

Principes de Programmation

Syllabus destiné aux étudiants

de première année du

Baccalauréat en Technologie de l’Informatique

Rédigé par

Cécile PIROTTE

Année académique 2011-2012

# Sommaire

[Sommaire 2](#_Toc209179819)

[Organisation du cours 6](#_Toc209179820)

[Introduction 8](#_Toc209179821)

[Chapitre 1 Introduction à la programmation 9](#_Toc209179822)

[1.1. Qu’est-ce que l’algorithmique ? 9](#_Toc209179823)

[1.1.1. Définition 9](#_Toc209179824)

[1.1.2. Exemple de sensibilisation 9](#_Toc209179825)

[1.1.3. Conclusion 9](#_Toc209179826)

[1.2. Utilisateur ou programmeur ? 10](#_Toc209179827)

[1.2.1. Utilisateur 10](#_Toc209179828)

[1.2.2. Programmeur 10](#_Toc209179829)

[1.3. Qu’est-ce que la programmation ? 11](#_Toc209179830)

[1.3.1. Programmation 11](#_Toc209179831)

[1.3.2. Compilation et édition des liens 12](#_Toc209179832)

[1.3.3. Conclusion 12](#_Toc209179833)

[1.4. Comment bien programmer ? 13](#_Toc209179834)

[1.4.1. Étapes dans la résolution d’un problème 13](#_Toc209179835)

[1.4.2. Qualités requises en programmation 14](#_Toc209179836)

[1.5. Et l'ordinateur ? 15](#_Toc209179837)

[1.5.1. Les composants 15](#_Toc209179838)

[1.5.2. Exemple d'exécution 15](#_Toc209179839)

[1.6. En avant la programmation ! 16](#_Toc209179840)

[1.6.1. Notion de variable 16](#_Toc209179841)

[1.6.2. Exemple de programme 16](#_Toc209179842)

[1.6.3. Et les opérations ? 18](#_Toc209179843)

[1.6.4. Structures de contrôle 20](#_Toc209179844)

[1.6.5. Plus de convivialité … 20](#_Toc209179845)

[Chapitre 2 Les Alternatives 21](#_Toc209179846)

[2.1. Qu'est-ce qu'une alternative ? 21](#_Toc209179847)

[2.2. Qu'est-ce qu'une condition ? 21](#_Toc209179848)

[2.2.1. Booléen 21](#_Toc209179849)

[2.2.2. Condition simple 21](#_Toc209179850)

[2.2.3. Condition composée 22](#_Toc209179851)

[2.3. Les structures alternatives 23](#_Toc209179852)

[2.3.1. Alternative simple 23](#_Toc209179853)

[2.3.2. Alternative double 23](#_Toc209179854)

[2.3.3. Case 25](#_Toc209179855)

[2.4. Exemples 26](#_Toc209179856)

[Chapitre 3 Les Répétitives 37](#_Toc209179857)

[3.1. Qu'est-ce qu'une répétitive ? 37](#_Toc209179858)

[3.1.1. A quoi cela sert-il ? 37](#_Toc209179859)

[3.1.2. "Sans commencement" ou "Sans fin" 39](#_Toc209179860)

[3.2. Compter en bouclant 40](#_Toc209179861)

[3.3. Des boucles dans des boucles 43](#_Toc209179862)

[3.4. En règle générale 44](#_Toc209179863)

[3.4.1. Vérifications d'usage 44](#_Toc209179864)

[3.4.2. Un méta-diagramme d’actions 44](#_Toc209179865)

[3.5. Exemples 45](#_Toc209179866)

[Chapitre 4 Les tableaux à simple indice 49](#_Toc209179867)

[4.1. Utilité des tableaux 49](#_Toc209179868)

[4.2. Définition 49](#_Toc209179869)

[4.3. Quelques manipulations simples des tableaux 50](#_Toc209179870)

[4.3.1. Remplir un tableau avec des nombres obtenus au clavier 50](#_Toc209179871)

[4.3.2. Initialiser un tableau de nombres 52](#_Toc209179872)

[4.3.3. Afficher à l’écran le contenu d’un tableau 53](#_Toc209179873)

[4.3.4. Rechercher un élément dans un tableau 54](#_Toc209179874)

[4.4. Un peu d'organisation avec les modules 55](#_Toc209179875)

[4.4.1. Pourquoi utiliser des modules ? 55](#_Toc209179876)

[4.4.2. Qu'est-ce qu'un module ? 56](#_Toc209179877)

[4.4.3. Module de sortie et passage d'arguments 57](#_Toc209179878)

[4.4.4. Module d'obtention et renvoi de résultats 58](#_Toc209179879)

[4.4.5. Programme complet avec des modules 59](#_Toc209179880)

[4.5. Les structures 60](#_Toc209179881)

[4.5.1. Pourquoi utiliser des structures ? 60](#_Toc209179882)

[4.5.2. Qu'est-ce qu'une structure ? 61](#_Toc209179883)

[4.5.3. Tableau de structures 62](#_Toc209179884)

[4.5.4. Exemple 63](#_Toc209179885)

[4.6. Les tableaux triés 66](#_Toc209179886)

[4.6.1. Recherche simple 66](#_Toc209179887)

[4.6.2. Recherche dichotomique 67](#_Toc209179888)

[4.6.3. Décalages 69](#_Toc209179889)

[4.6.4. Insertion d’un élément 71](#_Toc209179890)

[4.6.5. Suppression d'un élément 72](#_Toc209179891)

[Chapitre 5 Les Tris 73](#_Toc209179892)

[5.1. Pourquoi trier les éléments d'un tableau ? 73](#_Toc209179893)

[5.2. Comment choisir le bon algorithme ? 73](#_Toc209179894)

[5.3. Tri par sélection 75](#_Toc209179895)

[5.3.1. Principe 75](#_Toc209179896)

[5.3.2. Première version 75](#_Toc209179897)

[5.3.3. Deuxième version 78](#_Toc209179898)

[5.4. Tri par échange ou tri à bulle 80](#_Toc209179899)

[5.4.1. Principe 80](#_Toc209179900)

[5.4.2. Exemple 80](#_Toc209179901)

[5.4.3. Algorithme 82](#_Toc209179902)

[Chapitre 6 Les fichiers 83](#_Toc209179903)

[6.1. Pourquoi des fichiers ? 83](#_Toc209179904)

[6.1.1. L'échange d'informations 83](#_Toc209179905)

[6.1.2. La mémorisation à long terme 83](#_Toc209179906)

[6.2. La notion de fichier 84](#_Toc209179907)

[6.2.1. Introduction 84](#_Toc209179908)

[6.2.2. Définitions 84](#_Toc209179909)

[6.3. Types d'enregistrements 85](#_Toc209179910)

[6.4. Représentations d'un enregistrement logique 85](#_Toc209179911)

[6.5. Types de fichiers 86](#_Toc209179912)

[6.6. Différentes organisations de fichier 86](#_Toc209179913)

[6.7. Le lien entre la représentation logique et physique 88](#_Toc209179914)

[6.7.1. Représentation interne d'un fichier 88](#_Toc209179915)

[6.7.2. L'établissement de la liaison 88](#_Toc209179916)

[6.8. Formats d'enregistrement physiques 89](#_Toc209179917)

[6.9. En programmation … 90](#_Toc209179918)

[Chapitre 7 Les Piles et les Files 91](#_Toc209179919)

[7.1. Introduction 91](#_Toc209179920)

[7.2. Les piles 91](#_Toc209179921)

[7.2.1. Qu'est-ce qu'une pile ? 91](#_Toc209179922)

[7.2.2. Algorithmes 92](#_Toc209179923)

[7.2.3. Exemples 93](#_Toc209179924)

[7.3. Les files 94](#_Toc209179925)

[7.3.1. Qu'est-ce qu'une file ? 94](#_Toc209179926)

[7.3.2. Algorithmes 95](#_Toc209179927)

[7.3.3. Exemples 97](#_Toc209179928)

[Chapitre 8 Les Tableaux à double indice 98](#_Toc209179929)

[8.1. Pourquoi des tableaux à plusieurs dimensions ? 98](#_Toc209179930)

[8.2. Définition 99](#_Toc209179931)

[8.3. Exemples d'utilisation 100](#_Toc209179932)

[8.4. Manipulations des tableaux à double indice 101](#_Toc209179933)

[8.4.1. Afficher le contenu d'un tableau à double indice 101](#_Toc209179934)

[8.4.2. Afficher le total d'une ligne 102](#_Toc209179935)

[8.4.3. Afficher le minimum d'une colonne 103](#_Toc209179936)

[8.4.4. Afficher la position d'un élément donné 104](#_Toc209179937)

[8.5. Tableau de structure ou Tableau à double indice ? 105](#_Toc209179938)

[8.5.1. Tableau de structure 105](#_Toc209179939)

[8.5.2. Tableau à double indice 107](#_Toc209179940)

[8.5.3. Conclusion 108](#_Toc209179941)

[Chapitre 9 Les listes chaînées 109](#_Toc209179942)

[9.1. Pourquoi des listes chaînées ? 109](#_Toc209179943)

[9.2. Définition 111](#_Toc209179944)

[9.3. Notations 111](#_Toc209179945)

[9.4. Fonctionnement 111](#_Toc209179946)

[9.5. Opérations sur les listes 112](#_Toc209179947)

[9.6. Exemples d'utilisation 113](#_Toc209179948)

[9.6.1. Gestion de la mémoire par listes chainées 113](#_Toc209179949)

[9.6.2. Gestion de l'accès à un fichier séquentiel 113](#_Toc209179950)

[9.6.3. Piles et files 113](#_Toc209179951)

[9.7. Exercice introductif sur les listes chaînées 114](#_Toc209179952)

[9.8. Manipulations des listes chaînées 115](#_Toc209179953)

[9.8.1. Énoncé commun 115](#_Toc209179954)

[9.8.2. Recherche d’un élément de la liste 116](#_Toc209179955)

[9.8.3. Mise-à-jour d'un élément de la liste 118](#_Toc209179956)

[9.8.4. Ajout d’un élément dans une liste 119](#_Toc209179957)

[9.8.5. Suppression d’un élément de la liste 124](#_Toc209179958)

[9.8.6. Comptage du nombre d’éléments de la liste 125](#_Toc209179959)

[Chapitre 10 Les Arbres 126](#_Toc209179960)

[10.1. Pourquoi des arbres ? 126](#_Toc209179961)

[10.2. Quelques notions de la théorie des graphes 126](#_Toc209179962)

[10.2.1. Définition 126](#_Toc209179963)

[10.2.2. Caractéristiques d'un arbre 127](#_Toc209179964)

[10.3. Exercice introductif sur les arbres 128](#_Toc209179965)

[10.4. Arbre binaire 129](#_Toc209179966)

[10.4.1. Définition 129](#_Toc209179967)

[10.4.2. Représentation en mémoire 129](#_Toc209179968)

[10.5. Arbre binaire de recherche 130](#_Toc209179969)

[10.5.1. Définition 130](#_Toc209179970)

[10.5.2. Représentation en mémoire 130](#_Toc209179971)

[10.6. Notations 131](#_Toc209179972)

[10.7. Opérations 131](#_Toc209179973)

[10.8. Exemples d'utilisation 131](#_Toc209179974)

[10.8.1. Dans la vie quotidienne 131](#_Toc209179975)

[10.8.2. En informatique 131](#_Toc209179976)

[10.9. Manipulation des arbres 132](#_Toc209179977)

[10.9.1. Énoncé commun 132](#_Toc209179978)

[10.9.2. Recherche d’un élément 133](#_Toc209179979)

[10.9.3. Ajout d’un élément 134](#_Toc209179980)

[10.9.4. Suppression d’un élément 136](#_Toc209179981)

[10.10. Parcours 138](#_Toc209179982)

[10.10.1. Introduction 138](#_Toc209179983)

[10.10.2. Récursivité 138](#_Toc209179984)

[10.10.3. Parcours d'un arbre binaire de recherche 139](#_Toc209179985)

[10.10.4. Algorithme récursif du parcours infixe 139](#_Toc209179986)

[Conclusion 140](#_Toc209179987)

[Bibliographie 141](#_Toc209179988)

# Organisation du cours

1. Horaire

Le cours de Principes de programmation occupe 75 heures de la grille horaire, dont 45 heures de théorie et 30 heures d’exercices, réparties comme suit :

Au premier quadrimestre :

* **Théorie :** 2 heures par semaine ;
* **Exercices :** 2 heures par semaine.

Au second quadrimestre :

* **Théorie :** 2 heures par semaine sous la forme de modules (pas toutes les semaines) ;
* **Exercices :** 1 heure par semaine.

1. Planning

**Chapitre 1 Introduction à la programmation**

**Chapitre 2 Alternatives**

* Simples
* Doubles

**Interro N°1**

**Chapitre 3 Répétitives**

**Interro N°2**

**Chapitre 4 Tableaux à simple indice**

* Non triés
* Triés

**Chapitre 5 Tris**

**Chapitre 6 Fichier**

**Interro Récapitulative**

**Chapitre 7 Piles et Files**

**Chapitre 8 Tableaux à double indice**

**Chapitre 9 Listes chaînées**

**Interro N°3**

**Chapitre 10 Arbres**

**Examen**

1. Évaluation

Au niveau de l'évaluation, il y a 2 interrogations au premier quadrimestre et 1 au second. A la fin de chaque quadrimestre, une interrogation récapitulative est organisée.

La **répartition des points** est la suivante :

Quadrimestre 1 : 2 interrogations 15 %

40 %

1 interrogation récapitulative 25 %

100 %

Quadrimestre 2 : 1 interrogation 20 %

60 %

1 interrogation récapitulative 40 %

Il est important de spécifier que les notions vues au cours de **Langage de programmation** (Langage C) sont basées sur celles vues au cours de Principes de programmation, ce qui permet l'application directe des algorithmes sur ordinateur.

De plus, le cours de principes de Programmation fait partie des **cours de spécialité**.

# Introduction

Le cours de Principes de programmation a pour objectif de fournir les notions de base dans le domaine de l’**algorithmique**.

Ce syllabus est le support utilisé pour dispenser les principes inhérents à la programmation. Il aborde les diverses structures de contrôle et de données utilisées dans la réalisation d'un algorithme.

Voici un aperçu de chacun des différents chapitres du cours :

* Le **chapitre 1** décrit ce qu'est l'algorithmique et ses principes de base. Il fournit également une méthode de travail constructive.
* Le **chapitre 2** présente les différents cas d'alternatives ainsi que la notion de condition.
* Le **chapitre 3** expose de manière précise la structure répétitive.
* Le **chapitre 4** concerne une structure de données plus élaborée : les tableaux à simple indice. Il est également question des structure qui permettent un autre agencement des données.
* Le **chapitre 5** s'intéresse à diverses méthodes de tri.
* Le **chapitre 6** aborde la notion de fichier et leur utilisation.
* Le **chapitre 7** donne deux exemples d'utilisation des tableaux à simple indice : les piles et les files.
* Le **chapitre 8** définit les tableaux à double indice et leur utilisation.
* Le **chapitre 9** donne un aperçu complet de ce que sont les listes chaînées.
* Le **chapitre 10** traite des arbres ainsi que de la récursivité.

Les chapitres du cours étant présentés, il est important de comprendre de quel type de matière il s'agit, afin de pouvoir mieux l'aborder.

L'algorithmique fait partie des **savoirs procéduraux**, c'est-à-dire de ce qui permet de savoir "comment il faut faire". Il s'agit en effet de connaître des procédures pour résoudre divers types de problèmes.

À la différence des savoirs de type théorique qui sont exprimés dans une forme indépendante des actions qui pourraient les utiliser, les savoirs procéduraux sont décrits en vue d'une action à réaliser. C'est la distinction entre le savoir "comment ça marche?" et le savoir "comment faire pour que ça marche?".

Connaître une procédure ou une méthode n'est pas encore savoir l'appliquer. Ce passage du savoir procédural au **savoir-faire procédural** passe par l'expérience pratique. La maîtrise d'une procédure suppose son apprentissage. Ce n'est qu'après entraînement que le savoir deviendra savoir-faire.

C'est dans ce but que le cours de Principes de Programmation est accompagné de **séances d'exercices** permettant la mise en pratique des notions vues lors des cours magistraux. Cet aspect est primordial dans l'apprentissage de cette matière. Il s'agit en effet du même type d'apprentissage que la conduite automobile, en d'autres termes : il faut conduire pour apprendre à conduire.

Le cours ayant été brièvement présenté, il s'agit maintenant d'entrer dans le vif du sujet : la programmation !

# Introduction à la programmation

## Qu’est-ce que l’algorithmique ?

### Définition

Suivre une recette de cuisine, lire un mode d’emploi, indiquer le chemin à un touriste, … Toutes ces activités sont en fait des algorithmes. L’algorithmique n’est donc pas un savoir énigmatique réservé à quelques rares initiés, c'est une **aptitude partagée par tous**.

Plus précisément, un **algorithme** est une **suite d’instructions, qui, une fois exécutée correctement, conduit à un résultat donné**.

Si l’algorithme est exact, le résultat sera celui attendu. S’il ne l'est pas, le résultat sera aléatoire et, dans ce cas, risque de ne pas être correct.

De plus, pour fonctionner, l’algorithme doit contenir uniquement des instructions écrites dans un **langage compréhensible** par celui qui devra l’exécuter.

### Exemple de sensibilisation

Le but de cet exemple est de vous familiariser à l'algorithmique et de vous sensibiliser à son utilité.

### Conclusion

La **résolution d'un problème** nécessite une **démarche**, et le but du cours de Principes de programmation est d'apprendre à la développer par l'écriture d'un **algorithme** permettant de résoudre un problème. En suivant une démarche **rigoureuse**, tous les problèmes informatiques peuvent être résolus.

## Utilisateur ou programmeur ?

### Utilisateur

L’utilisateur utilise l’ordinateur comme un **outil**.

UTILISATEUR

TRAITEMENT

RESULTATS

(SORTIES)

DONNEES

(ENTREES)

L’utilisateur introduit des données via les périphériques d’entrées (clavier, souris, etc.), attend le traitement de celles-ci par l’ordinateur et reçoit les résultats de ce traitement via les périphériques de sorties (écran, imprimante, etc.).

**Objectif :** apprendre à se servir au mieux d’outils adaptés à une tâche donnée, c’est-à-dire connaître le mode d’emploi du programme.

⇨ ORDINATEUR + PROGRAMME = OUTIL

### Programmeur

Le programmeur utilise l’ordinateur en tant qu’**exécutant**.

PROGRAMMEUR

FAIRE FAIRE

ORDINATEUR

TACHE

COMMENT FAIRE ?

QUOI FAIRE ?

➂

➀

➁

programme

COMMENT FAIRE FAIRE ?

algorithme

**Objectif :** développer une marche à suivre (algorithme) pour rendre l’exécutant (ordinateur) capable d’effectuer un ensemble de tâches.

⇨ ORDINATEUR + ALGORITHME + LANGAGE = EXECUTANT

## Qu’est-ce que la programmation ?

En informatique, les algorithmes sont exprimés sous la forme d'un programme composé uniquement d’instructions compréhensibles par la machine. La mise en œuvre d'un programme informatique se fait en plusieurs étapes : l’analyse ➀, la programmation ➁, la compilation et l’édition des liens ➂.

Analyse

Comme il s'agit de fournir la solution à un problème, la première étape consiste à analyser le problème, c'est-à-dire à en cerner les limites et à l’exprimer dans un langage descriptif. L'**algorithme** sera le résultat de cette démarche. On parle généralement d'**analyse** pour décrire le processus par lequel le problème est formalisé.

