])$

1 EXPRESSION DE LA COMPLEXITE

Supposons qu'un algorithme nécessite 3n+7 opérations élémentaires pour une donnée de taille n. Ce qui est important de retenir c'est que dans ce cas, la complexité est proportionnelle à n (de manière approchée).
En effet, si la taille est n, alors le temps est t=3n+7. Si la taille est 2n alors le temps est t'=6n+7=2t
La complexité va donc mesurer l'évolution du coût en fonction de l'évolution de la taille des données.

1.1 NOTATIONS

Soient deux fonctions f et g N➔N
On dit que f=O(g) « f est dominée par g » ⇔ A partir d'un rang n_0 $f(n) \leq Cste.g(n)$
On dit que f=Θ(g) « f et g sont asymptotiquement équivalentes » ⇔ f=O(g) et g=O(f)
En pratique, on exprime donc les complexités sous la forme Θ(g) avec g très simple.

1.2 EXEMPLES

- Θ(1) : complexité constante (exemple : la somme des entiers de 1 à n)
- (log(n)) : logarithmique (exemple : recherche dichotomique)
- Θ(n) : complexité linéaire (exemple : recherche séquentielle)
- Θ(n*log(n)) : complexité sub-quadratique (exemple : tri rapide)
- Θ(n²) : complexité quadratique (exemple : tris simples)
- $\Theta(n^k)$: complexité polynomiale
- Θ(2n) : complexité exponentielle (exemple : Tours de Hanoi)

On fera donc par exemple : $3n^3+6n^2+17n+3$ ➔ $\Theta(n^3)$
Influence de la complexité sur l'exécution :
Prenons l'exemple d'une machine qui fait une opération élémentaire en 1µs (soit 10^6 opérations par seconde)

Taille de n\Complexité	1	log(n)	n	n*log(n)	n²	n^3	2^n
100	1µs	7 µs	100 µs	0.7 ms	10 ms	1 s	4.10^{16} années
10000	1 µs	14 µs	10 ms	0.14 s	1,5 min	11.5 jours	infini

10^6	1 µs	20 µs	1s	20s	11.5jours	32.10^6années	infini

2 DETERMINATION DE LA COMPLEXITE

2.1 DANS L'ESPACE

C'est en général facile, surtout pour les données statiques : la taille est connue en fonction du type. Pour les ordres de grandeurs, seuls les tableaux comptent. La seule difficulté apparaît dans les allocations dynamiques (mémoire demandée en cours d'exécution) ou dans les algorithmes récursifs (il faut compter le nombre d'appels pour avoir une idée de la taille.

2.2 DANS LE TEMPS

- Enchaînements séquentiels : L'ordre de grandeur de la séquence correspond à la valeur maximale.
- Séquences d'actions élémentaires : $\Theta(1)$
- S'il y a un appel de fonction, il faut inclure la complexité de cette fonction dans la complexité de la séquence.
- Alternative (Si…alors…sinon) : On prend toujours la complexité maximale. Sinon on peut aussi faire une complexité moyenne mais il faut avoir des informations sur le résultat moyen de la condition.
- Boucles : Complexité = (Complexité du corps de boucle) x (Nombre de tours).
- Algorithme récursif : Complexité = (Complexité de la fonction) x (nombre d'appels).

2.3 REMARQUES

Pour un algorithme, on considère uniquement les opérations fondamentales, c'est à dire celles qui ont de l'influence sur la complexité :

- Pour un algorithme de recherche, il s'agit de comparer deux objets : proportionnel à la taille de l'objet.
- Pour un algorithme de tri : comparaisons + transferts
- Pour un algorithme de fichiers : accès aux disques

Attention aux grands nombres : un petit nombre est un type prédéfini de taille connue ou d'opérations élémentaires.

2.4 EXEMPLES : LES ALGORITHMES DE TRIS

Les algorithmes de tris ont deux types d'opérations fondamentales : les comparaisons et les transferts.

On peut distinguer deux types de tris :

- Les tris internes : en mémoire centrale :
  - Tri d'un tableau
  - Tri d'un tableau sur lui-même
- Les tris externes : tri de fichiers

2.4.1 Les tris simples

Ils sont faciles à mettre en œuvre mais leur complexité est mauvaise ($\Theta(N^2)$) et ne sont utilisables que s'il y a peu de valeurs à trier (10^4 valeurs).

2.4.1.1 La sélection ordinaire

Idée : on cherche le minimum et on le met à la case 0, puis on cherche le minimum suivant que l'on met à la case suivante et ainsi de suite. Cela correspond à l'algorithme suivant :

```
Action TriSelection (ES : T :Ttableau, E : N : entier)
Var, I, pos, posmin, : entiers
Début
        Pour I de 0 à N-2 faire
                {recherche de la position du minimum entre I et N-1}
                posmin←T[I]
                Pour pos de I+1 à N-1 faire
                        Si T[pos]<T[posmin] alors posmin←pos    {comparaison}
                Echanger (T, I, posmin)        {transfert}
Fin
```

La fait que tout le tableau soit en entrée implique qu'il y a autant de transferts que de comparaisons. On peut la trouver ainsi :

- Nombre de transferts : 3 par boucle I, faite N-1 fois donc : 3N-3 soit une complexité $\Theta(N)$. c'est donc optimal.
- Nombre de comparaisons : 1 par boucle pos, et il y a (N-1)+(N-2)+…+1=N(N-1)/2 boucles. La complexité est donc de $\Theta(N^2)$, ce qui est très mauvais.