Historiquement, plusieurs types de notations ont été utilisés pour représenter des algorithmes. Il y a eu notamment une représentation graphique, avec des carrés, des losanges, etc. qu’on appelle des organigrammes.

Aujourd’hui, cette représentation est quasiment abandonnée, pour deux raisons. Premièrement, dès que l’algorithme commence à grossir un peu, ce n’est plus pratique. Deuxièmement, cette représentation favorise le glissement vers un certain type de programmation, dite non structurée, qui n’est plus du tout au "goût du jour". C’est pourquoi on utilise généralement une série de conventions appelée "pseudo-code", qui ressemble à un langage authentique, la plupart des problèmes de syntaxe étant mis de côté.

A l'IESN, on ajoute au pseudo-code quelques éléments graphiques qui permettent une meilleure lisibilité. Cette écriture est appelée **diagramme d’actions**. On définit un diagramme d’actions de la façon suivante :

C'est un texte décrivant dans un certain langage

* une suite d’opérations (actions) portant sur des objets,
* à exécuter dans un ordre précis,
* pour résoudre un problème.

Chaque diagramme porte un nom (le plus significatif possible) et est rédigé par une personne à destination d’un exécutant.

### Programmation

L'étape suivante consiste à traduire le diagramme d’actions dans un langage de programmation spécifique, il s'agit de la phase de **programmation**.

La **programmation** est l’art d’écrire des instructions indiquant à l’ordinateur un traitement à effectuer.

Un **programme** est une séquence d’instructions à exécuter par l’ordinateur.

Le **langage de programmation** est l'intermédiaire entre l'humain et la machine, il permet d'écrire dans un langage proche de la machine, mais intelligible par l'humain, les opérations que l'ordinateur doit effectuer.

### Compilation et édition des liens

Le programme est ensuite transformé en **langage machine** lors d'une étape appelée **compilation**. La compilation est une phase réalisée par l'ordinateur lui-même grâce à un autre programme appelé compilateur.

La dernière phase s'appelle l'**édition de liens**. Elle consiste à lier le programme avec tous les éléments externes. C’est seulement après cette phase que le programme est exécutable.

### Conclusion

Le cours de **Principes de programmation** ne concerne que la première étape : l’**algorithmique**. Étant donné que l’algorithmique permet d’exprimer les instructions résolvant un problème donné indépendamment des particularités de tel ou tel langage, il est important d’apprendre à bien écrire des algorithmes avant d’apprendre à programmer. A l’inverse, il n’est pas nécessaire de connaître un langage de programmation pour analyser et résoudre un problème. Le cours de Langage de programmation (Langage C) repose donc sur celui de Principes de programmation.

Dans notre cas, l'algorithmique étant à la base de la programmation et non de l'art culinaire, nous préférerons le terme de **programmation**.

## Comment bien programmer ?

### Étapes dans la résolution d’un problème

Afin de s’assurer de la complétude d’un diagramme d’actions, nous allons définir une démarche de travail que nous appellerons **ODERIC**. Elle se compose de six étapes : l’**O**bservation, le **D**iagramme, l'**É**valuation, le **R**ésultat, l'**I**nterprétation et la **C**onclusion. Ces six étapes peuvent se traduire de la manière décrite ci-dessous.

#### Observation : lecture de l'énoncé

* Lire l'énoncé en entier une première fois afin d'avoir une vue d'ensemble du problème ;
* Lire l'énoncé une seconde fois et déterminer quelles informations sont fournies, c'est-à-dire les **données** ;
* Lire à nouveau l'énoncé de manière à détecter les informations devant être demandées à l'utilisateur, c'est-à-dire les **entrées** ;
* Lire à nouveau l'énoncé de manière à en extraire les informations concernant le(s) résultat(s) attendu(s), c'est-à-dire les **sorties**.

#### Diagramme : écrire le diagramme d’actions

* En fonction des entrées, des données et de leur organisation, déterminer les **opérations** qui devront leur être appliquées pour obtenir les sorties ;
* Sur base des entrées, des données, des sorties et des opérations choisies, déterminer les **variables** qui seront nécessaires à l'algorithme ;
* **Initialiser** les variables qui doivent l'être ;
* Écrire le **Diagramme d’actions** correspondant à ces procédures.

#### Évaluation : exécuter le programme

* Pour chaque instruction de traitement, déterminer les modifications apportées aux **valeurs des variables** en utilisant un tableau de variables ;
* Pour chaque instruction d'entrée, se mettre à la place de l'utilisateur et noter méthodiquement les **valeurs entrées** ;
* Pour chaque instruction de sortie, se mettre à la place de l'ordinateur et noter méthodiquement les **valeurs sorties**.

#### Résultat : résultat de l'exécution du programme

* Le **tableau des variables**, avec pour chaque étape du **Diagramme** d’actions, la valeur de chaque variable ;
* Les valeurs entrées et sorties.

#### Interprétation : Vérification du résultat obtenu

Vérifier que le **résultat obtenu** correspond bien à ce qui était demandé dans l'énoncé. Pour chaque élément ne correspondant pas, noter ce en quoi il diffère du **résultat attendu**.

#### Conclusion : prendre la décision qui s'impose

* Si le résultat obtenu ne correspond pas, **recommencer** la première étape en tenant compte des notes prises lors de l’**Interprétation** ;
* Sinon, le **Diagramme** d’actions est **correct**.

### Qualités requises en programmation

Le cours de Principes de programmation a pour but de vous apprendre à résoudre divers types de problèmes. A chaque type de problème est associé une démarche particulière qu'il est nécessaire de suivre pour le résoudre.

La maîtrise de l’algorithmique requiert donc deux qualités :

* **La rigueur**. En effet, chaque fois que l’on écrit une série d’instructions estimée juste, il faut se mettre à la place de la machine qui va l’exécuter pour vérifier si le résultat obtenu est bien celui recherché. Cette opération requiert simplement rigueur et méthode.
* **Une certaine intuition**. Aucune recette ne permet de savoir a priori quelles instructions permettront d’obtenir le résultat voulu. C’est là qu’intervient la forme "d’intelligence" requise pour l’algorithmique. Il semble évident que certaines personnes ont au départ davantage cette intuition que d'autres. Cependant, des réflexes peuvent s’acquérir et, dès lors, l’expérience finit par compenser ce manque d'intuition.

## Et l'ordinateur ?

### Les composants

Voici un bref aperçu des éléments composant un ordinateur et de leur interaction.

PERIPHERIQUES D’ENTREE

(1)

MEMOIRES DE MASSE (EXTERNES)

(5)

PERIPHERIQUES

DE SORTIE

(4)

UNITE CENTRALE

PROCESSEUR

(3)

MEMOIRE CENTRALE

(2)

Chacun de ces éléments a un rôle particulier dans l'exécution d'un programme :

* Les périphériques d'entrées permettent de prendre connaissance de l’information que l'utilisateur veut traiter ;
* La mémoire centrale permet de mémoriser l’information pendant l'exécution du programme ;
* Le processeur exécute le traitement sur les informations ;
* Les périphériques de sortie transmettent le résultat des traitements à l'utilisateur ;
* Les mémoires de masses ont pour but de sauvegarder les informations à long terme.

### Exemple d'exécution

L’utilisateur souhaite que l’ordinateur effectue la somme de deux nombres introduits au clavier et que le résultat soit affiché à l’écran.

UNITE D’ENTREE

L’utilisateur tape au clavier deux nombres :

7

5

MEMOIRES DE MASSE

Le résultat peut être stocké sur le disque dur

UNITE DE SORTIE

Le résultat s’affiche à l’écran :

12

UNITE CENTRALE

PROCESSEUR

Calcule la somme 7+5=12

MC

7

5 12

## En avant la programmation !

### Notion de variable

Nom

Contenu

Adresse

En mathématiques, une "variable" est généralement une inconnue, qui recouvre un nombre non précisé de valeurs. En informatique, la notion de variable est différente. En effet, dans notre cas, une variable possède à un moment donné une et une seule valeur. On peut donc la comparer à un récipient d’information.

Elle est caractérisée par :

* Son **nom** : il faut choisir un nom significatif (mnémonique) qui rappelle le contenu.
* Par convention, le nom de variables commence par une majuscule et si c’est un nom composé, on colle les différents mots (ou on les réunit par un trait d’union) avec une majuscule à chaque mot.
* Son **type** : les variables ont un contenu soit **numérique**, entier ou réel, soit **alphanumérique**, c'est-à-dire des caractères alphabétiques ou autres.
* Son **contenu** : le contenu d’une variable est la valeur qui lui est attribuée. Cette valeur doit correspondre à son type et à son nom.
* Le nom et le type d’une variable ne peuvent en aucun cas changer. Son contenu, par contre, peut être modifié en cours d’exécution du programme.

On déterminera la fonction de l’adresse dans la suite du cours.

### Exemple de programme

#### Énoncé

Soit deux nombres *Nbr1* et *Nbr2* introduits par l'utilisateur, on veut calculer et afficher la somme de ces nombres.

#### Résolution

* **Observation** 
  + Donnée : rien ;
  + Entrées : deux nombres ;
  + Sortie : la somme.
* **Diagramme** 
  + Opérations : obtention, somme, sortie ;
  + Variables : Nbr1, Nbr2;

Res ;

* + Initialisation : aucune.
  + Diagramme d'actions : Somme2Nbr

Somme2Nbr

~~Prendre connaissance du 1er nombre et retenir sa valeur dans~~

**Obtenir** *Nbr1*

~~Prendre connaissance du 2ième nombre et retenir sa valeur dans~~

**Obtenir** *Nbr2*

ou **Obtenir** *Nbr1***,** *Nbr2*

si les variables sont de même type

~~Retenir dans Res le résultat du calcul de la somme de~~ *~~Nbr1~~* ~~et de~~ *~~Nbr2~~*

*Res* **=** *Nbr1* **+** *Nbr2*

**Opérateur d'addition**

**Opérateur d'affectation**

~~Afficher la valeur de~~

**Sortir** *Res*

* **Évaluation – Résultats**
  + Obtentions :

7 ( = *Nbr1*)

5 ( = *Nbr2*)

* + Tableau des variables :

|  |  |  |  |
| --- | --- | --- | --- |
| **Instruction** | *Nbr1* | *Nbr2* | *Res* |
| Obtentions | 7 | 5 | ? |
| *Res* = *Nbr1* + *Nbr2* | 7 | 5 | 12 |

* + Sortie : 12 (Res)
* **Interprétation**

Le programme demande deux nombres à l'utilisateur, en fait la somme et l'affiche.

* **Conclusion**

Le programme fait bien ce qui était demandé dans l'énoncé.

### Et les opérations ?

Après avoir vu ce qu'est un diagramme d’actions et ce que représente une variable, il reste à préciser ce que sont les opérations qui composent ces diagrammes d'actions.

#### Opérations de lecture et d'écriture

Il existe deux opérations particulières qui permettent au programmeur de dialoguer avec l'utilisateur.

La première est **l'opération de lecture**, qui permet d'obtenir une valeur donnée par l'utilisateur via un périphérique d'entrée.

*Exemple :* **Obtenir** *Nom*

La seconde, **l'opération d'écriture**, permet d'afficher la valeur d'une variable ou un simple message à destination de l'utilisateur, et ce, via un périphérique de sortie.

*Exemples :* **Sortir** *Nom*

**Sortir** "Bonjour !"

On peut également combiner les deux.

*Exemples :* **Sortir** "Bonjour", *Nom*, "!"

#### Opération d'affectation

L'affectation permet d'attribuer une valeur (un contenu) à une variable. En reprenant la représentation d'une variable ci-dessus, l'affectation a pour objectif de remplir le récipient.

Le signe représentant l'affectation est le "**=**". Comme la notion de variable, le "**=**" de l'affectation n'a pas la même signification que l'égalité mathématique. En effet, à gauche du signe d'affectation, se trouve le nom de la variable et à droite la valeur qui lui est affectée. Cette valeur peut être une constante, le contenu d'une autre variable ou le résultat d'une expression (expliquée au point suivant).

*Exemple :* *Nbr* **=** 12

* Une condition importante est que les deux parties de l'affection soient du même type.
* Une instruction d’affectation ne modifie que ce qui est situé à gauche du "**=**".

#### Les expressions

Une expression est un ensemble de **valeurs**, reliées par des **opérateurs**, et équivalant à une seule valeur.

Comme nous venons de le voir, dans l'opération d’affectation, il est fréquent de placer une expression à la droite du "**=**".

Un opérateur est un **signe** qui relie deux valeurs, pour produire un résultat. Les opérateurs possibles dépendent du type des valeurs qui sont en jeu.

* **Les opérateurs numériques**

Ce sont les quatre opérations arithmétiques : addition (+), soustraction (-), multiplication (\*), division (/). Nous utiliserons également l'exposant (^) et le modulo (%).

*Exemple :* Cinq au carré s’écrit **5 ^ 2** (25).

Le reste de la division de 43 par 10 s'écrit **43%10** (3).

De plus, pour récupérer la partie entière d'un réel, on utilise la notation suivante : [valeur]ENT.

*Exemple :* NbrEnt = **[**NbrReel**]ENT**

NbrEnt = **[**expression**]ENT**

Enfin, on a le droit d’utiliser les parenthèses, avec les mêmes règles qu’en mathématiques. La multiplication et la division ont "naturellement" priorité sur l’addition et la soustraction. Les parenthèses ne sont ainsi utiles que pour modifier cette priorité naturelle.

* **Opérateurs logiques (ou booléens)**

Il s’agit, dans notre cas, du AND, du OR et du NOT. Nous en parlerons plus en détail dans la suite du cours.

#### Conclusion

Le contenu d'une variable peut donc être modifié au cours du programme soit par une lecture, soit par une affectation.

On peut affecter une valeur (constante), le contenu d'une variable ou la valeur résultant d'une expression.

*Exemples :* **Obtenir** *Nbr1* *Res* **=** *Nbr1*

*Nbr2* **=** 5 *Res* **=** *Res* \* *Nbr2*

*Mot* **=** "Bonjour"

Ce contenu peut également être affiché via l'opération d'écriture.

*Exemple :* **Sortir** *Res*

### Structures de contrôle

En général, les opérations sont accomplies une seule fois séquentiellement. Cependant, les conditions de déroulement d’une action peuvent varier, ce qui induit deux opérations plus particulières : les opérations conditionnelles et les opérations répétées.

Les structures de contrôles représentent ces différentes manières de gérer une suite d'instructions.

#### Structure séquentielle

Les actions sont toujours effectuées.

*Exemple :* on fait ceci, ensuite on fait cela, puis …

⇨ ACTIONS INCONDITIONNELLES

#### Structure alternative (sélective)

Les actions sont effectuées en fonction d’une condition.

*Exemple :* - si … alors…sinon

- cas 1…, cas 2…, cas 3…

⇨ ACTIONS CONDITIONNELLES

#### Structure répétitive (itérative)

Les actions sont effectuées plusieurs fois.

*Exemple :* - tant que …

- jusqu’à ce que …

- faire x fois

⇨ ACTIONS REPETITIVES

### Plus de convivialité …

Pour plus de convivialité, nous vous conseillons vivement d'ajouter les dialogues qui permettront de mieux guider l'utilisateur du programme.

*Exemple :*

Somme2Nbr

**Sortir** " Veuillez introduire deux nombres : "

**Obtenir** *Nbr1*, *Nbr2*

*Res* **=** *Nbr1* **+** *Nbr2*

**Sortir** " La somme de ", *Nbr1*, " et de ", *Nbr2*, " est de ", *Res*

# Les Alternatives

## Qu'est-ce qu'une alternative ?

Une **alternative**, est une instruction qui prévoit, en fonction de deux situations différentes, deux façons d’agir. Cela suppose que l’exécutant sache analyser la condition fixée à son comportement pour effectuer la séquence d'instructions correspondante.

Ce type d'instruction permet ainsi de demander à l'exécutant d'effectuer des séquences d'instructions selon qu'une situation se présente d’une manière ou d’une autre. Cette structure logique porte le nom de test, nous parlerons de **structure alternative**.

En général, il y a deux formes de structures alternatives : l'alternative simple et l'alternative double. Dans le cadre du cours de Principes de programmation, une version un peu particulière d'alternative s'ajoute à ces deux formes de base : la structure "case".

Avant d'aborder ces diverses formes d'alternatives, nous allons définir ce qu'est une **condition**.

## Qu'est-ce qu'une condition ?

Pour parler de **condition**, nous devons aborder la notion de **booléen**.

### Booléen

Un **booléen** est une expression dont la valeur est VRAI ou FAUX. Un booléen peut s'exprimer de deux manières différentes :

* Une **variable** de type booléen (sa valeur est VRAI ou FAUX)

*Exemple :* Soit *Trouvé*, une variable booléenne, on peut lui affecter la valeur VRAI, d'où :

*Trouvé* = VRAI

* Une condition.

### Condition simple

Une condition simple est une **comparaison** entre deux valeurs.

Cela signifie qu’une condition est composée de trois éléments :

* Une valeur ;
* Un opérateur de comparaison ;
* Une autre valeur.

Les valeurs peuvent être a priori de n’importe quel type (numérique ou alphanumérique). Mais si l’on veut que la comparaison ait un sens, il faut que les deux valeurs de la comparaison soient du même type ! On peut bien entendu comparer la valeur d'une variable à une constante ou les valeurs de deux variables.

*Exemple :* Est-ce que la valeur de *Age* est plus grande que 18 ?

Est-ce que la valeur de *Trouvé* est égale à VRAI ?

Est-ce que la valeur de *Nbr1* est plus petite ou égale à que celle de *Nbr2* ?

Pour comparer ces valeurs, on utilise des **opérateurs de comparaison** :

* = égal à …
* ≠ différent de …
* < strictement plus petit que …
* > strictement plus grand que …
* ≤ plus petit ou égal à …
* ≥ plus grand ou égal à …

*Exemples :* (Age > 18)

(Trouvé = VRAI)

(Nbr1 ≤ Nbr2)

L’ensemble des trois éléments composant la condition constitue donc, si l’on veut, une affirmation, qui, à un moment donné, est VRAI ou FAUX. Dans le premier exemple, si la valeur de la variable Age est bien plus grande que 18, la condition est VRAI, sinon, la condition est FAUX.

Il est à noter que ces opérateurs de comparaison peuvent tout à fait s’employer avec des caractères. Ceux-ci sont codés par la machine dans l’ordre alphabétique (voir le code ASCII), les majuscules étant systématiquement placées avant les minuscules.

*Exemple :* Est-ce que la valeur de *Nom* est plus petite que " Daudet " ?

(*Nom* < " Daudet ")

### Condition composée

Certains problèmes exigent parfois de formuler des conditions qui ne peuvent pas être exprimées sous la forme simple exposée ci-dessus. En effet, dans certaines conditions, se cachent non une mais plusieurs conditions.

*Exemple :* Est-ce que la valeur de *Age* est plus grande que 18 ET est-ce que la valeur de *Nom* est plus petite que " Daudet " ?

Est-ce que la valeur de *Age* est plus grande que 18 OU est-ce que la valeur de *Trouvé* est égale à VRAI ?

Pour relier ces conditions, on utilise les **opérateurs logiques**. Les opérateurs logiques utilisés en Principes de programmation sont :

* L'opérateur **AND** ;
* L'opérateur **OR** ;
* L'opérateur **NOT**.

Le **AND** représente l'opérateur logique **ET** qui signifie que **les deux** conditions doivent être remplies. En effet, soit les conditions simples COND1 et COND2, la condition composée (COND1 AND COND2) est VRAI si la condition COND1 est VRAI et la condition COND2 est VRAI.

*Exemple :* ((*Age* > 18) AND (*Nom* < "Pirotte"))

Le **OR** représente l'opérateur logique **OU inclusif** qui signifie que **l'une des deux** conditions doit être remplie **au moins**. En effet, soit les conditions simples COND1 et COND2, la condition composée (COND1 OR COND2) est VRAI si au moins une des deux conditions est VRAI. D'où, soit la condition COND1 est VRAI, soit la condition COND2 est VRAI, soit les deux conditions sont VRAI.

*Exemple :* ((*Age* > 18) OR (*Trouvé* = VRAI))

Le **NOT** représente la négation d'une expression.

## Les structures alternatives

Voici les trois types de structures alternatives utilisées dans les diagrammes d'actions.

* Qu'il s'agisse d'une variable de type booléen ou d'une condition (simple ou composée), nous utiliserons le terme condition.

### Alternative simple

La forme la plus simple d'alternative est :

**SI** condition **ALORS** séquence d'instructions **FINSI**

Cela signifie que, suite à l'examen de la condition, si la valeur est VRAI, on exécute la séquence d’instructions qui suit alors. En revanche, dans le cas où cette valeur est FAUX, l'exécutant saute directement aux instructions situées après FINSI.

Dans les deux cas, les instructions situées juste après FINSI seront exécutées normalement.

Le diagramme d’actions correspondant est le suivant :

**if (** Condition **)**

Séquence d'instructions

ALORS *séquence d'instructions*

SI *condition*

FINSI

### Alternative double

Dans le cas de la structure complète, la forme de l'alternative devient :

**SI** condition **ALORS** séquence d'instructions 1

**SINON** séquence d'instructions 2 **FINSI**

Suite à l'examen de la condition, si la valeur est VRAI, on exécute la séquence d’instructions 1 qui suit alors. Au moment où l'exécutant arrive au SINON, il saute directement à la première instruction située après FINSI. Dans le cas où la valeur de la condition est FAUX, il traite la séquence d'instructions 2 qui suit le SINON.

Dans tous les cas, les instructions situées juste après FINSI seront exécutées normalement.

Le diagramme d’actions correspondant est le suivant :

ALORS *séquence d'instructions* ***1***

SI *condition*

SINON

*séquence d'instructions* ***2***

FINSI

Séquence d'instructions **1**

**if (** Condition **)**

Séquence d'instructions **2**

else

Une alternative étant une instruction, elle peut faire partie d'une séquence d'instructions qui fait elle-même partie d'une alternative. Les alternatives peuvent donc s'imbriquer.

*Exemple :*

**if (** Condition 1 **)**

else

Séquence d'instructions **1**

**if (** Condition 2 **)**

Séquence d'instructions **2**

else

**if (** Condition 3 **)**

Séquence d'instructions **3**

### Case

Cette structure est appelée **case** car il s'agit de tester des cas – case en anglais. Dans certains cas, pour une meilleure lisibilité, cette structure est préférée à une imbrication d'alternatives doubles qui rendrait le diagramme incompréhensible.

En effet, on utilise cette notation dans le cas où le test porte sur une seule variable, un **entier** ou un **caractère**, celle-ci pouvant avoir diverses valeurs, et où la séquence d'instructions varie en fonction de ces valeurs.

*Exemple :* En fonction de la valeur d'une variable entière *Couleur*, afficher la couleur qui correspond à l'entier en fonction d'un code de couleur prédéfini : 1 = rouge, 2 = orange, 3 = vert.

**if (***Couleur* = 1**)**

**if (***Couleur* = 2**)**

**if (***Couleur* = 3**)**

else

**Sortir** " Rouge "

**Sortir** " Orange "

**Sortir** " Vert "

**Sortir** " Autre "

## Exemples

1. Écrire le DA qui sur base d’une valeur binaire *ValBin* obtenue de l’utilisateur au clavier, permet de vérifier si le courant passe (1) ou pas (0). Si le courant passe, afficher le message "Ordinateur sous tension !", sinon rien.

* **Observation** 
  + Donnée : rien ;
  + Entrée : *ValBin* ;
  + Sortie : un message.
* **Diagramme** 
  + Opérations : obtention, alternative, sortie ;
  + Variable : *ValBin* ;
  + Initialisation : aucune ;
  + Diagramme d'actions : TestValeurBinaire.

┌─\* TestValeurBinaire

│

│ **Sortir** "Entrer une valeur binaire (0 ou 1) : "

│ **Obtenir** ValBin

│

│ ┌─ **if** **(***ValBin* **=** 1**)**

│ │

│ │ **Sortir** "Ordinateur sous tension !"

│ └──

└──

* **Évaluation – Résultats**
  + Cas du 0
    - Obtention : 0 ( = *ValBin*)
    - Tableau des variables :

|  |  |  |
| --- | --- | --- |
| **Instruction** | **Valeur des conditions** | **ValBin** |
| Obtention | ? | 0 |
| (*ValBin* = 1) | FAUX | 0 |

* + - Sortie : Aucune.
  + Cas du 1
    - Obtention : 1 ( = *ValBin*)
    - Tableau des variables :

|  |  |  |
| --- | --- | --- |
| **Instruction** | **Valeur des conditions** | **ValBin** |
| Obtention | ? | 1 |
| (*ValBin* = 1) | VRAI | 1 |

* + - Sortie : Ordinateur sous tension.
* **Interprétation**

Le programme demande un binaire à l'utilisateur, et fait le test. Si le binaire obtenu est 0, on n’affiche rien ; si on obtient 1, on affiche "Ordinateur sous tension !".

* **Conclusion**

Le programme fait bien ce qui était demandé dans l'énoncé.

1. Écrire le DA qui étant donné un nombre entier introduit par l’utilisateur au clavier, permet de déterminer si ce nombre est pair ou impair.

* **Observation** 
  + Donnée : rien ;
  + Entrée : un nombre entier ;
  + Sortie : un message.
* **Diagramme** 
  + Opérations : obtention, alternative, sortie ;
  + Variable : *NbrEnt* ;
  + Initialisation : aucune ;
  + Diagramme d'actions : TestPair.

┌─\* TestPair

│

│ **Sortir** "Entrer un nombre entier : "

│ **Obtenir** *NbrEnt*

│

│ ┌─ **if (***NbrEnt* **%** 2 **≠** 0 **)**

│ │

│ │ **Sortir** "Nombre impair !"

│ │

│ ├─ **else**

│ │

│ │ **Sortir** "Nombre pair !"

│ └──

└──

* **Évaluation – Résultats**
  + Cas d'un nombre impair
    - Obtention : 5 ( = *NbrEnt*)
    - Tableau des variables :

|  |  |  |
| --- | --- | --- |
| **Instruction** | **Valeur des conditions** | **NbrEnt** |
| Obtention | ? | 5 |
| (*NbrEnt* % 2 ≠ 0) | VRAI | 5 |

* + - Sortie : Nombre impair !
  + Cas d'un nombre pair
    - Obtention : 8 ( = *NbrEnt*)
    - Tableau des variables :

|  |  |  |
| --- | --- | --- |
| **Instruction** | **Valeur des conditions** | **NbrEnt** |
| Obtention | ? | 8 |
| (*NbrEnt* % 2 ≠ 0) | FAUX | 8 |

* + - Sortie : Nombre pair !
* **Interprétation**

Le programme demande un nombre entier à l'utilisateur, et fait le test. Si le nombre est impair, on affiche "Nombre impair !" ; si le nombre obtenu est pair, on affiche "Nombre pair!".

* **Conclusion**

Le programme fait bien ce qui était demandé dans l'énoncé.

* Pour un énoncé, plusieurs résolutions sont possibles !

*Exemple :*

┌─\* TestPair

│

│ **Sortir** "Entrer un nombre entier : "

│ **Obtenir** *NbrEnt*

│

│ ┌─ **if (***NbrEnt* **%** 2 **=** 0 **)**

│ │

│ │ **Sortir** "Nombre pair !"

│ │

│ ├─ **else**

│ │

│ │ **Sortir** "Nombre impair !"

│ └──

└──

1. Étant donné un utilisateur ayant choisi de quitter le programme Borland C, écrire le DA qui permet de sortir un message lui proposant la sauvegarde des dernières modifications effectuées et en fonction de sa réponse, d’afficher un message à l’écran.

- S'il répond "OUI" - sortir "Sauvegarde en cours d’exécution!" ;

- S'il répond "NON" - sortir "Attention ! Perte potentielle d’informations." ;

- S'il répond "ANNULER" - sortir "Retour à Borland C.".

###### Première version

* **Observation**
  + Donnée : rien ;
  + Entrée : le choix de l'utilisateur ;
  + Sortie : un message.
* **Diagramme**
  + Opérations : obtention, alternatives, sortie ;
  + Variable : *Choix* ;
  + Initialisation : aucune ;
  + Diagramme d'actions : QuitterBorlandC.

┌─\* QuitterBorlandC

│

│ **Sortir** "Voulez-vous sauvegarder ?"

│ **Obtenir** *Choix*

│

│ ┌─ **if (***Choix*= "OUI"**)**

│ │

│ │ **Sortir** "Sauvegarde en cours d’exécution!"

│ │

│ ├─ **else**

│ │

│ │ ┌─ **if** **(***Choix*= "NON"**)**

│ │ │

│ │ │ **Sortir** "Attention ! Perte potentielle d’informations."

│ │ │

│ │ ├─ **else**

│ │ │

│ │ │ **Sortir** "Retour à Borland C."

│ │ └──

│ └──

└──

* **Évaluation – Résultats**
  + Cas du OUI
    - Obtention : "OUI" ( = *Choix*)
    - Tableau des variables :

|  |  |  |
| --- | --- | --- |
| **Instruction** | **Valeur des conditions** | **Choix** |
| Obtention | ? | OUI |
| (*Choix* = "OUI") | VRAI | OUI |

* + - Sortie : Sauvegarde en cours d’exécution!
  + Cas du NON
    - Obtention : "NON" ( = *Choix*)
    - Tableau des variables :

|  |  |  |
| --- | --- | --- |
| **Instruction** | **Valeur des conditions** | **Choix** |
| Obtention | ? | NON |
| (*Choix* = "OUI") | FAUX | NON |
| (*Choix* = "NON") | VRAI | NON |

* + - Sortie : Attention ! Perte potentielle d’informations.
  + Cas du ANNULER
    - Obtention : "ANNULER" ( = *Choix*)
    - Tableau des variables :

|  |  |  |
| --- | --- | --- |
| **Instruction** | **Valeur des conditions** | **Choix** |
| Obtention | ? | ANNULER |
| (*Choix* = "OUI") | FAUX | ANNULER |
| (*Choix* = "NON") | FAUX | ANNULER |

* + - Sortie : Retour à Borland C.
* **Interprétation**

Le programme demande à l'utilisateur de choisir s'il veut sauver ou pas. On teste d'abord si le choix est "OUI" ou pas. Si c'est le cas, on affiche "Sauvegarde en cours d’exécution!" ; sinon, on teste si le choix est "NON". Si l'utilisateur a choisi "NON", on affiche "Attention ! Perte potentielle d’informations." ; sinon, on affiche "Retour à Borland C.".

* **Conclusion**

Le programme fait bien ce qui était demandé dans l'énoncé.

* ***Pour les résolutions suivantes, par souci de concision, la démarche ODERIC ne sera détaillée que pour le premier exercice du chapitre. Pour les autres, seules les deux premières étapes de cette méthode (Observation et Diagramme) seront présentées. Cela ne remet en aucun cas en cause l'utilité de cette méthode lors de la résolution d'un exercice. En effet, ces étapes peuvent être utilisées lors des exercices individuels de manière à s'assurer de la correction des diagrammes réalisés.***

###### Deuxième version : Utiliser une variable supplémentaire

* **Observation**
  + Donnée : rien ;
  + Entrée : le choix de l'utilisateur ;
  + Sortie : un message.
* **Diagramme**
  + Opérations : obtention, alternatives, sortie ;
  + Variables : *Choix*, *Message* ;
  + Initialisation : aucune ;
  + Diagramme d'actions : QuitterBorlandC.

┌─\* QuitterBorlandC

│

│ **Sortir** "Voulez-vous sauvegarder ?"

│ **Obtenir** *Choix*

│

│ ┌─ **if** **(***Choix*= "OUI"**)**

│ │

│ │ *Message* = "Sauvegarde en cours d’exécution!"

│ │

│ ├─ **else**

│ │

│ │ ┌─ **if (***Choix*= "NON"**)**

│ │ │

│ │ │ *Message* = "Attention ! Perte potentielle d’informations."

│ │ │

│ │ ├─ **else**

│ │ │

│ │ │ *Message* = "Retour à Borland C."

│ │ └──

│ └──

│ **Sortir** *Message*

└──

###### Troisième version : Prévoir une option par défaut

* **Observation**
  + Donnée : rien ;
  + Entrée : le choix de l'utilisateur ;
  + Sortie : un message.
* **Diagramme**
  + Opérations : obtention, alternatives, sortie ;
  + Variables : *Choix*, *Message* ;
  + Initialisation : *Message* = "Sauvegarde en cours d'exécution!" ;
  + Diagramme d'actions : QuitterBorlandC.

┌─\* QuitterBorlandC

│

│ **Sortir** "Voulez-vous sauvegarder ?"

│ **Obtenir** *Choix*

│

│ *Message* = "Sauvegarde en cours d’exécution!"

│

│ ┌─ **if** **(***Choix*= "NON"**)**

│ │

│ │ *Message* = "Attention ! Perte potentielle d’informations."

│ │

│ ├─ **else**

│ │

│ │ ┌─ **if** **(***Choix*= "ANNULER"**)**

│ │ │

│ │ │ *Message* = "Retour à Borland C."

│ │ └──

│ └──

│ **Sortir** *Message*

└──

###### Quatrième version : Prévoir les cas d’erreur

Il s'agit du même diagramme d’actions que la deuxième version en ajoutant le cas d'erreur. Tous les cas valides sont testés de manière systématique afin de cibler correctement les cas d'erreur. Le diagramme est le suivant :

┌─\* QuitterBorlandC

│

│ **Sortir** "Voulez-vous sauvegarder ?"

│ **Obtenir** *Choix*

│

│ ┌─ **if** **(***Choix*= "OUI"**)**

│ │

│ │ *Message* = "Sauvegarde en cours d’exécution!"

│ │

│ ├─ **else**

│ │

│ │ ┌─ **if** **(***Choix*= "NON"**)**

│ │ │

│ │ │ *Message* = "Attention ! Perte potentielle d’informations."

│ │ │

│ │ ├─ **else**

│ │ │

│ │ │ ┌─ **if** **(***Choix*= "ANNULER"**)**

│ │ │ │

│ │ │ │ *Message* = "Retour à Borland C."

│ │ │ │

│ │ │ ├─ **else**

│ │ │ │

│ │ │ │ *Message* = "Choix incorrect : tapez OUI/NON/ANNULER !"

│ │ │ └──

│ │ └──

│ └──

│ **Sortir** *Message*

└──

1. Écrire le DA qui, étant donné trois nombres (*Nbr1*, *Nbr2* et *Nbr3*) introduits par l’utilisateur au clavier, permet de déterminer et de sortir le plus petit d'entre eux.

┌─\* Minimum3N

│

│ **Sortir** "Entrez trois nombres : "

│ **Obtenir** *Nbr1*, *Nbr2*, *Nbr3*

│

│ *Min* = *Nbr1*

│

│ ┌─ **if** **(***Nbr2* < *Min***)**

│ │

│ │ *Min = Nbr2*

│ │

│ └──

│ ┌─ **if** **(***Nbr3* < *Min***)**

│ │

│ │ *Min = Nbr3*

│ │

│ └──

│ **Sortir** "Le minimum est : ", *Min*

└──

* **Observation**
  + Donnée : rien ;
  + Entrées : les trois nombres ;
  + Sortie : le minimum.
* **Diagramme**
  + Opérations : obtention, alternatives, sortie ;
  + Variables : *Nbr1*, *Nbr2*, *Nbr3*, *Min* ;
  + Initialisation : *Min* = *Nbr1* ;
  + Diagramme d'actions : Minimum3N.

1. Une adresse IP (Internet Protocol) peut être découpée en 4 zones de 8 bits, c'est-à-dire en 4 octets. La valeur maximale de chacun d'entre eux est 255.

La valeur de la première zone détermine la classe (A, B, C, D ou E) et la classe détermine l’envergure du réseau. Voici un tableau reprenant les différentes valeurs de ce premier octet ainsi que la classe qui leur correspond et le type de réseau qui lui est associé :

|  |  |  |
| --- | --- | --- |
| CLASSES | VALEUR DU 1ER OCTET | TYPE DE RÉSEAU |
| A | 1 → 126 | Grande envergure |
| B | 128 → 191 | Moyenne envergure |
| C | 192 → 223 | Petits réseaux locaux |
|  | 0 ou 127 | Tests |

Étant donnée la valeur décimale du 1er octet de l’adresse IP, écrire le DA qui détermine la classe correspondante (sans tenir compte des cas d'erreur).

* **Observation**
  + Donnée : rien ;
  + Entrée : le premier octet ;
  + Sortie : la classe.
* **Diagramme**
  + Opérations : obtention, alternatives, sortie ;
  + Variables : *Octet*, *Classe* ;
  + Initialisation : aucune ;
  + Diagramme d'actions : AdresseIP.

┌─\* AdresseIP

│

│ **Sortir** "Entrez la valeur du premier octet de l'adresse IP : "

│ **Obtenir** *Octet*

│

│ ┌─ **if** **(***Octet* = 0 **OR** *Octet = 127***)**

│ │

│ │ **Sortir** "Test"

│ │

│ ├─ **else**

│ │

│ │ ┌─ **if** **(***Octet* ≥ 1 **AND** *Octet* ≤ *126***)**

│ │ │

│ │ │ *Classe* = "A"

│ │ │

│ │ ├─ **else**

│ │ │

│ │ │ ┌─ **if** **(***Octet* ≥ 128 **AND** *Octet* ≤ *191***)**

│ │ │ │

│ │ │ │ *Classe* = "B"

│ │ │ │

│ │ │ ├─ **else**

│ │ │ │

│ │ │ │ ┌─ **if** **(***Octet* ≥ 192 **AND** *Octet* ≤ *223***)**

│ │ │ │ │

│ │ │ │ │ *Classe* = "C"

│ │ │ │ │

│ │ │ │ └──

│ │ │ └──

│ │ └──

│ │ **Sortir** "La classe correspondant à cette adresse IP est : ", *Classe*

│ └──

└──

# Les Répétitives

## Qu'est-ce qu'une répétitive ?

Également appelée **boucle** ou **structure itérative**, la **structure répétitive** exprime qu’un ensemble d'actions peut se dérouler plusieurs fois tant qu'une condition reste vérifiée.

### A quoi cela sert-il ?

Pour mieux comprendre l'utilité des répétitives, prenons un exemple concret tel que le cas d’une saisie au clavier (une lecture) où, par exemple, le programme pose une question à laquelle l’utilisateur doit répondre par "OUI" ou "NON".