2.4.1.2 Le tri Bulle

On compare la première case avec la seconde. Si la case 1 est plus petite que la case 2, on change de place. Puis on compare la case 2 et la case 3 et ainsi de suite.

```
Action Tri Bulle (ES : T : Ttableau ; E : N : entier)
Var : I, J, entiers        fini : booléen
Début
        I←0
        Répéter :
                Fini←vrai
                Pour J de n-1 à I+1 par pas de –1 faire
                        Si T[j]<T[j-1] alors Echanger (T, J, J-1) et fini ← faux.
                        I←I+1
        Jusqu'à (fini=vrai ou I=N-1)
Fin
```

Complexité de cet algorithme :

- Dans le meilleur des cas (tableau déjà trié) : on a N-1 comparaisons $\Theta(N)$ et 0 transferts $\Theta(1)$
- Dans le pire des cas (tableau trié à l'envers) on obtient N(N-1)/2 nombres de transferts et N(N-1)/2 nombres de comparaisons. On a alors une complexité de $\Theta(N^2)$; ce qui est très mauvais.
- En moyenne : Le calcul d'une moyenne est trop compliqué. On admettra que la complexité moyenne est de $\Theta(N^2)$.

2.4.1.3 Insertion séquentielle

Il s'agit de l'algorithme suivant :

```
Action tri_insertion (….)
Var : pos, I : entiers
Element : Tinformations
Trouvé : booléen
```

Début :

Pour I de 1 à N-1 faire :

{on cherche la position de T[i] entre 0 et i-1 de droite a gauche en décalant les plus grands}

Element←T[i]

Pos←i-1,

trouvé←false

tant que (non trouvé et pos≥0) faire

si T[pos]>Element alors T[pos+1]←T[pos] et pos←pos-1

sinon trouvé←vrai

{on range l'élément}

T[pos+1]←Element {transfert}

Fin

Fin

Complexité de cet algorithme :

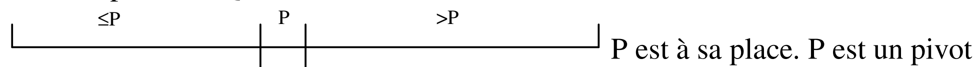
- Dans le meilleur des cas (tableau trié) la complexité du nombre de comparaisons et de transferts est de $\Theta(N)$
- Dans le pire des cas : $\Theta(N^2)$
- En moyenne : $\Theta(N^2)$

Remarques :

- Il existe une amélioration à cet algorithme : en faisant une recherche de la position d'insertion par dichotomie. La complexité du nombre de comparaisons est alors $\Theta(N\log N)$, ce qui est optimal.
- Lorsqu'on tri de gros objets, on ne peut pas éviter de les comparer mais on peut éviter de les déplacer en créant un deuxième tableau contenant les indices de l'ordre du tri du premier tableau.

2.4.2 Les tris rapides

Les tris rapides ou Quick sort. L'idée est la suivante



P est à sa place. P est un pivot

- On choisit une valeur
- Réorganiser le tableau en fonction de ce pivot
- Récursivement, on tri les 2 côtés.

Remarque : On tri des parties de tableau, il faut donc indiquer le début et la fin de la portion a traiter : les paramètres seront donc T, début, fin.

Action Tri Rapide (ES T : Ttableau, E : début, fin :entiers)

Var : pospivot : entier

Début

Si fin > début {il faut au moins deux cases !}

Réarranger (T, début, fin, pospivot)

Tri Rapide (T, début, pospivot-1)

Tri Rapide (T, pospivot+1, fin

Fin

Pour trier le tableau entier il suffit d'appeler l'action Tri Rapide (T, 0, n-1)

Qu'en est il de l'action réarranger ? Etudions tout d'abord son principe.

Comment fait-on le choix de la valeur pivot ? Au hasard, on peut prendre T[début] (première case de la portion) par exemple.

25	8	43	15	52	63	9	81
9	8	43	15	52	63	43	81
9	8	43	15	52	63	43	81
9	8	15	25	52	63	43	81

🡐 on sauve T[début] dans une variable temporaire et on cherche plus petit que T[début] depuis la droite. On peut le copier dans T[début] car la valeur a été sauvegardée. 🡐 Puis on cherche plus grand depuis la gauche et on le met dans l'ancienne case du premier plus petit trouvé. Et ainsi de suite. A la fin du traitement on a tous les plus petits d'un coté, et les plus grand de l'autre. Il ne reste plus qu'a réinsérer la valeur temporaire à sa place.

```
Action Réarranger (ES : T : Ttableau, E : début, fin : entiers S : pospivot : entier)
Var : pivot : Tinfo, G, D : entiers {indices pour la recherche}
Trouvé, cherchedroite : booléen {Si vrai on cherche à droite sinon, on cherche à gauche}
Début
        Pivot 🡐 T[début]
        G 🡐 début
        D 🡐 fin
        cherchedroite🡐vrai
        Tant que D>G faire
        |       Si cherchedroite alors
        |       |       Trouvé🡐faux
        |       |       Tant que (D>G ET non trouvé) faire
        |       |       |       Si T[d]≤pivot alors trouvé🡐vrai
        |       |       |       Sinon D🡐D-1
        |       |       Si trouvé alors
        |       |       |       T[G]🡐T[D]
        |       |       |       G🡐G+1
        |       Sinon
        |       |       Trouvé🡐faux
        |       |       Tant que (G<D ET non trouvé) faire
        |       |       |       Si T[G]>pivot alors trouvé🡐faux
        |       |       |       Sinon G🡐G+1
        |       |       Si trouvé alors
        |       |       |       T[D]🡐T[G]
        |       |       |       D🡐D-1
        |       cherchedroite🡐 ! cherchedroite
        T[D]🡐pivot
        pospivot🡐D
Fin
```

Complexité de cet algorithme

- En moyenne, cet algorithme est optimal : $\Theta(N\log N)$
- Au maximum $\Theta(N^2)$

Remarque :

Pour trouver mieux que le tri Quick Sort, il faudrait un programme qui ait une complexité moyenne de NlogN et une complexité maximale de NlogN. Cet algorithme existe : c'est le tri par pas.

2.4.3 Les tris externes

Ces tris peuvent être adaptés aux tris internes mais en utilisant 2 tableaux.

2.4.3.1 Tri balancé par monotonies de longueur 2^n.

Soit le fichier F1 : 8 15 3 9 25 32 22 6

- étape 1 : « éclater le fichier » F1 sur F3 et F4 qui seront de même taille. On éclate alternativement (c'est à dire que l'on distribue alternativement les données de F1 dans F3 et F4) on obtient les fichiers :
  F3 : 8 3 25 22
  F4 : 15 9 32 6
  On a alors construit des monotonies de longueur 1.
- étape 2 : on fusionne les monotonies de longueur pour avoir des monotonies de longueur 2, éclatées sur F1 et F2. On obtient donc les fichiers suivants :
  F1 : 8 15 25 32
  F2 : 3 9 6 22
- étape 3 : On prend les monotonies de longueur 2 pour avoir des monotonies de longueur 4.
  F3 : 3 8 9 15
  F4 : 6 22 25 32
- étape 4 : on prend les monotonies de longueur 4 pour obtenir une monotonie de longueur 8, c'est à dire le résultat final.
  F1 : 3 6 8 9 15 22 25 32

Complexité de cet algorithme :

Le nombre d'étape est logN et le coût d'une étape est de N. Donc, dans tous les cas, on a une complexité $\Theta(N\log N)$.

Remarque :

Pour transformer cet algorithme en interne, il ne faut pas oublier qu'il exige 4 fichiers. On peut les coder sous forme de deux tableaux. En codant F1 sur le tableau 1, depuis le début et F2 sur le tableau 1 depuis la fin (on code ainsi deux fichiers par tableau. C'est possible car le contenu du tableau ne varie pas.)

2.4.3.2 Idem mais…

On peut appliquer le même modèle, mais en prenant les monotonies naturelles du fichier F1 initial :

F1 : 8 15 3 9 25 32 22 6

L'avantage par rapport à l'algorithme précédent, c'est que si le tableau est déjà trié, il n'effectue pas les opérations pour rien.

Avec un tel programme de tri, on obtient une complexité optimale en moyenne ($\Theta(N\log N)$), une complexité optimale au maximum ($\Theta(N\log N)$) et optimale au minimum ($\Theta(N)$)

LES TYPES ABSTRAITS DE DONNEES

1 INTRODUCTION

Le but des données, c'est de pouvoir *faire des opérations sur ces données (primitives) et faire des spécifications sur les opérateurs (sémantique).* La manipulation de données est alors possible, donc on peut créer des algorithmes qui combinent les différents types de données définies.

Remarque : On n'a pas besoin de savoir comment, concrètement, ces données sont représentées dans l'ordinateur.

Par exemple, un entier peut être représenté par décomposition binaire, ou bien en base 10 par tableau de caractères.

1.1 UN EXEMPLE CONCRET DE TYPE ABSTRAIT DE DONNEES

La date (jour, mois, année)

Constructeur :

- Action CréerDate (S d : date)
- Action LireDate (S d : date)
- Action EcrireDate (E d : date)

Accesseurs :

- Fonction jour (E d : date) : entier
- Fonction mois (E d : date) : entier
- Fonction année (E d : date) : entier

Modification

- Action Modifie (ES d : date, E : entier)

Comparaisons

- Fonction précède (E d1, d2 : date) : booléen
- Fonction Succède (E d1, d2 : date) : booléen
- Fonction Egale (E d1, d2 : date) : booléen

Calcul

- Fonction DuréeEnJour (E d1, d2 : date) : entier

Service

- Fonction Lendemain (E d : date) : date

Remarque : Un type abstrait de données ressemble donc fortement à une classe (voir cours de C/C++). C'est d'ailleurs par des classes qu'il faudra de préférence définir et manipuler les types abstraits de données.

1.2 HISTORIQUE

Historiquement, les types abstraits de données désignent plutôt des structures abstraites de données :

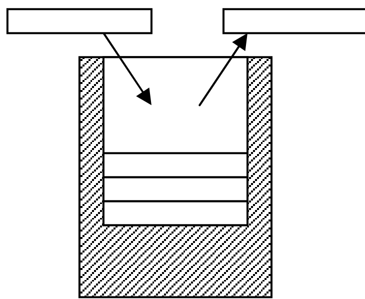
- *Structure linéaire à accès direct* : tableaux
- *Structure séquentielle* : listes (cas particulier des piles et des files)
- *Structures arborescente* : arbres binaires, arbres quelconques
- *Structures relationnelles* : graphe

- *Structures associatives à tableaux associatifs, tableaux de hachage*.

Nous allons essayer de voir tous ces types abstraits de données. Les tableaux ont déjà été abordés, voyons tout d'abord ce que sont les piles et les files.