Tôt ou tard, l’utilisateur, facétieux ou maladroit, risque de taper autre chose que la réponse attendue. Dès lors, le programme peut poser problème soit par une erreur d’exécution (parce que le type de réponse ne correspond pas au type de la variable attendu) soit par une erreur fonctionnelle (il se déroule normalement jusqu’au bout, mais en produisant des résultats fantaisistes).

Pour palier ce problème, on met en place ce qu’on appelle un contrôle de saisie, afin de vérifier que les données entrées au clavier correspondent bien à celles attendues par l’algorithme. Lorsqu'on fait ce contrôle à l'aide d'une alternative, le diagramme d’actions est le suivant :

┌─\* ContrôleDeSaisie

│

│ **Sortir** "Voulez-vous sauvegarder ?"

│ **Obtenir** *Choix*

│

│ ┌─ **if** **(***Choix*≠ "OUI" **AND** *Choix* ≠ " NON"**)**

│ │

│ │ **Sortir**"Réponse erronée. Recommencez"

│ │ **Obtenir** *Choix*

│ └──

└──

Cela semble impeccable, du moins tant que l’utilisateur a le bon goût de ne se tromper qu’une seule fois, et d’entrer une valeur correcte à la deuxième demande.

Si l’on veut à nouveau vérifier le choix, en cas de deuxième erreur, il faut rajouter une alternative, et ainsi de suite.

La solution consistant à imbriquer des alternatives est donc une impasse. La seule issue est d'utiliser une **structure répétitive**.

La forme d'une répétitive est la suivante :

**TANT QUE** condition **FAIRE** séquence d'instructions **FINTANTQUE**

Cela signifie que lorsque le programme arrive sur la ligne du Tant Que. Il examine la valeur de la condition (qui, pour rappel, peut être une variable booléenne ou, plus fréquemment, une condition). Si cette valeur est VRAI, le programme exécute les instructions qui suivent le FAIRE, jusqu’à ce qu’il rencontre la ligne FinTantQue. Il retourne ensuite sur la ligne du TANTQUE, procède au même examen, et ainsi de suite. Cela ne s’arrête que lorsque la valeur de la condition prend la valeur FAUX.

Le diagramme d’actions correspondant est le suivant :

Faire *séquence d'instructions*

TANT QUE *condition*

FINTANTQUE

╔═ **do while** **(***condition***)**

║

║ *Séquence d'instructions*

║

║

╙─

Si on reprend le problème du contrôle de saisie, une première solution consiste à écrire :

┌─\* ContrôleDeSaisie

│

│ **Sortir** "Voulez-vous sauvegarder ?"

│ **Obtenir** *Choix*

│

│╔═ **do while** **(***Choix* ≠ "OUI" **AND** *Choix* ≠ "NON"**)**

│║

│║ **Sortir** "Voulez-vous sauvegarder ?"

│║ **Obtenir** *Choix*

│╙─

└──

Il est important de faire attention au fait que l'obtention de la valeur de *Choix* doit se faire au moins **une fois** avant de tester la condition. En effet, si ce test se fait sans avoir affecté une valeur à cette variable, le résultat sera aléatoire et risque de poser problème.

Pour éviter de devoir écrire deux fois l'obtention, on peut utiliser une autre structure répétitive :

**FAIRE** séquence d'instructions **TANT QUE** condition **FINTANTQUE**

Cela signifie que lorsque le programme arrive sur la ligne FAIRE, il effectue les instructions. Lorsqu'il atteint la ligne TANT QUE, il examine la valeur de la condition. Si cette valeur est VRAI, il retourne sur la ligne du FAIRE et recommence jusqu'à ce que la condition prenne la valeur FAUX qui le fait passer à la ligne FINTANTQUE.

Le diagramme d’actions correspondant est le suivant :

Faire

*séquence d'instructions*

TANT QUE *condition*

FINTANTQUE

╔═ **do**

║

║ *Séquence d'instructions*

║

║

╙─ **while** **(***condition***)**

Si on reprend le problème du contrôle de saisie, la deuxième solution consiste à écrire :

┌─\* ContrôleDeSaisie

│

│╔═ **do**

│║

│║ **Sortir** "Voulez-vous sauvegarder ?"

│║ **Obtenir** *Choix*

│║

│╙─ **while** **(***Choix* ≠ "OUI" **AND** *Choix* ≠ "NON"**)**

└──

### "Sans commencement" ou "Sans fin"

* Dans le cas d'un "TANT QUE condition FAIRE séquence d'instructions FINTANTQUE", si la valeur de la condition n'est jamais à VRAI, le programme ne rentrera jamais dans la boucle et celle-ci ne servira à rien. Cette boucle n'a donc pas de commencement.
* A l'inverse, une structure répétitive dans laquelle la valeur de la condition ne passe jamais à FAUX est une boucle sans fin ou boucle infinie. Le programme rentre alors sans arrêt dans la boucle et n’en sort plus.

Il est donc important, comme pour les structures alternatives, de bien vérifier les diagrammes d'actions et de s'assurer que les conditions soient correctement écrites.

## Compter en bouclant

Outre le contrôle de saisie, les boucles sont également utilisées pour compter.

*Exemple :* Calculer la somme de **5 entiers** entrés par l'utilisateur au clavier et afficher le résultat.

* **Observation**
  + Donnée : rien ;
  + Entrées : les 5 nombres;
  + Sortie : la somme.
* **Diagramme**
  + Opérations : obtention, somme, sortie ;
  + Variables : *Nbr1*, *Nbr2*, *Nbr3*, *Nbr4*, *Nbr5*, *Som* ;
  + Initialisation : aucune ;
  + Diagramme d'actions : Somme5N.

La solution la plus simple qui aurait pu être envisagée est la suivante :

┌─\* Somme5N

│

│ **Sortir** "Entrez 5 entiers : "

│ **Obtenir** *Nbr1, Nbr2, Nbr3, Nbr4, Nbr5*

│

│ *Som* = *Nbr1* + *Nbr2* + *Nbr3* + *Nbr4* + *Nbr5*

│

│ **Sortir** *Som*

└──

Cependant, si on modifie un tant soit peu l'énoncé et qu'au lieu de demander la somme de 5 entiers, on demande la somme de **100 entiers**, cette solution est-elle toujours la plus simple ?

Il semble évident que dans cette situation, la meilleure solution serait d'utiliser une structure répétitive. En effet, le calcul de la somme se ferait au fur et à mesure de l'obtention des nombres et non en une seule instruction longue et illisible. De plus, cela permettrait non seulement de limiter le nombre de variables utilisées, mais également d'adapter facilement le diagramme en fonction du nombre d'entiers demandés.

*Exemple :* Calculer la somme de 100 entiers entrés par l'utilisateur au clavier et afficher le résultat.

* **Observation**
  + Donnée : rien ;
  + Entrées : les 100 nombres;
  + Sortie : la somme.
* **Diagramme**
  + Opérations : obtention, somme, sortie ;

💣 **compteur**

* + Variables : *Nbr*, *Som*, *Cpt* ;

💣 **initialisations**

* + Initialisations : *Som* = 0, *Cpt* = 1 ;
  + Diagramme d'actions : Somme100N.

**Remarques**

* Quand s'arrête la répétitive?

Dans cet exemple, la boucle est utilisée pour augmenter la valeur d’une variable. Cette utilisation des boucles est fréquente, et dans ce cas, il arrive souvent qu’on ait besoin d’effectuer un nombre **déterminé** de passages ou itérations (séquence d'[instruction](http://www.linux-france.org/prj/jargonf/I/instruction.html)s destinée à être exécutée plusieurs fois).

Pour cet exemple, il y a 100 itérations.

Or, même si on connaît le nombre d'itérations, la structure répétitive ne fait pas automatiquement de comptage de passages dans la boucle. La solution pour s’arrêter doit donc être gérée par le programmeur à l’aide d’un **compteur** qui va permettre de comptabiliser le nombre d’itérations déjà réalisées. Ce compteur doit être **initialisé** avant la boucle et **incrémenté** à chaque passage dans la boucle.

* Comment s'assurer que la somme est correcte ?

Comme la somme est calculée au fur et à mesure de l'obtention des nombres en **additionnant à la somme initiale le nombre lu**, il faut qu'elle soit **initialisée** à 0 (le neutre de l'addition).

* Pour cela, on initialise la variable Som à 0 avant la boucle.

┌─\* Somme100N

│

│ *Som* = 0

│ *Cpt* = 1

│

│╔═ **do while** **(***Cpt* ≤ 100**)**

│║

│║ **Sortir** "Entrez un nombre : "

│║ **Obtenir** *Nbr*

│║

│║ *Som* = *Som* + *Nbr*

│║

│║ *Cpt* ++

│╙─

│

│ **Sortir** *Som*

└──

Raccourci de

*Cpt* = *Cpt* + 1

* **Évaluation – Résultats**
  + Itération 1

**Initialisations**

* + - Obtention : 5 (= *Nbr*)
    - Tableau des variables :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Instruction** | **Valeur des conditions** | **Nbr** | **Cpt** | **Som** |
| (*Cpt* ≤ 100) | VRAI | ? | 1 | 0 |
| Obtention | VRAI | 5 | 1 | 0 |
| *Som* = *Som* + *Nbr* | VRAI | 5 | 1 | 5 |
| *Cpt* ++ | VRAI | 5 | 2 | 5 |

* + - Sortie : aucune.
  + Itération 2
    - Obtention : 8 (= *Nbr*)
    - Tableau des variables :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Instruction** | **Valeur des conditions** | **Nbr** | **Cpt** | **Som** |
| (*Cpt* ≤ 100) | VRAI | 5 | 2 | 5 |
| Obtention | VRAI | 8 | 2 | 5 |
| *Som* = *Som* + *Nbr* | VRAI | 8 | 2 | 13 |
| *Cpt* ++ | VRAI | 8 | 3 | 13 |

* + - Sortie : aucune.
  + Itérations 3 à 99 …

Supposons qu'après l'itération 99 (quand *Cpt* = 100) :

* + - *Som* = 157 ;
    - *Nbr* = 9.
  + Itération 100
    - Obtention : 10 (= *Nbr*)
    - Tableau des variables :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Instruction** | **Valeur des conditions** | **Nbr** | **Cpt** | **Som** |
| (*Cpt* ≤ 100) | VRAI | 9 | 100 | 157 |
| Obtention | VRAI | 10 | 100 | 157 |
| *Som* = *Som* + *Nbr* | VRAI | 10 | 100 | 167 |
| *Cpt* ++ | VRAI | 10 | 101 | 167 |

* + - Sortie : aucune.
  + Sortie de la boucle
    - Obtention : aucune, on sort de la boucle.
    - Tableau des variables :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Instruction** | **Valeur des conditions** | **Nbr** | **Cpt** | **Som** |
| (*Cpt* ≤ 100) | FAUX | 10 | 101 | 167 |

* + - Sortie : 167.
* **Interprétation**

Le programme demande en effet à l'utilisateur d'entrer 100 nombres (ni plus, ni moins) les uns à la suite des autres et d'en faire la somme.

* **Conclusion**

Le programme fait bien ce qui était demandé dans l'énoncé.

## Des boucles dans des boucles

Pourquoi imbriquer des boucles ? Pour la même raison qu’on imbrique des alternatives. Une structure alternative permet de tester différents types de situations et d'y apporter la solution adéquate. Comme il a été montré dans les exemples et les exercices du Chapitre 2, une situation peut être décomposée en plusieurs situations, d'où l'imbrication des alternatives.

De même, une boucle est un traitement systématique, un examen d’une série d’éléments un par un (par exemple les classes d'une école). On peut imaginer que pour chaque élément ainsi considéré (pour chaque classe), on doive procéder à un examen systématique d’autre chose (les étudiants d'une classe). Ceci est un exemple typique de boucles imbriquées : on devra programmer une boucle principale (celle qui traite les classes une par une) et à l’intérieur, une boucle secondaire (celle qui prend les étudiants de cette classe un par un).

*Exemple :*

Écrire le DA qui permet de lister pour chaque classe (ou groupe) de première TI, les étudiants qui la composent. Il s'agit donc d'obtenir la lettre correspondante à la classe et pour chacune de ces classes, d'obtenir le nom de chaque étudiant de celle-ci. On demande en même temps de sortir le nom de chaque étudiant précédé de la lettre de sa classe à l'écran et ce, sous le format suivant : G - Martin

N.B. : On sait qu'il y a 6 classes. Pour signifier qu'il n'y a plus d'étudiant, l'utilisateur entrera "ZZZ".

* **Observation**

┌─\* ListingEtudiants1TI

│

│ *CptCla* = 1

│

│╔═ **do while** **(***CptCla* ≤ 6**)**

│║

│║ **Sortir** "Entrez la lettre de la classe : "

│║ **Obtenir** *Classe*

│║

│║ **Sortir** "Entrez le nom du 1er étudiant : "

│║ **Obtenir** *Nom*

│║

│║╔═ **Do while** **(***Nom ≠* "ZZZ"**)**

│║║

│║║ **Sortir** *Classe*, " – ", *Nom*

│║║

│║║ **Sortir** "Entrez le nom du suivant : "

│║║ **Obtenir** *Nom*

│║╙─

│║ *CptCla* ++

│╙─

└──

* + Donnée : rien ;
  + Entrées :

les classes et

les noms d'étudiant ;

* + Sorties :

les classes et

les noms d'étudiant.

* **Diagramme**
  + Opérations :

obtentions, sorties ;

* + Variables : *Classe*,

*Nom*,

*CptCla* ;

* + Initialisation : *CptCla* = 1 ;
  + Diagramme d'actions : ListingEtudiants1TI.

## En règle générale

### Vérifications d'usage

Lorsqu'on utilise une **répétitive**, il est impératif d'effectuer les trois vérifications suivantes en ce qui concerne la variable qui fait partie de la condition, la variable testée :

#### Avant la boucle

La variable testée doit être initialisée ou obtenue.

Il est important de faire attention au fait que l'obtention de la valeur de la variable testée ou à défaut, son initialisation, doit se faire **au moins une fois** avant de tester la condition. En effet, si ce test se fait sans avoir affecté une valeur à cette variable, le résultat sera aléatoire et risque de poser problème.

#### Dans la condition de la boucle

La condition sur la variable test doit être exprimée clairement et écrite correctement.

Dans le cas du "**do while** **(**condition**)**", Si la valeur de la condition n'est jamais à VRAI, le programme ne rentrera jamais dans la boucle et celle-ci ne servira à rien. Cette boucle n'a donc pas de commencement.

#### Dans la boucle

La variable testée doit être mise à jour.

Une structure répétitive dans laquelle la valeur de la condition ne passe jamais à FAUX est une boucle sans fin ou **boucle infinie**.

### Un méta-diagramme d’actions

De manière générale, lors de l'emploi d'une répétitive, le diagramme d’actions a la structure suivante :

┌─\* DiagrammeAvecRépétitive

│

│ Initialisations de la/des variable(s) test

│ *Ex.* : *Cpt* = 0

│ la/des autres variable(s)

│ *Ex.* : *Som =* 0 (pour + ) et *Prod* = 1 (pour \*)

│

│ [ Sortie(s) à effectuer avant la boucle ]

│

│╔═ **do while** **(***condition***)**

│║

│║ [Calcul(s )à effectuer plusieurs fois]

│║ [Sortie(s) à effectuer au fur et à mesure, dans la boucle]

│║

│║ Mise(s) à jour de la/des variable(s ) test

│║ de la /des autres variable(s)

│╙─

│

│ [Calculs finaux]

│

│ [Sortie(s) à effectuer après la boucle]

└──

## Exemples

1. Écrire le DA qui, étant donnés 100 nombres introduits par l’utilisateur au clavier, détermine et sort la moyenne et le maximum de ces nombres.

* **Observation**
  + Donnée : rien ;
  + Entrées : les 100 nombres;
  + Sorties : le maximum, la moyenne.
* **Diagramme**
  + Opérations : obtention, répétitive, sortie ;

***LV → Low Value*** = −∞

***HV → Hight Value*** = +∞

* + Variables : *Nbr*, *Cpt*, *Max*, *Moy* ;
  + Initialisations : *Cpt* = 1, *Max* = LV, *Moy* = 0 ;
  + Diagramme d'actions : MoyMax100N.

┌─\* MoyMax100N

│

│ *Max* = LV

│ *Cpt* = 1

│ *Moy* = 0

│

│╔═ **Do while** **(***Cpt* ≤ 100**)**

│║

│║ **Sortir** "Entrez un nombre : "

│║ **Obtenir** *Nbr*

│║

│║ *Moy* += *Nbr*

│║

│║

│║┌─ **if** **(***Nbr > Max***)**

│║│

│║│ *Max* = *Nbr*

│║└──

│║ *Cpt* ++

│╙─

│ *Moy* /= 100

│

│ **Sortir** "La moyenne des 100 nombres est ", *Moy*, "et leur

│ maximum est ", *Max*,"."

└──

Raccourci de

*Moy* = *Moy* + *Nbr*

1. Écrire le DA qui, étant donnés x nombres introduits par l’utilisateur au clavier, calcule et sort la somme des nombres positifs (non nuls) et le produit des nombres négatifs. L’introduction des nombres s’arrêtera lorsque l’utilisateur entrera la valeur 0.

* **Observation**

┌─\* SomProdN

│

│ **Sortir** "Entrez un nombre : "

│ **Obtenir** *Nbr*

│

│*Som* = 0

│*Prod* = 1

│

│╔═ **Do while** **(***Nbr* ≠ 0**)**

│║

│║┌─ **if** **(***Nbr* > *0***)**

│║│

│║│ *Som* += *Nbr*

│║│

│║├─ **else**

│║│

│║│ *Prod* \*= *Nbr*

│║└──

│║ **Sortir** "Entrez un nombre (0 pour terminer) : "

│║ **Obtenir** *Nbr*

│╙─

│

│ **Sortir** "La somme des positifs est ", *Som*,". **↵**"

│ "Le produit des négatifs est ", *Prod*,"."

└──

* + Donnée : rien ;
  + Entrées : les nombres (0 pour terminer);
  + Sorties : la somme des positifs (non nuls), le produit des négatifs.
* **Diagramme**
  + Opérations : obtention, répétitive, sortie ;
  + Variables : *Nbr*, *Som*, *Prod* ;
  + Initialisations : *Som* = 0, *Prod* = 1 ;
  + Diagramme d'actions : SomProdN.

1. Écrire le DA qui, étant donné un nombre entier positif non nul entré par l’utilisateur au clavier, transforme ce nombre en binaire.

*Exemple :* 10

5

2

1

0

0

1

0

1

*Reste*

*Quotient*

0

1

2

3

(10)10 = (1010)2

= 1\*103 + 0\*102 + 1\*101 + 0\*100

*Cpt*

10 / 2 = +

5 / 2 = +

2 / 2 = +

1 / 2 = +

* **Observation**
  + Donnée : rien ;
  + Entrée : le nombre entier positif non nul ;
  + Sortie : le nombre en binaire.
* **Diagramme**
  + Opérations : obtention, répétitive, sortie ;
  + Variables : NbrDec, NbrBin, Quotient, Reste, Cpt ;
  + Initialisations : *Quotient* = 0, *Cpt* = 0, *NbrBin* = 0;
  + Diagramme d'actions : ConversionDecBin.