2 Les piles

Une pile est une suite ordonnée d'éléments dans laquelle l'ajout ou la suppression d'éléments se fait toujours du même côté.



Pour tous les exercices suivants, on se définit les actions et fonctions de bases sur les piles définies sur la fiche III : Types Abstraits de Données.

2.1 Exercice 1

- a) Ecrire une fonction qui prend en entrée une pile et retourne le nombre d'éléments de cette pile.
- b) Ecrire une fonction qui prend en entrée une pile et qui sort une autre pile, identique à la pile d'entrée.
- c) Faire des variantes de b) en écrivant une action qui prend en entrée sortie une pile et qui sort une pile en prenant un élément sur deux

a)

```
Fonction NombreElement (E p : Tpile ) : entier
Var : n : entier
Début
        N=0
        Tant que ( !Pilevide (p)) faire
                N← N+1
                Dépiler(p)
        Retourner N
Fin
```

b)

```
Action clone (ES p1 : Tpile, p3 : Tpile)
Var : p2 ; p3 : Tpile
Début
        CréerPile(p2)
        CréerPile(p3)
        Tant que ( !PileVide(p1))
                Empiler (p2, ValeurSommet(p1))
                Dépiler(p1)
        Tant que ( !PileVide(p2))
                Empiler(p3, ValeurSommet(p2))
                Empiler(p1, ValeurSommet(p2)) // car p1 est en ES et on veut la récupérer dans
le même état.
                Dépiler (p2)
Fin
```

c)

```
Action UnSurDeux (ES p1 : Tpile, p3 : Tpile)
Var : p2, p3 : Tpile  cpt : entier
Début
        Cpt=0 ;
        CréerPile(p2) ;
        CréerPile(p3) ;
        Tant que ( !PileVide(p1))
                Si cpt%2=0 alors Empiler (p2, ValeurSommet(i1))
                Dépiler(p1)
                Cpt++
        Tant que ( !PileVide(p2))
                Empiler(p3, ValeurSommet(p2))
                Dépiler(p2) ;
Fin
```

2.2 EVALUATION D'EXPRESSIONS ARITHMETIQUES AVEC LES PILES

2.2.1 Expressions infixées

Dans l'expression infixées, l'opérateur (+ - * /) est placé entre ses 2 opérateurs. Le problème de l'expression infixée, c'est qu'en lisant au fur et à mesure, on n'a pas une vision des priorité sur les opérations.

Exemple : 5*3+2 ≠ 5+3*2

Pour simplifier, dans un premier temps, on peut utiliser les expressions complètement parenthésées (ECP) : Un nombre, ou une variable est une ECP. Si α est un opérateur et x et y sont des ECP alors (x*y) est un ECP.

Dans une ECP, on n'a donc plus besoin d'introduire les priorités entre opérateurs car les parenthèses définissent ces priorités.

Voir et faire tourner la fonction donnée évalECP (feuille ci jointe) sur ((((A+B)*C)/D)-(E-F)) avec l'environnement a=5 ; B=3 ; C=6 ; D=2 ; E=5 ; F=3.

2.2.2 Expressions postfixées

Une expression postfixée (EP) : Une variable, ou un nombre est une EP. Si αest un opérateur et x et y deux EP alors xyα est une EP.

Exemple : 2+3 ➔ 2.3*

Le but est de mettre au point un algorithme permettant d'évaluer une EP (comme évalECP du 2.2.1)

Remarque : Dans l'expression postfixée, l'ordre des opérateurs donne l'ordre dans lequel on doit faire les calculs (dès que l'on a un opérateur, on sait qu'il faut faire un calcul et qu'il est dans l'ordre)

Idée de l'algorithme d'évaluation :

Tant que l'on tombe sur une opérande, on empile sa valeur. Dès que l'on tombe sur un opérateur, on dépile les deux dernières opérandes, on effectue le calcul et on place le résultat dans la pile. A la fin, la seule valeur présente dans la pile est le résultat de l'expression qu'il fallait évaluer.

Exercice : faire l'algorithme de evalEP

```
Fonction evalEP ( E : EP : tableau[MAX] de caractères) : réel
Var : i entier ; vald, valg, résultat : réels ; p : Tpile)
Début
        CréerPile(p)
        I←0
        tant que EP[i]<> '%' faire
                si variable(EP[i]) alors empiler(p, ValeurSommet(eP[i])
                sinon
                        vald←ValeurSommet(p)
                        dépiler(p)
                        valg←ValeurSommet(p)
                        dépiler(p)
                        empiler(p, oper(valg, EP[i], vald)
                i++
        fin tant que
        résultat←ValeurSommet(p)
        dépiler(p)
        retourner(résultat)
fin
```

2.2.3 Passage d'une écriture infixée à une écriture postfixée

Par exemple, on veut passer de l'expression : a*b + c/(d*e) à son expression postfixée ab*cde*/+. Les opérandes apparaissent dans le même ordre dans les deux expressions, donc il n'est pas besoin de les stocker dans un temporaire. En revanche, on va stocker les opérateurs dans une pile en faisant attention de les sortir au bon moment.

Entrée EI	Sortie EP	Pile
Lit a	Ecrit a	vide
Lit *	a	Ecrit *
Lit b	Ecrit b : ab	*
Lit +	ab*	Compare + et * : * (qui est dans la pile) est plus fort, on le sort de la pile et on l'écrit dans EP, puis on met + dans la pile
Lit c	écrit c : ab*c	+
Lit /	ab*c	Compare + et / : + (qui est dans la pile) est plus faible, on ajoute alors / à la pile
Lit (	ab*c	On ajoute (à la pile.
Lit d	écrit d : ab*cd	+ / (
Lit *	ab*cd	Les deux opérandes sont identiques : on empile : + / (*
Lit e	écrit e : ab*cde	+ / (*
Lit)	écrit * : ab*cde*	On sort tous les opérateurs de la pile qui sont après (: donc ici on enlève * et on le met dans EP
	ab*cde*	puis on écrit tous les opérateurs de la pile en ne mettant pas (
	ab*cde*/+	c'est le résultat final. La pile est vide.

Exercice : Faire l'algorithme de changement d'écriture (d'une expression infixée à une expression postfixée) en vous aidant du tableau explicatif de fonctionnement si dessus.

3 LES FILES

La notion de file en algorithme correspond à la notion de file d'attente. Une file est donc un ensemble d'éléments ordonnés. Toute insertion se fait à la fin de la file, et toute suppression se fait en début de file. Seule la valeur en début de file est accessible.

Une file est donc une structure First In – First Out (FIFO)

3.1 LES PRIMITIVES

Voir la fiche ASD III : « les types abstraits de données »

3.2 UN EXEMPLE

Ecriture d'une action qui retourne la file inverse d'une file passée en entrée sortie.

```
Action FileInverse (ES F : Tfile)
Var : p : Tpile
Début
        CréerPile (p)
        Tant que ( !PileVide (p))
                Empiler (p, valeurPremier(F))
                Défiler (F)
        Tant que ( !PileVide(p))
                Enfiler (F, valeurSommet(p))
                Dépiler (p)
Fin
```

4 LES LISTES

4.1 GENERALITES

Une liste est une suite d'éléments placés dans un certain ordre. (exemple : (1, 3, 2, 5) (3, 1, 2, 5) (1, 2, 3, 5) sont 3 listes différentes. On peut ajouter ou supprimer un élément en n'importe quelle position dans la liste avec conservation des positions respectives des autres éléments.

On se donne un moyen de parcourir la liste : pour savoir quel est le premier élément, ou bien, étant donné un élément, savoir aller au suivant.

4.2 DEFINITIONS FORMELLES

Une liste est soit la liste vide (notée '( )'), soit un couple composé d'une valeur et d'une liste (notée '(valeur, liste)'). Donc, par exemple, la liste (1, 3, 2, 5) est notée en notation formelle : (1, (3, (2, (5, ( ))))).

Une liste est composé d'un premier élément et d'une sous liste qui est la liste des suivant (elle même composée d'un premier élément et d'une sous liste, et ainsi de suite).

4.3 PRIMITIVES

Voir la feuille ASD III sur les types abstraits de données.

4.3.1 En quoi les primitives peuvent être utiles ?

- Pour savoir si une liste (ou une sous liste) est vide

- Etant donné une liste, on peut obtenir la valeur du premier élément, ou bien obtenir la liste des suivants
- L'insertion en-tête d'un élément se fait par création d'une nouvelle liste à partir d'un couple.
- L'insertion en milieu de liste se fait par remplacement de la liste des suivants

4.3.2 Primitives de manipulation de listes

Pour cela, on se donne des primitives de manipulation de liste. Voir fiche ASD III types abstraits de données.

4.4 EXEMPLE

Affichage des éléments d'une liste :

```
Action Affichage (E L : Tliste)
Var : Adr : Tadresse
Début
        Adr 🡐 AdressePremier(L)
        Tant que Adr != NULL faire
                Ecrire (ValeurElement(L, Adr))
                Adr🡐AdresseSuivant(L, Adr)
Fin
```

4.5 EXERCICES SUR LES LISTES

4.5.1 Exercice 1

On suppose qu'on a en entrée une liste de TEtudiants, dont un champ s'appelle note. Calculer la moyenne de la classe (on suppose que la liste est non vide).

```
Fonction Moyenne (E L : Tliste) : réel
Var :   Adr : Tadresse
        Som, cpt : réel
        Etd : TEtudiants
Début
        Adr🡐AdressePremier(L)
        Som🡐0
        Cpt🡐0
        Tant que Adr != NULL faire
                Etd🡐ValeurElement(L, Adr)
                Som🡐Som+Etd.note
                Cpt++
        Retourner (Som/Cpt)
Fin
```

4.5.2 Exercice 2

On suppose qu'on a en entrée une liste triée par ordre croissant d'entiers. Ecrivez l'action qui rajoute un entier donné en paramètre à cette liste de façon à ce qu'elle soit encore triée après insertion.

```
Action InsertTrié (ES L : Tliste, E Nb : entier)
Var : Adr, Adrpr : Tadresse
Début
```

```
	Adr←AdressePremier(L)
	Si (Adr==NULL ou ValeurElement(L, Adr)≥n)
		InsérerEnTete(L, Nb)
	Sinon
		Tant que (AdresseSuivant(L, Adr) ≠ NULL et n > ValeurElement(L, 
AdresseSuivant(L, Adr)) )
			Adr←AdresseSuivant(L, Adr)
		InsérerAprès(L, Nb, Adr)
Fin
```

4.5.3 Quelques astuces pour des algorithmes simples avec les listes

Pour retourner l'adresse de l'élément au milieu de la liste, on déplace deux adresses, d'adresse en adresse, avec une adresse qui se déplace une fois sur deux… Ainsi, quand l'une des adresses arrive à la fin de la liste, l'autre adresse est au milieu. On a alors l'adresse de l'élément au milieu.