┌─\* ConversionDecBin

│

│ **Sortir** "Entrez un nombre positif non nul : "

│ **Obtenir** *NbrDec*

│

│ *Quotient* = *NbrDec*

│ *NbrBin* = 0

│ *Cpt* = 0

│

│╔═ **Do while** **(***Quotient* ≠ 0**)**

│║

│║ *Quotient* = [Quotient / 2]ENT

│║ *Reste* = *NbrDec* % 2

│║ *NbrBin* += *Reste* \* 10^*Cpt*

│║

│║

│║ *Cpt* ++

│╙─

│

│ **Sortir** *NbrBin*

└──

1. Écrire le DA qui, à partir d’un chiffre compris entre 1 et 9 obtenu de l’utilisateur, permet de sortir une demi-pyramide.

*Exemple :* si le chiffre entré est 4

Ligne

1

2 2

3 3 3

┌─\* Pyramide

│ **Sortir** "Entrez un chiffre entre 1 et 9 : "

│ **Obtenir** *Nbr*

│

│ *CptL* = 1

│

│╔═ **Do while** **(***CptL* ≤ *Nbr***)**

│║

│║ *CptC* = 1

│║

│║╔═ **Do while** **(***CptC* ≤ *CptL***)**

│║║

│║║ **Sortir** *CptL*

│║║

│║║ *CptC* ++

│║╙─

│║ **Sortir** "**↵**"

│║ *CptL* ++

│╙─

└──

4 4 4 4

Colonne

* **Observation**
  + Donnée : rien ;
  + Entrée : le chiffre (∈ [1,9]);
  + Sortie : la demi-pyramide.
* **Diagramme**
  + Opérations : obtention, répétitive, sortie ;
  + Variables : *Nbr*, *CptL* (compteur de lignes), *CptC* (compteur de colonnes);
  + Initialisations : *CptL* = 1, *CptC* = 1 ;
  + Diagramme d'actions : Pyramide.

# Les tableaux à simple indice

## Utilité des tableaux

Pour mieux comprendre l'utilité des tableaux, voici une mise en situation : dans un programme, on pourrait avoir besoin de 10 valeurs à conserver **simultanément** en mémoire centrale de manière à les utiliser ultérieurement.

Par exemple, on demande 10 cotes pour calculer leur moyenne, et, en fonction de cette moyenne, on désire afficher la cote qui s'en rapproche le plus.

Évidemment, la seule solution dont nous disposons à l’heure actuelle consiste à déclarer dix variables (appelées par exemple *N1*, *N2*, *N3*, etc.) et à calculer la moyenne comme suit :

*Moy* = (*N1*+*N2*+*N3*+*N4*+*N5*+*N6*+*N7*+*N8*+*N9*+*N10*)/10

On doit ensuite procéder à la comparaison de chacune des variables avec cette moyenne de manière à trouver celle qui en est la plus proche.

Il est évident que cette façon de travailler est d’autant plus illisible et laborieuse que le nombre de valeurs à traiter est important.

De plus, si l’on ne connaît pas avec exactitude la quantité de valeurs à traiter, le programme devient encore plus complexe à réaliser.

C’est pourquoi la programmation permet de rassembler toutes ces variables en une seule, au sein de laquelle chaque valeur est désignée par un numéro. Il s'agit d'un tableau.

## Définition

Un ensemble de valeurs portant le même nom de variable et repérées par un nombre s’appelle un **tableau** ou encore une **variable indicée**.

Le nombre qui, au sein d’un tableau, sert à repérer chaque valeur s’appelle l’**indice**.

Il est important de noter dès maintenant que, pour plus de facilité lors de la traduction d'un DA en langage C on décide par convention de commencer un tableau à l’indice à 0. Un tableau comprenant **N éléments** verra son **indice** aller de **0 à N-1**.

Chaque fois que l’on doit désigner un élément du tableau, on fait figurer le nom du tableau, suivi de l’indice de l’élément, entre crochets : *NomTab* **[** *indice* **]**.

Un tableau de N éléments peut être représenté graphiquement de la manière suivante :

NomTab

NomTab[0]

NomTab[1]

NomTab[2]

NomTab[N-1]

?

?

?

?

…

## Quelques manipulations simples des tableaux

### Remplir un tableau avec des nombres obtenus au clavier

*Exemple :* On désire demander à l'utilisateur d'introduire 10 nombres au clavier et de remplir un tableau avec ces 10 nombres.

Pour cela, on doit prévoir un tableau appelé, par exemple, *TabCote*, composé de 10 cellules qui ont pour nom *Cell*. Chaque cellule contient une valeur correspondant à une des cotes. La structure du tableau est décrite de la manière suivante :

TabCote

Cell

Cote

10 \*

La représentation graphique du tableau est :

TabCote[0]

?

?

?

?

?

?

?

?

?

?

TabCote

TabCote[1]

TabCote[2]

TabCote[3]

TabCote[5]

TabCote[6]

TabCote[7]

TabCote[8]

TabCote[9]

TabCote[4]

* Le tableau est rempli de points d’interrogation puisqu’on ne sait pas, pour l’instant, ce qui se trouve dans cette partie de mémoire. Il n'a pas été initialisé !
* **Observation**
  + Donnée : rien ;
  + Entrées : les 10 nombres ;
  + Sortie : rien.
* **Diagramme**
  + Opération : garnir un tableau ;
  + Variables : *TabCote*, *Ind* ;
  + Initialisation : *Ind* = 0 ;
  + Diagramme d'actions : GarnirTableau

|  |  |
| --- | --- |
| ┌─\* GarnirTableau  │  │ Ind = 0  │  │╔═ Do while (Ind < 10)  │║  │║ Sortir "Entrez un nombre : "  │║ Obtenir TabCote [ Ind ]  │║  │║ Ind++  │╙─  └── | On choisit une variable Ind qui représente l'indice du tableau afin de pouvoir parcourir toutes les cellules de celui-ci, c'est-à-dire en commençant par la cellule 0 au premier passage dans la boucle et en terminant par la cellule 9 au dernier passage.  L’instruction **Obtenir** *TabCote***[***Ind***]** permet de prendre une valeur introduite par l’utilisateur au clavier et de la placer dans la cellule d'indice *Ind* du tableau. |

* **Évaluation – Résultats**
  + Itération 1
    - Obtention : 5 (= *TabCote*[0], car *Ind* = 0)
    - Représentation graphique :

TabCote[0]

?

5

?

?

?

?

?

?

?

?

TabCote

TabCote[1]

TabCote[2]

TabCote[3]

TabCote[5]

TabCote[6]

TabCote[7]

TabCote[8]

TabCote[9]

TabCote[4]

* + - Sortie : aucune.
  + Itération 2
    - Obtention : 8 (= *TabCote*[1], car *Ind* = 1)
    - Représentation graphique :

TabCote[0]

?

5

8

?

?

?

?

?

?

?

TabCote

TabCote[1]

TabCote[2]

TabCote[3]

TabCote[5]

TabCote[6]

TabCote[7]

TabCote[8]

TabCote[9]

TabCote[4]

* + - Sortie : aucune.
  + Itérations 3 à 9
    - Obtentions : 3 (= *TabCote*[2]), 1 (= *TabCote*[3]), 9 (= *TabCote*[4]),

4 (= *TabCote*[5]), 6 (= *TabCote*[6]), 2 (= *TabCote*[7]),

0 (= *TabCote*[8])

* + - Représentation graphique :

TabCote[0]

1

5

8

3

9

4

6

2

0

?

TabCote

TabCote[1]

TabCote[2]

TabCote[3]

TabCote[5]

TabCote[6]

TabCote[7]

TabCote[8]

TabCote[9]

TabCote[4]

* + - Sortie : aucune.
  + Itération 10
    - Obtention : 7 (= *TabCote*[9])
    - Représentation graphique :

TabCote[0]

1

5

8

3

9

4

6

2

0

7

TabCote

TabCote[1]

TabCote[2]

TabCote[3]

TabCote[5]

TabCote[6]

TabCote[7]

TabCote[8]

TabCote[9]

TabCote[4]

* + - Sortie : aucune.
  + Sortie de la boucle
    - Obtention : aucune, on sort de la boucle.
    - Tableau des variables :

|  |  |  |
| --- | --- | --- |
| **Instruction** | **Valeur des conditions** | **Ind** |
| (*Ind* < 10) | FAUX | 10 |

* + - Sortie : aucune.
* **Interprétation**

Le programme demande en effet à l'utilisateur d'entrer 10 nombres (ni plus, ni moins) les uns à la suite des autres et les stocke dans le tableau.

* **Conclusion**

Le programme fait bien ce qui était demandé dans l'énoncé.

### Initialiser un tableau de nombres

Lorsqu’une cellule est destinée au cumul d’un ensemble de valeurs, elle doit être initialisée à 0.

Dans l’exemple précédent, il n’était pas nécessaire d’initialiser les cellules puisqu'on affectait automatiquement une valeur à chaque cellule. Autrement dit, pour une cellule, la valeur qui était obtenue remplaçait directement le contenu précédent de cette cellule.

*Exemple :* Soit un tableau *TabNbr* de 10 cellules décrit comme le précédent. On demande d’initialiser ce tableau à 0.

TabNbr

Cell

Nbr

10 \*

La structure du tableau s'écrit :

La représentation graphique du tableau est :

TabNbr[0]

?

?

?

?

?

?

?

?

?

?

TabNbr

TabNbr[1]

TabNbr[2]

TabNbr[3]

TabNbr[5]

TabNbr[6]

TabNbr[7]

TabNbr[8]

TabNbr[9]

TabNbr[4]

* **Observation**
  + Donnée : rien ;
  + Entrée : rien ;
  + Sortie : rien.
* **Diagramme**
  + Opération : initialiser un tableau ;
  + Variables : *TabNbr*, *Ind* ;
  + Initialisation : *Ind* = 0 ;
  + Diagramme d'actions : InitialiserTableau.

|  |  |
| --- | --- |
| ┌─\* InitialiserTableau  │  │ *Ind* = 0  │  │╔═ **Do while** (*Ind* < 10)  │║  │║ *TabNbr* [ *Ind* ] = 0  │║  │║ *Ind*++  │╙─  └── | Même principe que l’exemple précédent sauf qu'au lieu de demander une valeur pour chaque cellule, on initialise chacune d'elles à 0. |

La représentation graphique du résultat est :

TabNbr[0]

0

0

0

0

0

0

0

0

0

0

TabNbr

TabNbr[1]

TabNbr[2]

TabNbr[3]

TabNbr[5]

TabNbr[6]

TabNbr[7]

TabNbr[8]

TabNbr[9]

TabNbr[4]

### Afficher à l’écran le contenu d’un tableau

*Exemple :* On demande à l'utilisateur d'afficher les nombres qui sont dans le tableau *TabCote* (présenté au point 4.4.1) déjà stocké en mémoire centrale.

* **Observation**
  + Donnée : le tableau ;
  + Entrée : rien ;
  + Sorties : les 10 nombres.
* **Diagramme**
  + Opération : sortir les éléments d'un tableau ;
  + Variables : *TabCote*, *Ind* ;
  + Initialisation : *Ind* = 0 ;
  + Diagramme d'actions : SortirTableau.

|  |  |
| --- | --- |
| ┌─\* SortirTableau  │  │ *Ind* = 0  │  │╔═ **Do while** (*Ind* < 10)  │║  │║ Sortir *TabCote* [ *Ind* ], " "  │║  │║ *Ind*++  │╙─  └── | On prend un indice Ind (initialisé à 0) qui permet de parcourir toutes les cellules du tableau.  L’instruction Sortir *TabCote*[*Ind*] permet de sortir la valeur de la Indième cellule du tableau. |

* **Évaluation – Résultats**

Si on reprend le tableau du point 4.4.1, après avoir été entièrement garni, il apparaît comme suit :

TabCote[0]

1

5

8

3

9

4

6

2

0

7

TabCote

TabCote[1]

TabCote[2]

TabCote[3]

TabCote[5]

TabCote[6]

TabCote[7]

TabCote[8]

TabCote[9]

TabCote[4]

L'exécution du DA affichera la sortie suivante :

5 8 3 1 9 4 6 2 0 7

### Rechercher un élément dans un tableau

*Exemple :* Étant donné un tableau, en mémoire centrale, contenant un maximum de 100 noms de famille, on désire permettre à l'utilisateur de rechercher parmi ceux-ci un nom encodé au clavier. Si le nom recherché est présent dans le tableau, le message "Ce nom a déjà été enregistré !" sera affiché, sinon, le message affiché sera "Ce nom n'a pas encore été enregistré !".

Pour cela, on doit prévoir un tableau appelé, par exemple, *TabNom*, composé de 100 cellules qui ont pour nom Cell. Chaque cellule contient une valeur correspondant à un nom. La structure du tableau est décrite de la manière suivante :

TabNom

Cell

Nom

100 \*

* **Observation**
  + Donnée : le tableau ;
  + Entrée : un nom ;
  + Sortie : un message.
* **Diagramme**
  + Opérations : obtention, recherche, sortie ;
  + Variables : *TabNom*, *Nom*, *Ind* ;
  + Initialisation : *Ind* = 0 ;
  + Diagramme d'actions : RechercheTableau.

┌─\* RechercheTableau

│

│ *Ind* = 0

│

│ **Sortir** "Nom recherché : "

│ **Obtenir** *Nom*

│

│╔═ **Do while** **(***Ind* < 100 **AND** *Nom*≠ *TabNom* [ *Ind* ]**)**

│║

│║ *Ind*++

│╙─

│┌─ **if (***Ind =*100**)**

││

││ **Sortir** "Ce nom n'a pas encore été enregistré !"

││

│├─ **else**

││

││ **Sortir** "Ce nom a déjà été enregistré !"

│└──

└──

## Un peu d'organisation avec les modules

### Pourquoi utiliser des modules ?

Un programme, surtout s'il est long, doit souvent effectuer des traitements similaires à plusieurs moments de son déroulement. Par exemple, la saisie d’une réponse "oui" ou "non" (et le contrôle qu’elle implique), peuvent être répétés à des moments différents de la même application.

La manière la plus évidente, mais aussi la moins habile, de programmer ce genre de choses, est bien entendu de répéter le code correspondant autant de fois que nécessaire, c'est-à-dire en recopiant presque mot à mot les lignes de codes voulues.

Cependant, en procédant de cette manière, le programmeur prend des risques.

En effet, bien que la structure d'un programme écrit de cette manière puisse paraître simple, elle est en réalité inutilement longue. Elle contient des répétitions, et pour peu que le programme soit conséquent, il peut devenir parfaitement illisible.

Or, le fait d’être lisible et facilement modifiable est un critère essentiel pour l’élaboration d’un programme informatique. Cela devient même une nécessité incontournable lorsque l’on travaille dans le cadre d’une organisation (entreprise ou autre) et que le programme doit être mis à jour par des personnes qui n’en sont pas les concepteurs de base.

Il est clair qu’une structure trop lourde pose des problèmes considérables de maintenance. En effet, toute modification du code impose la recherche des différentes apparitions de ce code et, en cas d’oubli, le déroulement du programme devient incertain.

Il faut donc opter pour une autre stratégie, qui consiste à séparer ce traitement du corps du programme et à appeler ces instructions (qui ne figurent donc plus qu’en un seul exemplaire) à chaque fois qu’on en a besoin. Ainsi, la lisibilité est assurée, le programme devient **modulaire**, et il suffit de faire une seule modification au bon endroit, pour que cette modification prenne effet dans la totalité de l’application.

Il est ainsi intéressant de regrouper un ensemble d'instructions sous un même nom, dans un module :

* Chaque fois qu'une même suite d'instructions doit être répétée plusieurs fois dans un programme ;
* Lorsqu'une suite d'instructions forme une action globale.

Le programme est alors plus clair et les erreurs plus facilement détectables.

Le corps du programme s’appelle alors le **module principal**, et ces groupes d’instructions auxquels on a recours s’appellent des **modules**.

Ces termes sont bien sûr génériques. En effet, en langage C, la fonction "main" est la fonction principale et les autres modules sont appelés fonctions, alors qu'en Pascal, on appelle ces modules des procédures si elles ne renvoient pas de résultat et fonctions dans le cas contraire.

### Qu'est-ce qu'un module ?

Avant d'aborder la description d'un module, il est important de spécifier que le **module principal** est celui qui contient la première instruction du programme.

Ce module principal contient une séquence d'instructions parmi lesquelles peuvent figurer des alternatives, des répétitives, mais également des **appels à un module**.

Il s'agit d'une nouvelle instruction qui, comme les alternatives et les répétitives, a une représentation graphique. En effet, un appel à un module se fait en citant le nom de celui-ci dans un rectangle, comme suit :

NomDuModule

*Exemple :* On désire écrire un diagramme d’actions qui fait appel à un module permettant d'afficher "Bonjour !" à l'écran.

* **Observation**
  + Donnée : rien ;
  + Entrée : rien ;
  + Sortie : un message.
* **Diagramme**
  + Opération : sortie ;
  + Variable : aucune ;
  + Initialisation : aucune;
  + Diagramme d'actions : AppelModule.

┌─\* AppelModule

│

│

│

└──

┌─\* Sortie

│

│ **Sortir** "Bonjour !"

│

└──

Sortie

Module Principal

Module

### Module de sortie et passage d'arguments

En modifiant l'exemple qui précède, l'utilisateur peut introduire le message qu'il désire voir apparaître à l'écran. Le programme doit alors demander à l'utilisateur d'introduire un message, et au lieu d'afficher ce message directement dans le module principal, c'est le **module de sortie** qui va s'en charger.

Cela a deux implications :

* Lorsqu’on appelle le module, il faut lui préciser quel message l'utilisateur a introduit ;
* Le module doit, d'une part, être "prévenu" qu’il recevra un message, et, d'autre part, être capable de le récupérer pour l’afficher.

Ces deux indications sont représentées par la **flèche** positionnée dans le **coin supérieur droit** de l'appel au module de sortie. Le **nom de variable** situé à la droite de cette flèche est un **argument** du module, dans ce cas il s'agit du message.

* **Observation**
  + Donnée : rien ;
  + Entrée : un message ;
  + Sortie : le message.
* **Diagramme**
  + Opérations : obtention, sortie ;
  + Variable : *Message* ;
  + Initialisation : aucune;
  + Diagramme d'actions : AppelModuleSortie.

┌─\* AppelModuleSortie

│

│ **Sortir** "Entrez un message : "

│ **Obtenir** *Message*

│

│ ↓ *Message*

│

└──

┌─\* Sortie

│

│ **Sortir** *Message*

└──

Sortie

Il est bien sûr possible de passer plusieurs arguments à un module. Ceux-ci seront séparés par une virgule. Cependant, il est important de ne passer que les arguments nécessaires au bon déroulement du programme.

*Exemple :* Voici un module de sortie auquel est passé en argument un message et un nom.

Sortie

↓ *Message*, *Nom*

### Module d'obtention et renvoi de résultats

Dans ce cas, comme dans l'exemple précédent, l'utilisateur introduit un message, et le programme affiche celui-ci. Cependant, en plus d'utiliser le module de sortie, ce programme va aussi faire appel à un **module d'obtention**. Ce module a pour objectif de demander le message à l'utilisateur et de le renvoyer au module principal.

Cela a deux implications :

* Lorsqu’on quitte le module, il faut préciser au module principal quel message l'utilisateur a introduit ;
* Le module doit être "prévenu" qu’il doit renvoyer un message et être capable de le renvoyer.

Ces deux indications sont représentées par la **flèche** positionnée dans le **coin inférieur droit** de l'appel au module de sortie. Le **nom de variable** situé à la droite de cette flèche est un **résultat** du module ; dans ce cas, il s'agit du message.

* **Observation**
  + Donnée : rien ;
  + Entrée : un message ;
  + Sortie : le message.
* **Diagramme**
  + Opérations : obtention, sortie ;
  + Variable : *Message* ;
  + Initialisation : aucune;
  + Diagramme d'actions : AppelModuleObtention.

┌─\* AppelModuleObtention

│

│

│

│

│

│

│

└──

┌─\* Obtention

│

│ **Sortir** "Entrez un message : "

│ **Obtenir** *Message*

└──

┌─\* Sortie

│

│ **Sortir** *Message*

└──

Sortie

↓ *Message*

Obtention

↓ *Message*

Un module peut aussi renvoyer plusieurs résultats. Ceux-ci seront également séparés par une virgule. Cependant, il est important de ne renvoyer que les résultats nécessaires au bon déroulement du programme.

*Exemple :* Voici un module d'obtention qui renvoie le nom et l'âge d'une personne.

Obtention

↓ *Nom*, *Age*

### Programme complet avec des modules

On désire demander à l'utilisateur de garnir un tableau de 10 nombres entiers afin de pouvoir en calculer la moyenne. Le programme affichera la liste des nombres introduits par l'utilisateur séparés par des virgules ainsi que leur moyenne.

La sortie aura le format suivant :

Les nombres sont : 5, 8, 15, 12, 23, 24, 14, 78, 20, 17

Leur moyenne est : 21.6

La structure du tableau est la suivante :

TabNbr

CellNbr

Nbr

10 \*

* **Observation** 
  + Donnée : rien ;
  + Entrées : les nombres ;
  + Sorties : les nombres et leur moyenne.
* **Diagramme** 
  + Opérations : module d'obtention, module de traitement, module de sortie ;
  + Variables : *TabNbr*, *Cpt*, *Moy* ;
  + Initialisation : aucune dans le programme principal ;
  + Diagramme d'actions : MoyenneNbr.

┌─\* Obtention

│

│ *Cpt* = 0

│

│ **Sortir** "Entrez 10 entiers : "

│

│╔═ **Do while** **(** *Cpt* < 10 **)**

│║

│║ **Obtenir** *TabNbr* [*Cpt*]

│║ *Cpt* ++

│╙─

└──

┌─\* Moyenne

│

│ Cpt = 0

│

│╔═ Do while ( Cpt < 10 )

│║

│║ Moy += TabNbr [Cpt]

│║ Cpt ++

│╙─

│ Moy /= 10

└──

┌─\* Sortie

│

│ **Sortir** "Les nombres sont : ", *TabNbr* [0]

│

│ *Cpt* = 1

│

│╔═ **Do while** **(** *Cpt* < 10 **)**

│║

│║ **Sortir** ", ", *TabNbr* [*Cpt*]

│║ *Cpt* ++

│╙─

│ **Sortir** "**↵**Leur moyenne est : ", *Moy*

└──

┌─\* MoyenneNbr

│

│

│

│

│

│

│

│

│

│

│

│

│

└──

Obtention

↓ *TabNbr*

Sortie

↓ *TabNbr*, *Moy*

Moyenne

↓ *Moy*

↓ *TabNbr*

## Les structures

Jusqu'à présent, deux façons de mémoriser les informations en mémoire centrale ont été abordées : les variables et les tableaux.

Une **variable** permet de mémoriser **une information** soit de type numérique, soit de type alphanumérique.

Un **tableau** permet de mémoriser **plusieurs informations** sous un même nom de variable. Cependant, ces informations doivent toutes être du même type. Elles sont repérées grâce à un numéro qui les identifie au sein du tableau.

### Pourquoi utiliser des structures ?

Si le programmeur a besoin de mémoriser plusieurs informations de types différents pour **une même entité** à traiter, il est contraint de prévoir plusieurs variables.

*Exemple :* Un programme nécessite de mémoriser les caractéristiques d'une personne. Celle-ci peut être représentée par son nom, son prénom et sa date de naissance (au format JJMMAAAA).

Dans ce cas, le programmeur devra utiliser une variable pour le nom, une variable pour le prénom et une variable pour la date de naissance :

* *Nom* : alphanumérique ;
* *Prenom* : alphanumérique ;
* *DateNais* : numérique.

Si, en plus de cela, le programmeur doit traiter **plusieurs entités ayant les mêmes caractéristiques**, il doit alors utiliser plusieurs tableaux, chacun consacré à une des caractéristiques.

*Exemple :* Un programme nécessite de mémoriser les caractéristiques d'un ensemble de 10 personnes.

Dans ce cas, le programmeur devra utiliser un tableau de 10 éléments contenant les noms, un tableau de 10 éléments contenant les prénoms et un tableau de 10 éléments contenant les dates de naissance :

* *TabNom* : alphanumérique ;
* *TabPre* : alphanumérique ;
* *TabDate* : numérique.

La représentation graphique des ces trois tableaux serait la suivante :

?

?

?

?

?

?

?

?

?

TabNom

TabNom[0]

TabNom[1]

TabNom[2]

TabNom[3]

TabNom[5]

TabNom[6]

TabNom[7]

TabNom[8]

TabNom[9]

TabNom[4]

?

?

?

?

?

?

?

?

?

?

TabPre

TabPre[0]

TabPre[1]

TabPre[2]

TabPre[3]

TabPre[5]

TabPre[6]

TabPre[7]

TabPre[8]

TabPre[9]

TabPre[4]

?

?

?

?

?

?

?

?

?

?

TabDate

TabDate[0]

TabDate[1]

TabDate[2]

TabDate[3]

TabDate[5]

TabDate[6]

TabDate[7]

TabDate[8]

TabDate[9]

TabDate[4]

Il est cependant nécessaire que ces trois tableaux soient **traités simultanément**, c'est-à-dire que, pour chaque personne traitée, chacune de ses caractéristiques se trouve dans les cellules des tableaux portant le même numéro.

*Exemple :* Si l'utilisateur veut enregistrer les caractéristiques de Denis Ritchie (qui a inventé le C en 1974 avec Brian Kernighan), il faut faire en sorte que celles-ci aient le même indice dans chacun des tableaux. D'où, ind représentant l'indice du tableau à un moment donné, on doit avoir :

*TabNom*[*ind*] = "Ritchie"

*TabPre*[*ind*] = "Denis"

*TabDate*[*ind*] = 09091941

L'utilisation d'une variable ou d'un tableau par caractéristique est assez encombrante et la manipulation simultanée de ces éléments rend le programme complexe et peu lisible. Comme cela a déjà été précisé, la lisibilité d'un algorithme est primordiale.

De ce fait, il a été décidé d'alléger la notation afin de permettre au programmeur de n'utiliser qu'un seul nom de variable pour une même entité.

### Qu'est-ce qu'une structure ?

Les **structures** permettent de remédier aux lacunes des tableaux, en regroupant des variables au sein d'une entité repérée par un seul nom de variable.   
Les éléments contenus dans la structure sont appelés **champs de la structure**.

Une structure, en pseudo langage, se présente de la manière suivante :

NomVar

NomChamp1

NomChamp …

NomChampN

Il s'agit ainsi de regrouper sous un même nom de variable un ensemble de champs représentant chacun une des caractéristiques de l'entité que l'on désire mémoriser. Le nombre de champs est fonction du nombre de caractéristiques.

*Exemple :* Si on reprend l'exemple d'une personne ayant un nom, un prénom et une date de naissance, la **structure** est présentée sous la forme qui suit :

Personne

Nom

Prenom

DateNais

Chaque champ devient une partie de la variable. Si on reprend les valeurs précédentes, la **représentation graphique** de celle-ci devient :

*Personne*

"Ritchie"

"Denis"

09091941

Pour pouvoir différencier les champs de la variable afin de leur affecter une valeur ou de les afficher, la **notation** suivante est utilisée :

*NomVar***.***NomChamp*

*Exemple :* Dans l'exemple précédent, les trois champs sont repérés comme suit :

*Personne***.***Nom*

*Personne***.***Prenom*

*Personne***.***DateNais*

### Tableau de structures

Dans le cas où le programmeur a besoin de plusieurs variables représentant le même type d'entité, il est conseillé d'utiliser un **tableau de structures** au lieu de plusieurs tableaux reprénant chacun une caractéristique.

Un **tableau de structure** est représenté de la manière suivante :

Cell

N\*

NomChamp1

NomChamp …

NomChampN

NomTab

La **représentation graphique** est la suivante :

*NomTab*

?

…

?

?

…

?

?

…

?

…

…

…

?

…

?

?

…

?

NomTab[0]

NomTab[1]

…

NomTab[N-2]

NomTab[N-1]

NomTab[2]

Pour pouvoir différencier les champs de chaque élément du tableau, afin de leur affecter une valeur ou de les afficher, la **notation** suivante est utilisée :

*NomTab*[*ind*]**.***NomChamp*

### Exemple

#### Énoncé

Les étudiants d'une Haute École peuvent réserver un ordinateur durant une des 100 périodes mises à disposition. Les périodes sont numérotées de 0 à 99.

Pour effectuer la réservation, les étudiants doivent entrer le numéro de la période (NumLu) souhaitée ainsi que leur identifiant (LoginLu).

* Si la période est libre, l'ordinateur est réservé et l'identifiant de l'étudiant est retenu.
* Sinon, l'étudiant a deux options :
  + Arrêter sa réservation en tapant -1 comme numéro de période ;
  + Donner le numéro d'une autre période.

Pour clôturer les réservations, l'administrateur entrera "ZZZ" comme nom d'utilisateur.

En fin de programme, l'administrateur souhaite connaître :

* La liste des périodes réservées avec le nom de l'étudiant correspondant ;
* Le(les) numéro(s) de période la(les) plus demandée(s) (ex æquo à envisager).

#### Solution

Pour résoudre cet exercice, on a besoin de deux tableaux : un pour les réservations (TabReserv) et un pour les périodes les plus demandées (TabPeriode)

Les structures de ces tableaux sont :

Cell

N\*

IdentEtud

NbrDem

TabReserv

Cell

N\*

NumPer

TabPeriode

* **Observation** 
  + Donnée : rien ;
  + Entrées : les logins et les numéros de période ;
  + Sorties : la liste des périodes réservées et la(les) période(s) la(les) plus demandée(s).
* **Diagramme** 
  + Opérations : module d'initialisation, module d'obtention, module de sortie des périodes réservées, module de recherche du (des) maximum(s), module de sortie du (des) maximum(s) ;
  + Variables : NumLu, LoginLu, TabReserv, CptRes, Max, TabPeriode, CptPer ;
  + Initialisation : le module d'initialisation de *Tabeserv* ;
  + Diagramme d'actions : GestionReservationOrdi.

┌─\* Initialisation

│

│ *CptRes* = 0

│

│╔═ **do while** **(** *CptRes ≤* 99**)**

│║

│║ *TabReserv*[*CptRes*]***.****IdentEtud* = " "

│║ *TabReserv*[*CptRes*]***.****NbrDem* = 0

│║

│║ *CptRes* ++

│╙─

│

│ **Sortir** "Entrez un login : "

│ **Obtenir** *LoginLu*

│

│╔═ **do while** **(** *LoginLu* ≠ "ZZZ" **)**

│║

│║ **Sortir** "Entrez le numéro de période demandé : "

│║ **Obtenir** *NumLu*

│║

│║ ╔═ **do while** **(** *NumLu*≠-1 **AND** *TabReserv*[*NumLu*]**.***IdentEtud* ≠ " " **)**

│║ ║

│║ ║ *TabReserv*[*NumLu*]**.***NbrDem ++*

│║ ║

│║ ║ **Obtenir** *NumLu*

│║ ╙─

│║┌─ **if** **(***NumLu*≠ -1**)**

│║│

│║│ *TabReserv*[*NumLu*]**.***NbrDem ++*

│║│ *TabReserv*[*NumLu*]**.***IdentEtud = LoginLu*

│║│

│║├─ **else**

│║│

│║│ **Sortir** "Recherche annulée."

│║└──

│║ **Sortir** "Entrez un login : "

│║ **Obtenir** *LoginLu*

│╙─

│ **Sortir** "Période**¬**Nom "

│

│ *CptRes* = 0

│

│╔═ **do while** **(** *CptRes* < 100 **)**

│║

│║┌─ **if** **(***TabReserv*[*CptRes*]**.***NbrDem* ≠ 0**)**

│║│

│║│ **Sortir** "**↵**", *CptRes*

│║│ **Sortir** "**¬**", *TabReserv*[*CptRes*]**.***IdentEtud*

│║└──

│║ *CptRes* ++

│╙─

.

.

.

.

.

.

│

│ *CptRes* = 0

│ *Max* = -1

│

│╔═ **do while** **(***CptRes* <100**)**

│║

│║┌─ **if** **(***TabReserv*[*CptRes*]**.***NbrDem >* *Max***)**

│║│

│║│ *Max = TabReserv*[*CptRes*]**.***NbrDem*

│║│ *CptPer* = 0

│║│ *TabPeriode*[*CptPer*] *= CptRes*

│║│

│║├─ **else**

│║│

│║│┌─ **if** **(***TabReserv*[*CptRes*]**.***NbrDem =* *Max***)**

│║││

│║││ *CptPer ++*

│║││ *TabPeriode*[*CptPer*] *= CptRes*

│║│└──

│║└──

│║ *CptRes* ++

│╙─

│

│┌─ **if** **(***CptPer >* 0**)**

││

││ **Sortir** "Les périodes les plus demandées sont : "

││

││ *CptRes* = 0

││

││╔═ **do while** **(** *CptRes* < *CptPer***)**

││║

││║ **Sortir** *TabPeriode*[*CptRes*]," "

││║ *CptRes* ++

││╙─

│├─ **else**

││

││ **Sortir** "La période la plus demandée est : ", *TabPeriode*[0]

│└──

└──

## Les tableaux triés

Avant de voir comment trier un tableau, technique indispensable à tout programmeur, il est d'abord question de savoir manipuler un tableau dont les éléments sont déjà triés.

### Recherche simple

*Exemple :* On désire écrire un diagramme d’actions qui permet de retrouver, dans un tableau de noms présent en mémoire centrale, un nom entré au clavier par l'utilisateur. Si le nom est présent dans le tableau, le message affiché est "Ce nom est répertorié en position : " suivi de la position. Sinon, le message est " Ce nom n'est pas répertorié !".

Considérant un tableau de noms, en mémoire centrale, dont la structure est la suivante :

TabNom

Cell

Nom

7 \*

Celui-ci ayant déjà été garni, la représentation graphique de ce tableau est :

TabNom[0]

Zoé

*TabNom*

TabNom[1]

TabNom[2]

TabNom[3]

TabNom[5]

TabNom[6]

TabNom[4]

Aline

Wim

Tom

Marc

Denis

Cédric

* **Observation**
  + Données : le tableau de noms, la taille du tableau ;
  + Entrées : le nom à rechercher ;
  + Sortie : un message.
* **Diagramme**
  + Opération : rechercher le nom ;
  + Variables : *TabNom*, *Cpt* ;
  + Initialisation : *Cpt* = 0 ;
  + Diagramme d'actions : RechercheTableauTrie.

┌─\* RechercheTableauTrie

│

│ *Cpt* = 0

│ **Sortir** "Nom recherché : "

│ **Obtenir** *Nom*

│

│╔═ **do while** **(***Cpt* < 7 **AND** *Nom*> *TabNom* [*Cpt*]**)**

│║ *Cpt* ++

│╙─

│┌─ **if (***Cpt* < 7 **AND** *Nom*= *TabNom* [*Cpt*]**)**

││

││ **Sortir** "Ce nom est répertorié en position : ", *Cpt+1*

││

│├─ **else**

││

││ **Sortir** "Ce nom n'est pas répertorié !"

│└──

└──

### Recherche dichotomique

Voici une technique célèbre de recherche qui révèle son utilité lorsque le nombre d'éléments est très élevé.

En effet, si au lieu d'avoir 7 éléments dans le tableau, il y en avait 40 000. Une première manière de vérifier si un nom se trouve dans ce tableau consiste à examiner successivement tous les noms, du premier au dernier, et à les comparer avec le nom recherché. Cette technique fonctionne, mais cela risque d'être long : si le nom ne se trouve pas dans le tableau, l'algorithme ne saura le signifier qu'après 40 000 tours de boucle ! Et même si le nom figure dans le tableau, la réponse exigera tout de même en moyenne 20 000 tours de boucle. C'est beaucoup, même pour un ordinateur.

Or, étant donné que les éléments du tableau sont triés (dans notre cas les noms sont triés par ordre alphabétique) il existe une manière beaucoup plus efficace d'effectuer la recherche.

Pour mieux comprendre cette technique, une mise en situation intéressante est celle de la recherche d'un mot dans le dictionnaire.

La manière la plus rationnelle de chercher un mot dans un dictionnaire est de comparer le mot à vérifier avec le mot qui se trouve au milieu du dictionnaire. Si le mot à vérifier est antérieur dans l'ordre alphabétique, on sait qu'on devra le chercher dorénavant dans la première moitié du dictionnaire. Si non, on sait qu'on devra le chercher dans la deuxième moitié.

A partir de là, on prend la moitié de dictionnaire qui reste, et on recommence : on compare le mot à chercher avec celui qui se trouve au milieu du morceau de dictionnaire restant. On écarte la mauvaise moitié, et on recommence, et ainsi de suite.

En continuant à couper le dictionnaire en deux, puis encore en deux, etc. on finit par se retrouver avec des morceaux qui ne contiennent plus qu'un seul mot. Et si on n'est pas tombé sur le bon mot à un moment ou à un autre, c'est que le mot à vérifier ne fait pas partie du dictionnaire.

Regardons ce que cela donne au niveau du nombre d'opérations à effectuer, en choisissant le pire cas : celui où le mot est absent du dictionnaire.

* Au départ, on cherche le mot parmi 40 000.
* Après le test n°1, on ne le cherche plus que parmi 20 000.
* Après le test n°2, on ne le cherche plus que parmi 10 000.
* Après le test n°3, on ne le cherche plus que parmi 5 000.
* etc.
* Après le test n°15, on ne le cherche plus que parmi 2.
* Après le test n°16, on ne le cherche plus que parmi 1.

Et là, on sait que le mot n'existe pas. D'où, la réponse est obtenue en 16 opérations contre 40 000 précédemment ! Cette différence de performance entre la technique de recherche simple et la technique de recherche par élimination montre bien l'utilité de cette dernière.

Si chaque mot du dictionnaire était placé dans une cellule d'un tableau, la recherche pourrait se faire en suivant le même principe. D'où, cette nouvelle technique de recherche dans un tableau trié est appelée **recherche dichotomiqu**e (du grec "couper en deux").

*Exemple :* Étant donné un tableau TabNbr de T cellules en mémoire centrale, on désire rechercher un nombre, entré par l'utilisateur au clavier. Si ce nombre est trouvé dans le tableau, le message affiché est "Le nombre est en position : " suivi de la position de celui-ci. Sinon, le message est "Ce nombre n'est pas dans le tableau".

TabNbr

Cell

Nbr

T \*

La structure du tableau s'écrit :

Afin de pouvoir délimiter la partie du tableau dans laquelle la recherche se fait, il est nécessaire d'utiliser deux variables permettant de connaître la limite inférieure et la limite supérieure de cette partie du tableau. De plus, il est nécessaire de connaître le milieu de cet intervalle.