Si la liste est triée (des réels par exemple) on obtient donc (en un seul passage de liste et sans avoir à connaître la longueur de la liste) la médiane de la liste.

Pour l'exercice 2 : dans ce type d'exercice, on est obligé de vérifier une condition, suivant que l'on est en début de liste ou non. Pour éviter cela, on peut insérer en début de liste un élément fictif, puis effectuer les opérations à partir du deuxième élément. En fin de calcul, on supprime cet élément fictif.

4.5.4 Exercice 3

Proposer un algorithme qui fait la concaténation de deux listes (en entrée) retournant une liste (en sortie). Par exemple, si l'on a les listes (3, 7, 2) et (5, 4) on obtient la liste (3, 7, 2, 5, 4).

Proposer ensuite un algorithme séparant une liste en deux. Etant donné une liste en entrée, un élément sur deux va dans la première liste et un élément sur deux va dans la deuxième liste.

Faire une action de fusion de listes supposées triées par ordre croissant en une troisième liste triée.

Faire, en utilisant au choix les trois actions précédentes une action de tri fusion qui étant donné deux listes en entrée (pas forcément triées) retourne une liste triée.

```
Action concaténation (E L1 : Tliste, E L2 : Tliste, S L : Tliste)
Var : Adr, AdrS : Tadresse
Début
	CréerListe (L)
	InsérerEnTete (L, '%')
	AdrS ← AdressePremier(L)
	Adr← AdressePremier(L1)
	Tant que Adr ≠NULL faire
		InsérerAprès (L, ValeurElement(L1, Adr), AdrS)
		Adr← AdresseSuivant (L1, Adr)
		AdrS← AdresseSuivant (L, AdrS)
	Adr←AdressePremier(L2)
	Tant que Adr ≠NULL faire
		InsérerAprès (L, ValeurElement(L2, Adr), AdrS)
		Adr←AdresseSuivant(L2, Adr)
		AdrS←AdresseSuivant(L, AdrS)
```

```
        SupprimerEnTete(L)
Fin

Action Séparer (E L : Tliste, S L1 : Tliste, S L2 : Tliste)
Var : Adr, Adr1, Adr2 : Tadresse ; B : booléen
Début
        CréerListe(L1)
        CréerListe (L2)
        B🡐true
        InsérerEnTete(L1, E)
        InsérerEnTete(L2, E)
        Adr1🡐AdressePremier(L1)
        Adr2🡐AdressePremier(L2)
        Adr🡐AdressePremier(L)
        Tant que Adr≠NULL faire
                Si B alors
                        InsérerAprès(L1, ValeurElement(L, Adr), Adr1)
                        Adr1🡐AdresseSuivant(L1, Adr1)
                Sinon
                        InsérerAprès(L2, ValeurElement(L, Adr), Adr2)
                        Adr2🡐AdresseSuivant(L2, Adr2)
                B🡐 !B
                Adr🡐AdresseSuivant(Adr)
        SupprimerEnTete(L1)
        SupprimerEnTete(L2)
Fin

Action Fusion (E L1 : Tliste, E L2 : Tliste, S L : Tliste)
Var : Adr1, Adr2, Adr : Tadresse
Début
        CréerListe(L)
        InsérerEnTete(L, E)
        Adr🡐AdressePremier(L)
        Adr1🡐AdressePremier(L1)
        Adr2🡐AdressePremier(L2)
        Tant que (Adr1≠NULL et Adr≠NULL)
                Si ValeurElement(L1, Adr1)<ValeurElement(L2, Adr2)
                        InsérerAprès(L, ValeurElement(L1, Adr1), Adr)
                        Adr1🡐AdresseSuivant(L1, Adr1)
                Sinon
                        InsérerAprès(L, ValeurElement(L2, Adr2), Adr)
                        Adr2🡐AdresseSuivant(L2, Adr2)
        Tant que Adr1≠NULL
                InsérerAprès(L, ValeurElement(L1, Adr1), Adr)
                Adr1🡐AdresseSuivant(L1, Adr1)
                Adr🡐AdresseSuivant(L2, Adr)
        Tant que Adr2≠NULL
                InsérerAprès(L, ValeurElement(L2, Adr2), Adr)
                Adr2🡐AdresseSuivant(L2, Adr2)
                Adr🡐AdresseSuivant(L, Adr)
```

```
	SupprimerEnTete(L)
Fin

Action TriFusion (E Le : Tliste, S Ls : Tliste)
Var L1, L2, L1b, L2b : Tliste
Début
	Si AdresseSuivant(Le, AdressePremier(Le))≠NULL alors
		Séparer (L, L1, L2)
		Trifusion(L1, L1b)
		TriFusion(L2, L2b)
		Fusion(L1b, L2b)
	Sinon
		CreerListe(Ls)
		InsérerEnTete(Ls, ValeurElement(Le, AdressePremier(Le))
Fin
```

5 LES ARBRES

5.1 PRESENTATION DES ARBRES

Un arbre peut être vu comme une liste ou chaque élément a un nombre quelconque de successeurs (rappel : une liste : 0 ou 1 successeur). On obtient alors une structure qui se ramifie de plus en plus.

Utilisation des arbres pour :

- Arborescence de fichiers, répertoires
- Arbre d'appel des fonctions
- Arbre généalogique
- Hiérarchie militaire
- Document XML
- Résultat tournois sportif

5.2 UN PEU DE VOCABULAIRE

- On appelle sommet les éléments de l'arbre.
- La racine est le seul sommet de l'arbre sans père.
- Un successeur d'un sommet s'appelle un fils.
- Un sommet sans fils est une feuille.
- Un sommet avec fils s'appelle un nœud.
- Un sommet, ses successeurs, les successeurs de ses successeurs, etc, forment un sous arbre dont la racine est le sommet en question.