* **Observation** 
  + Données : le tableau de nombres, la taille du tableau (*T*) ;
  + Entrée : le nombre à rechercher ;
  + Sortie : un message.
* **Diagramme** 
  + Opération : rechercher le nombre par dichotomie ;
  + Variables : TabNbr, Nbr, LInf, LSup, Milieu ;
  + Initialisations : *LInf* = 0, *LSup* = *T*-1, *Milieu* = [(*LInf* + *LSup*)/2]*ENT*;
  + Diagramme d'actions : RechercheDichotomique.

┌─\* RechercheDichotomique

│

│ *LInf* = 0

│ *LSup* = T-1

│ *Milieu* = [(*LInf* + *LSup*)/2]ENT

│

│ **Sortir** "Nombre recherché : "

│ **Obtenir** *Nbr*

│

│╔═ **do while** **(***LInf* < *LSup* **AND** *TabNbr* [*Milieu*] ≠ *Nbr***)**

│║

│║┌─ **if (***Nbr < TabNbr* [*Milieu*]**)**

│║│

│║│ *LSup* = *Milieu* - 1

│║├─ **else**

│║│

│║│ *LInf* = *Milieu* + 1

│║└──

│║ *Milieu* = [(*LInf* + *LSup*)/2]ENT

│╙─

│┌─ **if (***LInf* > *LSup***)**

││

││ **Sortir** "Ce nombre n'est pas dans le tableau !"

││

│├─ **else**

││

││ **Sortir** "Ce nombre est en position : ", *Milieu*

│└──

└──

### Décalages

Il est important de noter qu'il est impossible de supprimer ou d'ajouter une cellule à un tableau. En effet, comme cet espace est alloué de manière statique, aucune modification de cet espace n'est possible.

D'où, pour simuler ces manipulations, on déplace les valeurs des cellules déjà garnies, vers la droite ou vers la gauche, selon qu'il s'agit d'une insertion ou d'une suppression.

Lorsqu'on insère un élément dans un tableau, il est nécessaire, avant tout, de déplacer toutes les valeurs qui se trouvent dans les cellules suivant la cellule dans laquelle on va insérer l'élément. Dans le cas contraire, on perdra la valeur qui était dans cette cellule avant l'insertion. Il ne s'agirait donc plus d'une insertion, mais d'une modification.

Si on désire supprimer un élément du tableau, on se limite à effacer la valeur. La cellule devient alors vide à cet endroit. D'où, il est nécessaire dans ce cas de déplacer les valeurs de toutes les cellules qui suivent celle dont on a supprimé la valeur de manière à combler ce vide.

Les algorithmes qui permettent de réaliser ces décalages sont présentés ci-dessous sur base du tableau TabNbr dont la structure est :

TabNbr

Cell

Nbr

12 \*

Ce tableau est en mémoire centrale et a déjà été partiellement rempli. Le nombre d'éléments actuellement présents dans le tableau est *N*.

#### Décalage à droite

On désire créer un module permettant de décaler vers la droite tous les éléments du tableau, dont les N premiers éléments sont garnis, à partir de la position Pos, passée en argument.

* **Observation** 
  + Données : le tableau de nombres, la taille du tableau (12), le nombre d'éléments garnis (*N*) ;
  + Arguments : le tableau, le nombre d'éléments garnis (*N*), la position ;
  + Résultat : le tableau modifié.
* **Diagramme** 
  + Opération : décalage à droite ;
  + Variables : *TabNbr*, *Cpt*, *Pos* ;
  + Initialisation : *Cpt* = *N* ;
  + Diagramme d'actions : DecalageDroite.

┌─\* DecalageDroite

│

│ *Cpt* = *N*

│

│╔═ **do while** **(***Cpt* > *Pos***)**

│║

│║ *TabNbr* [*Cpt*] = *TabNbr* [*Cpt*-1]

│║ *Cpt* --

│╙─

└──

Représentation graphique :

***N* = 10** et ***Pos* = 4**

1

2

3

4

5

6

1

3

4

7

15

21

25

41

46

34

*Pos*

1

3

4

7

*15* 15

15

21

34

46

41

25

*Pos*

#### Décalage à gauche

On désire créer un module permettant de décaler vers la gauche tous les éléments du tableau, dont les *N* premiers éléments sont garnis, jusqu'à la position *Pos*, passée en argument.

* **Observation** 
  + Données : le tableau de nombres, la taille du tableau (12), le nombre d'éléments garnis (*N*) ;
  + Arguments : le tableau, le nombre d'éléments garnis (*N*), la position ;
  + Résultat : le tableau modifié.
* **Diagramme** 
  + Opération : décalage à gauche ;
  + Variables : *TabNbr*, *Cpt*, *Pos*, *N* ;
  + Initialisation : *Cpt* = *Pos* ;
  + Diagramme d'actions : DecalageGauche.

┌─\* DecalageGauche

│

│ *Cpt* = *Pos*

│

│╔═ **do while** **(***Cpt* < *N*-1**)**

│║

│║ *TabNbr* [*Cpt*] = *TabNbr* [*Cpt+*1]

│║ *Cpt* ++

│╙─

└──

Représentation graphique :

1

3

4

7

15

21

25

41

*41*

34

*Pos*

5

4

3

2

1

1

3

4

7

11

15

21

34

41

25

*Pos*

***N* = 10** et ***Pos* = 4**

### Insertion d’un élément

*Exemple :* On désire écrire un diagramme d’actions qui permet l'insertion d'un nouvel élément, introduit par l'utilisateur au clavier, dans un tableau trié de 12 nombres, dont *N* cellules sont déjà garnies. Si l'élément est déjà dans le tableau, le message "Élément déjà présent !" est affiché.

* **Observation** 
  + Données : le tableau de nombres, la taille du tableau (12), le nombre d'éléments garnis (*N*) ;
  + Entrée : le nouveau nombre ;
  + Sortie : Un message.
* **Diagramme** 
  + Opérations : recherche de la position, décalage à droite, insertion ;
  + Variables : *TabNbr*, *Pos*, *Nbr*, *N* ;
  + Initialisation : aucune ;
  + Diagramme d'actions : InsertionTrie.

┌─\* InsertionTrie

│

│┌─ **if (***N**<* 12**)**

││

││ **Sortir** "Nombre à insérer : "

││ **Obtenir** *Nbr*

││

││ *Pos* = 0

││╔═ **do while** **(***Pos* < *N* **AND** *Nbr* > *TabNbr*[*Pos*]**)**

││║

││║ *Pos* ++

││╙─

││┌─ **if (***Pos**<*N **AND** *Nbr* = *TabNbr*[*Pos*]**)**

│││

│││ **Sortir** "Elément déjà présent !"

│││

││├─ **else**

│││

│││ *Cpt* = *N*

│││

│││╔═ **do while** **(***Cpt* > *Pos***)**

│││║

│││║ *TabNbr* [*Cpt*] = *TabNbr* [*Cpt*-1]

│││║ *Cpt* --

│││╙─

│││

│││ *TabNbr*[*Pos*] = *Nbr*

│││ *N*++

││└──

│└──

└──

Si *N* est égal à 12, il n'y a plus de place dans le tableau pour y ajouter un élément.

### Suppression d'un élément

*Exemple :* On désire écrire un diagramme d’actions qui permet la suppression d'un élément, introduit par l'utilisateur au clavier, dans un tableau trié de 12 nombres, dont *N* cellules sont déjà garnies. Si l'élément n'est pas dans le tableau, le message affiché est "Élément absent !".

* **Observation** 
  + Données : le tableau de nombres, la taille du tableau (12), le nombre d'éléments garnis (*N*) ;
  + Entrées : le nombre à supprimer ;
  + Sortie : Un message.
* **Diagramme** 
  + Opérations : recherche de la position, décalage à gauche ;
  + Variables : *TabNbr*, *Pos*, *Nbr*, *N* ;
  + Initialisation : aucune ;
  + Diagramme d'actions : SuppressionTrie.

┌─\* SuppressionTrie

│

│┌─ **if (***N* > 0**)**

││

││ **Sortir** "Nombre à supprimer : "

││ **Obtenir** *Nbr*

││

││ *Pos* = 0

││╔═ **do while** **(***Pos* < *N* **AND** *Nbr* > *TabNbr*[*Pos*]**)**

││║

││║ *Pos* ++

││╙─

││┌─ **if (***Pos**= N***OR** *Nbr* < *TabNbr*[*Pos*]**)**

│││

│││ **Sortir** "Elément absent !"

│││

││├─ **else**

│││

│││*Cpt* = *Pos*

│││

│││╔═ **do while** **(***Cpt* < *N*-1**)**

│││║

│││║ *TabNbr* [*Cpt*] = *TabNbr* [*Cpt+*1]

│││║ *Cpt* ++

│││╙─

│││ *N --*

││└──

│└──

└──

Si *N* est égal à 0, il n'y a aucun élément à supprimer.

# Les Tris

## Pourquoi trier les éléments d'un tableau ?

Lorsqu'un programmeur doit réaliser un algorithme, il est très fréquent qu'il doive traiter des informations suivant un ordre particulier (en ordre croissant ou en ordre décroissant). Pour cela, il doit pouvoir trier ces informations.

Il existe plusieurs stratégies possibles pour trier les éléments d’un tableau : le tri par sélection, le tri par échange ou tri à bulles, le tri par insertion, le Quicksort, le Heapsort, etc. Seuls les deux premiers de cette liste non exhaustive sont présentés ci-après.

Pour plus de facilité, ne seront développés que les algorithmes permettant de trier les tableaux en ordre croissant, c’est-à-dire tel que **tout élément est plus petit que** **celui qui le suit**. Pour que ces algorithmes trient un tableau en ordre décroissant, seule la condition de l'alternative permettant l'échange des éléments est à modifier.

## Comment choisir le bon algorithme ?

La notion de **complexité des algorithmes** est un aspect non négligeable de la programmation. L'exécution d'un programme a toujours un coût.

Il existe deux paramètres essentiels pour mesurer ce coût :

* Le temps d'exécution : **la complexité en temps** ;
* L'espace mémoire requis : **la complexité en espace**.

L'objectif de cette mesure est de proposer des méthodes qui, pour la résolution d'un problème donné, permettent d'estimer le coût d'un algorithme, de **comparer** deux algorithmes différents (sans avoir à les programmer effectivement).

L'analyse de la complexité consiste à déterminer une fonction associant un coût (en unité de temps ou de mémoire) à chaque entrée soumise à l'algorithme.

Pour **la complexité en espace**, on se contente d'associer un coût à un paramètre entier n qui résume la taille de la donnée :

* Pour la recherche d'un élément dans un tableau : n = taille du tableau ;
* Pour le calcul d'un terme d'une suite définie par une relation de récurrence (par exemple la suite de fibonacci) : n = nombre de termes de la suite.

Pour **la complexité en temps**, on calcule, en fonction de n, le nombre d'opérations élémentaires (addition, comparaison, affectation,...) requises par l'exécution de l'algorithme. C'est l'ordre de grandeur du nombre d'opérations, noté O(n), qui est utilisé pour exprimer la complexité.

On dira que l'algorithme de recherche dans un tableau de taille *N* est de complexité O(*N*).

Selon l'algorithme, la complexité peut varier en fonction des données. Il y a trois manières de calculer la complexité :

* La complexité dans le pire des cas :

On calcule le coût dans le pire des cas ;

* La complexité en moyenne :

On calcule le coût pour chaque donnée possible puis on divise la somme de ces coûts par le nombre de données différentes ;

* La complexité dans le meilleur des cas :

On calcule le coût en se plaçant dans le meilleur des cas.

Pour un algorithme avec essentiellement des calculs numériques, on compte les opérations coûteuses : multiplications, exponentielles, logarithmes, racines carrées, etc. Les opérations qui, relativement aux autres, consomment peu de temps, telles que les tests, les affectations et les incrémentations ne sont pas comptabilisées.

Dans les autres cas, on compte les accès à la mémoire ou le nombre d'appels à l'opération la plus fréquente.

La complexité permet donc de **comparer et** ainsi de **choisir l'algorithme le plus adéquat aux types de données à traiter**.

Les algorithmes habituellement rencontrés peuvent être classés dans les catégories suivantes:

**Complexité O(1) :** Complexité constante. On rencontre cette complexité quand toutes les instructions sont exécutées une seule fois quelle que soit la taille n du problème.

**Complexité O(log(n)) :** Complexité logarithmique. La durée d'exécution croît légèrement avec n. Ce cas de figure se rencontre quand la taille du problème est divisée par une entité constante à chaque itération.

**Complexité O(n) :** Complexité linéaire. C'est typiquement le cas d'un programme avec une boucle de 1 à n et le corps de la boucle effectue un travail de durée constante et indépendante de n.

**Complexité O(n\*log(n)) :** Complexité n-logarithmique. Elle se rencontre dans les algorithmes où, à chaque itération, la taille du problème est divisée par une constante avec à chaque fois un parcours linéaire des données. Un exemple typique de ce genre de complexité est l'algorithme de tri "quick sort" qui, de manière récursive, trie la moitié du tableau puis l'autre moitié et ensuite "recolle" les deux morceaux. S'il n'y avait pas cette opération de "recollement" l'algorithme serait logarithmique puisqu'on divise par 2 la taille du problème à chaque étape, le fait de reconstituer à chaque fois le tableau en parcourant séquentiellement les données ajoute ce facteur n au log(n).

**Complexité O(n2) :** Complexité quadratique. C'est typiquement le cas d'algorithmes avec deux boucles imbriquées, chacune allant de 1 à n et avec le corps de la boucle interne qui est constant.

**Complexité O(n3) :** Complexité cubique. Elle est identique à la complexité quadratique mais avec ici, par exemple, trois boucles imbriquées.

**Complexité O(2n) :** Complexité exponentielle. Les algorithmes de ce genre sont dits "naïfs" car ils sont inefficaces et inutilisables dès que n dépasse 50. On rencontre typiquement ces algorithmes dans les parcours arborescents.

## Tri par sélection

### Principe

Le tri par sélection a pour objectif de comparer successivement chaque élément non trié du tableau **à tous les éléments qui le suivent** afin de l'échanger avec le minimum de ces nombres. Deux solutions sont alors possibles :

* Soit, on compare la valeur de la cellule courante, le premier élément non trié, avec les valeurs suivantes et on échange la valeur de la cellule courante avec celle dont la valeur est plus petite. Plusieurs échanges peuvent alors être réalisés avec la cellule courante durant le parcours des suivantes.
* Soit, on recherche l'élément dont la valeur est le minimum parmi les éléments non triés, c'est-à-dire les cellules qui suivent la cellule courante, et on échange ce minimum avec la valeur de la cellule courante. Un seul échange aura lieu avec la cellule courante.

Considérant un tableau d'entier, en mémoire centrale, dont la structure est la suivante :

TabNbr

Cell

Nbr

10 \*

Celui-ci ayant déjà été garni et les valeurs étant 5, 1, 7, 9, 0, 2, 4, 6, 3 et 8, la représentation graphique de ce tableau est :

TabNbr[0]

9

5

1

7

0

2

4

6

3

8

*TabNbr*

TabNbr[1]

TabNbr[2]

TabNbr[3]

TabNbr[5]

TabNbr[6]

TabNbr[7]

TabNbr[8]

TabNbr[9]

TabNbr[4]

### Première version

#### Exemple

Trier le tableau *TabNbr* en utilisant le tri par sélection dans sa première version.

**Etape 1**

On se positionne sur le premier élément du tableau, *TabNbr*[0], et on le compare successivement aux 9 éléments qui le suivent. Si cet élément est supérieur à l'un des éléments comparés, on échange le contenu des deux cellules.

1. On compare le premier élément, *TabNbr*[0], dont la valeur est 5, avec le suivant, *TabNbr*[1], dont la valeur est 1. La valeur du premier étant **supérieure** à celle du second, on **échange** les valeurs.

TabNbr[0]

9

**1**

**5**

7

0

2

4

6

3

8

TabNbr

TabNbr[1]

TabNbr[2]

TabNbr[3]

TabNbr[5]

TabNbr[6]

TabNbr[7]

TabNbr[8]

TabNbr[9]

TabNbr[4]

1. On compare le premier élément, *TabNbr*[0], dont la valeur est 1, avec le troisième élément du tableau, *TabNbr*[2], dont la valeur est 7. La valeur du premier étant **inférieure** à celle du troisième, on **n'échange pas**.
2. On compare le premier élément, *TabNbr*[0], dont la valeur est 1, avec le quatrième élément du tableau, *TabNbr*[3], dont la valeur est 9. La valeur du premier étant inférieure à celle du quatrième, on n'échange pas.
3. On compare le premier élément, *TabNbr*[0], dont la valeur est 1, avec le cinquième élément du tableau, *TabNbr*[4], dont la valeur est 0. La valeur du premier étant supérieure à celle du cinquième, on échange les valeurs.

TabNbr[0]

9

**0**

5

7

**1**

2

4

6

3

8

TabNbr

TabNbr[1]

TabNbr[2]

TabNbr[3]

TabNbr[5]

TabNbr[6]

TabNbr[7]

TabNbr[8]

TabNbr[9]

TabNbr[4]

1. Etc.
2. On compare le premier élément, *TabNbr*[0], dont la valeur est 0, avec le dixième élément du tableau, *TabNbr*[9], dont la valeur est 8. La valeur du premier étant inférieure à celle du dixième, on n'échange pas.

Après avoir comparé l'élément se trouvant dans la première cellule du tableau avec tous les autres éléments, et après l'avoir échangé, le cas échéant, avec ceux qui lui sont inférieurs, on est certain d'avoir placé dans la première cellule du tableau la valeur la plus petite.

**Etape 2**

On recommence le même cheminement avec le deuxième élément du tableau, *TabNbr*[1], dont la valeur est 5, et on le compare successivement aux 8 autres éléments.

Lors de cette comparaison, on n'échangera la valeur du deuxième élément qu'avec celle du cinquième, *TabNbr*[4], afin d'avoir le tableau suivant :

TabNbr[0]

9

0

**1**

7

**5**

2

4

6

3

8

TabNbr

TabNbr[1]

TabNbr[2]

TabNbr[3]

TabNbr[5]

TabNbr[6]

TabNbr[7]

TabNbr[8]

TabNbr[9]

TabNbr[4]

En poursuivant la comparaison, on ne trouvera aucune autre valeur inférieure à 1. On est alors certain d'avoir placé, dans la deuxième cellule du tableau, la valeur la plus petite après celle de la première cellule.

**Étapes suivantes**

On procède de la même façon pour les cellules suivantes.

Lorsqu'il ne reste plus qu'un seul élément non trié, celui-ci étant dans la dernière cellule du tableau et tous les autres éléments étant triés, on est certain que le tableau est trié.

#### Algorithme

L'analyse de cet exemple montre que deux boucles, l'une imbriquée dans l'autre, sont nécessaires :

Une première boucle (utilisant l’indice *CptI*) sert à parcourir les unes à la suite des autres les cellules du tableau à trier. La cellule en cours de traitement est la cellule courante qui ne contient pas encore la bonne valeur pour que le tableau soit trié.

TabNbr[0]

9

0

1

7

5

2

4

6

3

8

TabNbr

TabNbr[1]

TabNbr[2]

TabNbr[3]

TabNbr[5]

TabNbr[6]

TabNbr[7]

TabNbr[8]

TabNbr[9]

TabNbr[4]

*CptJ*

*CptI*

C'est dans la deuxième boucle, imbriquée dans la première (utilisant l’indice *CptJ*), que l'échange se fait. Celui-ci a lieu chaque fois que la valeur d'une cellule d’indice *CptJ* est inférieure à celle de la cellule courante d’indice *CptI*.