Ici, on se restreint au cas des arbres binaires (0, 1, 2) successeurs par sommet. Donc :

- Chaque sommet à éventuellement u fils gauche et un fils droit.
- On distingue fils gauche et fils droit lorsqu'un sommet n'a qu'un seul fils.

Remarque : Tout arbre peut se représenter sous la forme d'arbre binaire.

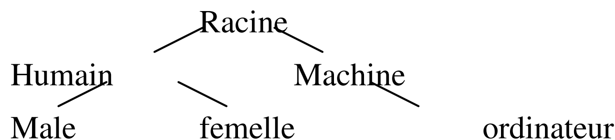
5.3 PRIMITIVES DE MANIPULATION DES ARBRES BINAIRES

5.3.1 Primitives

Voir fiche ASD IV

5.3.2 Exercice

Donner la suite d'instructions qui crée l'arbre suivant.




```
Action Arbre (S A : TarbBin)
Var : A TarbBin ; Adr : Tadresse
Début
        CréerArbre(A, « racine »)
        Adr🡐AdresseRacine(A)
        InsérerFilsGauche(A, Adr, « humain »)
        InsérerFilsDroit(A, Adr, « Machine »)
        Adr🡐AdresseFilsCauche(A, Adr)
        InsérerFilsGauche(A, Adr, « Male »)
        InsérerFilsDroit(A, Adr, « Femelle »)
        Adr🡐AdresseFilsDroit(A, AdresseRacine(A))
        InsérerFilsDroit(A, Adr, « Ordinateur »)
Fin
```

5.4 DEFINITION FORMELLE DES ARBRES BINAIRES

Un arbre binaire est un triplet (élément, sous arbre, sous arbre). Le sous arbre vide est représenté par ( ).

Pour manipuler un arbre, il suffit de savoir extraire le premier, le second, ou le troisième élément du triplet. A partir de un élément et de deux sous arbres, on peut construire le triplet qui réunit les deux sous arbres avec l'élément comme racine.

Exemple : (racine, (humain, (male, ( ), ( )), (femelle, ( ), ( ))), (machine, ( ), (ordinateur, ( ), ( )))).

5.5 PARCOURS EN PROFONDEUR DES ARBRES

Généralement, les traitements sur les arbres nécessites un parcours de tous les éléments de l'arbre binaire dans un certain ordre avec le même traitement pour chaque sommet.

Exemples : affichage des éléments de l'arbre, compter le nombre de sommet, ou de feuilles de l'arbre, mesurer la profondeur de l'arbre.

Le parcours en profondeur (à gauche) se fait de la manière suivante : à partir de la racine, on descend le plus possible en privilégiant la direction gauche. Lorsqu'on est bloqué, on re monte à la recherche d'une autre issue. Le parcours se termine lorsqu'on est revenu à la racine par le côté droit.

Propriété : Tout sommet ayant deux fils est visité trois fois. Le premier passage s'effectue avant tout ces descendants, le second passage après tous les descendants gauche et le troisième passage après tous les descendants (gauches et droits).

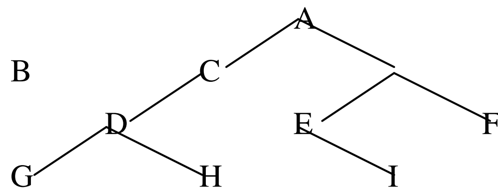
Remarque : on voit tous les sommets de la même façon en définissant des fils gauche et droit fictifs sur les sommets qui n'ont aucun ou un seul fils.

5.6 ETUDE D'UN ALGORITHME DE PARCOURS EN PROFONDEUR

```
Action Parcours (E A : TarbBin, E Adr : Tadresse)
Début
        Si Adr≠NULL alors
                Traitement1
                Parcours (A, AdresseFilsGauche(A, Adr))
                Traitement2
                Parcours (A, AdresseFilsDroit(A, Adr))
                Traitement3
Fin
```

Dans le cas que l'on étudie, on suppose que traitement1 = traitement2 = traitement3 = Affiche (ValeurSommet (A,Adr)).

Soit l'arbre suivant :

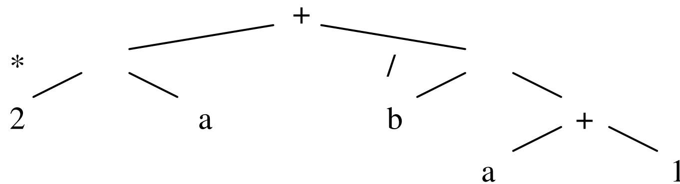


Si l'on effectue le traitement d'affichage :

- Traitement1, on obtient la trace d'affichage : ABDGHCEIF *(préfixé)*
- Traitement2, on obtient la trace d'affichage : GDHBAEICF *(infixé)*
- Traitement3, on obtient la trace d'affichage : GHDBIEFCA *(post-fixé)*

5.6.1 Expressions arithmétiques

Supposons que l'arbre code une expression arithmétique, chaque feuille est un nombre ou une variable et chaque nœud est une opération :



Qu'est ce que les parcours préfixé, infixé et postfixé affichent?

- Préfixé : +*2a/b+a1
- Infixé :2*a+b/a+1
- Postfixé : 2a*ba1+/+

5.6.2 Exercice 1

Faire une action qui affiche la forme complètement parenthésée d'une expression codée sous la forme d'un arbre.

```
Action ECP (E A : TarbBin, E Adr : Tadresse)
Début
        Si (EstUneFeuille(A, Adr)) alors
                Ecrire(ValeurSommet(A, Adr))
        Sinon
                Ecrire (‘(‘)
                ECP (A, AdresseFilsGauche(A, Adr))
                Ecrire(ValeurSommet(A, Adr)
                ECP(A, AdresseFilsDroit(A, Adr))
                Ecrire(‘)’)
Fin
```

La fonction EstUneFeuille est la fonction suivante :

```
Fonction EstUneFeuille (E A : TarbBin, Adr, Tadresse) : booléen
Début
        Retourner (AdresseFilsGauche(A, Adr)=NULL et AdresseFilsDroit(A, Adr)=NULL)
Fin
```

5.6.3 Exercice 2

Faire un algorithme permettant de compter le nombre de sommets d'un arbre binaire (fonction qui retourne un entier).

Faire un algorithme qui permet de compter le nombre de feuilles d'un arbre binaire.

L'astuce à utiliser : il suffit de reprendre l'algorithme classique de parcours en profondeur (récursivité). On remonte l'information comme valeur de retour de la fonction récursive ou comme paramètre d'entrée sortie.

```
Action Nombre_Sommet (E :A : TarbBin, E : Adr : Tadresse ; ES : Nb : entier)
Début
        Si Adr != NULL alors
                Nombre_Sommet(A, AdresseFilsGauche(A, Adr), Nb)
                Nombre_Sommet(A, AdresseFilsDroit(A, Adr), Nb)
                Nb++
Fin

Action Nombre_Feuille(E : A : TarbBin ; E : Adr : Tadresse ; ES : Nb : entier)
Début
        Si Adr !=NULL alors
                Si AdresseFilsGauche(A, Adr) = NULL et AdresseFilsDroit(A, Adr) = NULL alors
                        Nb++
                        Nombre_Feuille(A, AdresseFilsGauche(A, Adr), Nb)
```

```
                              Nombre_Feuille(A, AdresseFilsDroit(A, Adr), Nb)
Fin
```

5.7 PARCOURS EN LARGEUR

Le parcours en largeur d'un arbre peut s'effectuer avec l'algorithme suivant :

```
Action ParcoursLargeur(E :A : TarbBin)
Var : F : File de Tadresse ; Adr : Tadresse
Début
        CréerFile(F)
        Enfile(F, AdresseRacine(A))
        Tant que !FileVide(F) faire
                Adr ← ValeurSommet(F)
                Défiler(F)
                //Zone ou l'on peut faire des traitements sur l'adresse (affichage par exemple)
                Si AdresseFilsGauche(A, Adr) != NULL alors Enfiler(F, AdresseFilsGauche(A, Adr)
                Si AdresseFilsDroit(A, Adr) !=NULL alors Enfiler (F, AdresseFilsDroit(A, Adr)
Fin
```

Remarque : pour cet algorithme de parcours en largeur, on ne peut pas utiliser un algorithme récursif. Dans le parcours en profondeur, la manière récursive fonctionne, mais on peut aussi utiliser un algorithme de parcours en profondeur non récursif, et cette fois ci, au lieu d'utiliser une file, on utilise une pile.

Exercice : faire l'algorithme de parcours en profondeur non récursif.

Action profoncdeur (E : A :TarbBin)

```
Var : P : Pile de Tadresse, Adr : Tadresse
Début
        CréerPile(P)
        Empiler (P, AdresseRacine(A))
        Tant que !PileVide(P) faire
                Adr←ValeurSommet (P)
                Dépiler (P)
                Si AdresseFilsGauche(A, Adr) != NULL alors Empiler (P, AdresseFilsGauche(A, Adr))
                Si AdresseFilsDroit(A, Adr) != NULL alors Empiler (P, AdresseFilsDroit(A, Adr))
Fin
```

Propriété très importante : Tout algorithme récursif a une forme itérative. Attention, la réciproque est fausse.

5.8 ARBRE BINAIRE DE RECHERCHE

La but des arbres binaires de recherche peut être par exemple de rechercher l'adresse d'un élément dans un arbre (retourne NULL si non trouvé). Pour cela, il faut supposer que l'arbre est trié : Pour chaque sommet, le fils gauche est inférieur à la valeur du sommet, et le fils droit est supérieur à la valeur du sommet.

Idée : la recherche ne peut se poursuivre que d'un seul coté.

```
Fonction recherche (E A : ArbBin, E : v entier) : Tadresse
Var : Adr : Tadresse, trouvé Booléen
Début
        Adr←AdresseRacine(A)
        Trouvé←faux
        Tant que !Trouvé et Adr != NULL
                Si ValeurSommet (A, Adr) = v alors Trouvé = vrai
                Sinon
                        Si ValeurSommet(A, Adr) > v alors Adr ← AdresseFilsGauche(A, Adr)
                        Sinon Adr ← AdresseFilsDroit(A, Adr)
        Retourne Adr
Fin
```

Efficacité des requêtes classiques :

- Si l'arbre n'est pas équilibré, les recherches peuvent prendre jusqu'à N opérations
- Dans un arbre équilibré, la longueur des branches est de $[\log_2 N]+1$

Donc, comment réaliser un arbre de recherche équilibré ? En utilisant le théorème suivant : A tout ensemble d'actions, il existe un arbre équilibré. Comment ? On fait des rééquilibrages progressifs, par rotation de racines.