* **Observation**
  + Données : le tableau de nombres, la taille du tableau (*T*=10) ;
  + Entrée : rien ;
  + Sortie : rien.
* **Diagramme**
  + Opérations : trier le tableau ;
  + Variables : *TabNbr*, *CptI*, *CptJ*, *Temp* ;
  + Initialisations : *CptI* = 0, *CptJ* = *CptI*+1 ;
  + Diagramme d'actions : TriSelection.

┌─\* TriSelection

│

│ *CptI* = 0

│

│╔═ **do while** **(***CptI* < T-1**)**

│║

│║ *CptJ* = *CptI*+1

│║

│║╔═ **do while** **(***CptJ* <T**)**

│║║

│║║┌─ **if** **(***TabNbr*[*CptI*] > *TabNbr*[*CptJ*]**)**

│║║│

│║║│ *Temp* = *TabNbr*[*CptI*]

│║║│ *TabNbr*[*CptI*] = *TabNbr*[*CptJ*]

│║║│ *TabNbr*[*CptJ*] = *Temp*

│║║└──

│║║ *CptJ*++

│║╙─

│║ *CptI*++

│╙─

└──

### Deuxième version

#### Exemple

Trier le tableau TabNbr en utilisant le tri par sélection dans sa deuxième version.

**Etape 1** On recherche l'indice de la cellule contenant la valeur minimum du tableau. La valeur minimum est 0 et se trouve dans la cinquième cellule.

TabNbr[0]

9

**5**

1

7

***0***

2

4

6

3

8

TabNbr

TabNbr[1]

TabNbr[2]

TabNbr[3]

TabNbr[5]

TabNbr[6]

TabNbr[7]

TabNbr[8]

TabNbr[9]

TabNbr[4]

**Etape 2** On échange alors le premier élément du tableau, *TabNbr*[0], dont la valeur est 5, avec la valeur de la cellule contenant le minimum, *TabNbr*[4], dont la valeur est 0.

TabNbr[0]

9

**0**

1

7

**5**

2

4

6

3

8

TabNbr

TabNbr[1]

TabNbr[2]

TabNbr[3]

TabNbr[5]

TabNbr[6]

TabNbr[7]

TabNbr[8]

TabNbr[9]

TabNbr[4]

**Etape 3** On recommence le même cheminement en recherchant l'indice de la cellule contenant la valeur minimum parmi les cellules qui suivent la première cellule du tableau. La valeur minimum est 1 et se trouve dans la deuxième cellule.

TabNbr[0]

9

0

***1***

7

5

2

4

6

3

8

TabNbr

TabNbr[1]

TabNbr[2]

TabNbr[3]

TabNbr[5]

TabNbr[6]

TabNbr[7]

TabNbr[8]

TabNbr[9]

TabNbr[4]

L'indice du minimum étant celle de la cellule en cours de traitement, il est inutile de faire l'échange.

**Etape 4** On recommence le même cheminement en recherchant l'indice de la cellule contenant la valeur minimum parmi les cellules qui suivent la deuxième cellule du tableau. La valeur minimum est 2 et se trouve dans la sixième cellule.

TabNbr[0]

9

0

1

**7**

5

***2***

4

6

3

8

TabNbr

TabNbr[1]

TabNbr[2]

TabNbr[3]

TabNbr[5]

TabNbr[6]

TabNbr[7]

TabNbr[8]

TabNbr[9]

TabNbr[4]

On échange alors le troisième élément du tableau, *TabNbr*[2], dont la valeur est 7, avec la valeur de la cellule contenant le minimum, *TabNbr*[5], dont la valeur est 2.

TabNbr[0]

9

0

1

**2**

5

**7**

4

6

3

8

TabNbr

TabNbr[1]

TabNbr[2]

TabNbr[3]

TabNbr[5]

TabNbr[6]

TabNbr[7]

TabNbr[8]

TabNbr[9]

TabNbr[4]

**Étapes suivantes**

On procède de la même façon pour les cellules suivantes.

Lorsqu'il ne reste plus qu'un seul élément non trié, celui-ci étant dans la dernière cellule du tableau et tous les autres éléments étant triés, on est certain que le tableau est trié.

#### Algorithme

L'analyse de cet exemple montre que deux boucles sont nécessaires. L'une d'entre elle est dans le module principal et l'autre est dans le module de recherche du minimum.

La première boucle sert à parcourir les cellules du tableau les unes à la suite des autres à l’aide de l’indice CptI. Elle fait appel au module de recherche de l'indice de la cellule contenant la valeur minimum parmi les valeurs non encore triées. Si l'indice de ce minimum n'est pas celui de la cellule courante, elle échange les deux valeurs de ces cellules. Il n'est pas nécessaire d'aller jusqu'à la dernière cellule puisque celle-ci contiendra d'office la bonne valeur lors du dernier passage dans la boucle.

La deuxième boucle, dans le module de recherche, permet de parcourir le reste du tableau à l’aide de l’indice CptJ afin de trouver l'indice de l'élément contenant la valeur minimum (IndMin) parmi les cellules qui suivent les cellules dont les valeurs sont déjà triées.

* **Observation**
  + Donnée : le tableau de nombres, la taille du tableau (*T* = 10) ;
  + Entrée : rien ;
  + Sortie : rien.
* **Diagramme**
  + Opérations : trier le tableau ;
  + Variables : *TabNbr*, *CptI*, *CptJ*, *Temp* ;
  + Initialisation : *CptI* = 0, *CptJ* = *CptI*+1, *IndMin* ;
  + Diagramme d'actions : TriSelection.

┌─\* TriSelection

│

│ *CptI* = 0

│

│╔═ **do while** **(***CptI* <T-1**)**

│║

│║ *IndMin* = *CptI*

│║

│║ *CptJ* = *CptI*+1

│║ ╔═ **do while** **(***CptJ* <T**)**

│║ ║

│║ ║┌─ **if** **(***TabNbr*[*CptJ*] < *TabNbr*[*IndMin*]**)**

│║ ║│

│║ ║│ *IndMin = CptJ*

│║ ║└──

│║ ║ *CptJ*++

│║ ╙─

│║┌─ **if** **(***CptI* ≠ *IndMin***)**

│║│*Temp* = *TabNbr*[*CptI*]

│║│ *TabNbr*[*CptI*] = *TabNbr*[*IndMin*]

│║│ *TabNbr*[*IndMin*] = *Temp*

│║└──

│║ *CptI*++

│╙─

└──

## Tri par échange ou tri à bulle

### Principe

Le tri à bulle a pour objectif de comparer chaque élément du tableau **avec l’élément qui le suit**. Si l’ordre n’est pas correct, on permute ces deux éléments. On recommence cette démarche jusqu’à ce que l’on n’ait plus aucune permutation à effectuer.

Les éléments les plus grands "remontent" ainsi peu à peu vers "la surface" c'est-à-dire à la fin du tableau, ce qui explique la charmante dénomination de "tri à bulle".

A la différence du tri par sélection, le tri à bulle nécessite l'utilisation d'un booléen, un flag permettant d’indiquer si une permutation a été effectuée ou pas. En effet, comme on compare chaque élément directement avec celui qui le suit, on ne sait jamais à l'avance combien de parcours du tableau seront nécessaires. En fait, à priori, on devra effectuer le tri jusqu’à ce qu’il n’y ait plus d’éléments mal placés.

A chaque parcours du tableau, si l'élément courant (celui en cours de traitement) est plus grand que l'élément qui le suit, ils sont intervertis. Si une permutation est effectuée durant ce parcours, le booléen est mis à VRAI de manière à savoir que le tableau n'est pas encore trié complètement. Il suffit qu’il y ait eu une seule permutation pour qu’il faille tout recommencer encore une fois. Il est bien sûr primordial de remettre le booléen à FAUX à chaque début du parcours du tableau.

Lorsqu'aucune permutation n'a été effectuée lors du balayage du tableau, cela signifie que le tableau est trié. Le traitement peut être arrêté.

### Exemple

En reprenant le tableau initial de l'exemple du tri par sélection, le tri à bulle se déroule de la manière suivante :

TabNbr[0]

9

5

1

7

0

2

4

6

3

8

*TabNbr*

TabNbr[1]

TabNbr[2]

TabNbr[3]

TabNbr[5]

TabNbr[6]

TabNbr[7]

TabNbr[8]

TabNbr[9]

TabNbr[4]

**Etape 1**

On commence par mettre le flag à VRAI de manière à rentrer dans la boucle contenant le parcours du tableau.

On parcourt le tableau de manière à permuter les éléments qui sont plus grands que leur suivant avec celui-ci :

1. On compare le premier élément du tableau, *TabNbr*[0], dont la valeur est 5, avec son suivant, *TabNbr*[1], dont la valeur est 1. Comme l'élément courant est plus **grand** que son suivant, on **permute** et on met le booléen à VRAI.

TabNbr[0]

9

**1**

**5**

7

0

2

4

6

3

8

TabNbr

TabNbr[1]

TabNbr[2]

TabNbr[3]

TabNbr[5]

TabNbr[6]

TabNbr[7]

TabNbr[8]

TabNbr[9]

TabNbr[4]

1. On compare le deuxième élément du tableau, *TabNbr*[1], dont la valeur est 5, avec son suivant, *TabNbr*[2], dont la valeur est 7. Comme l'élément courant est plus **petit** que son suivant, on **ne permute pas**.
2. On compare le troisième élément du tableau, *TabNbr*[2], dont la valeur est 7, avec son suivant, *TabNbr*[3], dont la valeur est 9. Comme l'élément courant est plus petit que son suivant, on ne permute pas.
3. On compare le quatrième élément du tableau, TabNbr[3], dont la valeur est 9, avec son suivant, *TabNbr*[4], dont la valeur est 0. Comme l'élément courant est plus grand que son suivant, on permute et on met le booléen à VRAI.

TabNbr[0]

**0**

1

5

7

**9**

2

4

6

3

8

TabNbr

TabNbr[1]

TabNbr[2]

TabNbr[3]

TabNbr[5]

TabNbr[6]

TabNbr[7]

TabNbr[8]

TabNbr[9]

TabNbr[4]

1. Etc.
2. On compare le neuvième élément du tableau, *TabNbr*[8], dont la valeur est 9, avec son suivant, *TabNbr*[9], dont la valeur est 8. Comme l'élément courant est plus grand que son suivant, on permute et on met le booléen à VRAI.

TabNbr[0]

0

1

5

7

2

4

6

3

**8**

**9**

TabNbr

TabNbr[1]

TabNbr[2]

TabNbr[3]

TabNbr[5]

TabNbr[6]

TabNbr[7]

TabNbr[8]

TabNbr[9]

TabNbr[4]

**Etape 2**

Comme le booléen a été mis au moins une fois à vrai, on doit recommencer le parcours du tableau, et ainsi de suite.

Quand le booléen est resté à FAUX durant tout le parcours du tableau, on peut arrêter le traitement puisqu'on est certain que le tableau est complètement trié.

### Algorithme

* **Observation**
  + Donnée : le tableau de nombres, la taille du tableau (*T* = 10) ;
  + Entrée : rien ;
  + Sortie : rien.
* **Diagramme**
  + Opérations : trier le tableau ;
  + Variables : *TabNbr*, *CptI*, *Permut*, *Temp* ;
  + Initialisation : *CptI* = 0, *Permut* = VRAI;
  + Diagramme d'actions : TriBulle.

┌─\* TriBulle

│

│ *Permut* = VRAI

│

│╔═ **do while** **(***Permut* = VRAI**)**

│║

│║ *CptI* = 0

│║ *Permut* = FAUX

│║

│║╔═ **do while** **(***CptI* <T-1**)**

│║║

│║║┌─ **if** **(***TabNbr*[*CptI*] > *TabNbr*[*CptI+1*]**)**

│║║│

│║║│ *Temp* = *TabNbr*[*CptI*]

│║║│ *TabNbr*[*CptI*] = *TabNbr*[*CptI+1*]

│║║│ *TabNbr*[*CptI+1*] = *Temp*

│║║│ *Permut* = VRAI

│║║└──

│║║ *CptI*++

│║╙─

│╙─

└──

# Les fichiers[[1]](#footnote-1)

## Pourquoi des fichiers ?

Lors de l'exécution d'un programme, le processeur ne manipule que des objets situés en mémoire centrale, tels que les variables, les structures, les tableaux, les chaînons, les nœuds etc. Les caractéristiques générales de la mémoire centrale imposent que de tels objets ne puissent, en général, exister avant le chargement du programme, ni après la fin de son exécution. Leur durée de vie est donc limitée à la durée d'exécution du programme. Ces objets sont dits internes au programme.

Par opposition, les objets qui ne sont pas internes sont appelé externes. On peut distinguer deux objectifs dans l'utilisation des objets externes par un programme :

* Le premier est l'échange d'informations, que ce soit avec l'homme ou avec une autre machine ;
* Le second est de permettre au programmeur de disposer d'informations dont la durée de vie peut s'étendre au-delà de la durée de vie du programme. Il s'agit donc de la mémorisation à long terme d'informations.

### L'échange d'informations

Afin de permettre cet échange d'informations, il est nécessaire de trouver un langage commun entre les deux partenaires de l'échange.

Si l'échange a lieu entre l'homme et la machine, cela implique la transformation de la représentation interne des données en une forme compréhensible par l'homme, et réciproquement.

Lors de l'échange entre machines, selon la nature des machines qui communiquent, une conversion de la représentation interne des informations peut être nécessaire afin de rendre la communication possible. Cet aspect de l'échange d'informations étant principalement lié au matériel et non à l'algorithmique, il n'est pas traité dans le cadre de ce cours.

### La mémorisation à long terme

Il s'agit du cas où les données existent avant le chargement du programme, ou lorsqu'elles doivent être conservées après la fin de son exécution.

Contrairement au cas précédent, c'est la même machine qui crée ces données et les reprend ultérieurement. On peut dire effectivement qu'il s'agit encore de pouvoir échanger des informations mais cette fois entre des programmes qui s'exécutent sur la même installation. Le problème de conversion énoncé ci-dessus ne se pose plus. Les données, en général, peuvent et doivent être mémorisées en respectant leur représentation interne.

Un problème peut cependant se poser pour les données de type pointeur. En effet, s'il s'agit de la mémoire centrale, cela signifie que le pointeur désigne un objet interne au programme, et n'a de signification que pour cette exécution. Le conserver sous cette forme après la fin de l'exécution de ce programme n'a pas sens. Sa valeur doit être remplacée, dans l'objet externe, par une information qui désigne l'emplacement de la donnée interne pointée.

## La notion de fichier

### Introduction

Le mot **fichier** est parfois utilisé de façon ambiguë, car il désigne tantôt l'objet externe mémorisé sur un support magnétique (bande ou disque), tantôt l'entité manipulée par le programme.

Pour éviter cette ambiguïté, le premier pourrait être appelé fichier physique et le second fichier logique. En général nous n'utiliserons le terme fichier que pour désigner la façon dont le programme manipule l'objet, indépendamment de l'objet lui-même et des contraintes physiques d'implantation comme des caractéristiques du périphérique qui le supporte.

### Définitions

Un fichier est une collection d'enregistrements logiques, éventuellement structurée, sur laquelle le programme peut exécuter un ensemble d'opérations. Il correspond à la représentation logique de le l'élément externe situé sur un support physique.

*Exemple :* Le fichier des abonnés inscrits à une n*ewsletter* aura la structure suivante :

EnregAbonné

(S\*)

*NomAbon*

*MailAbon*

*NatAbon*

FiAbonnés

Un **enregistrement logique** est l'ensemble minimum de données qui peut être manipulé par une seule opération élémentaire du fichier.

Il est important de rappeler qu'un enregistrement logique n'est pas nécessairement identique à un enregistrement physique. En effet, la représentation logique des enregistrements d'un fichier n'est pas liée à la manière dont celui-ci est organisé sur le support physique.

Un enregistrement logique est souvent constitué de diverses données élémentaires, appelées **champ**. Chaque champ est caractérisé par :

* Son type (numérique, alphabétique ou alphanumérique) ;
* Sa longueur (fixe ou variable) ;
* Sa représentation interne (char, int, long, double, …).

Ces données élémentaires ne sont accessibles individuellement par le programme que lorsque l'enregistrement logique est recopié dans un objet interne au programme. Dans le cas du langage C, l'enregistrement logique est recopié dans une variable de type structure dont les champs correspondent à ceux de l'enregistrement.

Les opérations sur le fichier permettent, entre autres, de **transférer** le contenu d'un tel enregistrement entre la mémoire interne du programme et l'objet externe (le fichier) qui lui est associé.

Il est également possible de créer un fichier, de faire une **recherche** d'un ou plusieurs enregistrements, de **modifier** la valeur d'un ou plusieurs champs, d'**ajouter** ou de **supprimer** un champ ou un ou plusieurs enregistrements.

Pour identifier de manière unique un enregistrement par rapport à un autre, soit on utilise un champ déjà existant dont la valeur est identifiante, soit on ajoute un champ dont la valeur sera unique. Le champ servant d’identifiant est appelé **clé**.

## Types d'enregistrements

Les enregistrements peuvent être de deux types :

* Format **fixe**, c'est-à-dire que chacun des champs est présent et a la même taille pour tous les enregistrements du fichier.
* Format **variable**, c'est-à-dire que la taille de l'enregistrement est variable :
  + Parce que la taille d'un champ varie d'un enregistrement à l'autre ;
  + Parce que le type d'un champ varie d'un enregistrement à l'autre ;
  + Parce que certains champs sont absents dans des enregistrements.

## Représentations d'un enregistrement logique

Soit un fichier dont les enregistrements contiennent les informations concernant un logiciel :

* Son nom ;
* Sa version ;
* Son fournisseur.

Les différentes techniques utilisées pour représenter les données d'un enregistrement sont les suivantes.

#### Technique positionnelle

Dans le cas de la technique positionnelle, chaque **champ** d'un enregistrement est **de** **longueur fixe**.

Borland C++ Builder

2006

Borland

*Exemple :*

#### Technique relationnelle

La technique relationnelle permet de représenter des **champs de longueur variable**. Un **caractère spécial** est alors utilisé pour délimiter les champs.

Borland C++ Builder

2006

Borland

#

#

*Exemple :*

#### Technique indexée

Cette technique a pour spécificité de donner accès aux champs via des **pointeurs**.

*Exemple :*

Borland C++ Builder

2006

Borland

#### Technique étiquetée

Il s'agit, dans ce cas, de faire précéder chaque champ d'un enregistrement par une **étiquette**.

*Exemple :*

Borland C++ Builder

2006

Borland

v

f

n

## Types de fichiers

#### Permanents

Ce sont des fichiers qui ont une longue durée de vie.

* Signalétique Fichier permettant de conserver des informations concernant les entités logiques (étudiants, professeurs, local, etc.) et de les mettre à jour via des programmes spécifiques ;

*Exemple :* fichier des étudiants.

* Situation Fichier reflétant l'état actuel des informations à traiter ;

*Exemple :* fichier d'état des stocks.

* Historique – Archives Fichier contenant les résultats antérieurs des traitements.

*Exemple :* fichier des sites internet visités antérieurement.

#### Mouvements

La durée de vie de ce type de fichiers est limitée à une ou deux utilisations. Ils servent uniquement à transférer les informations.

#### Travail

Ces fichiers ont un rôle de lien entre deux programmes.

## Différentes organisations de fichier

L'**organisation de fichier** correspond à la manière dont les enregistrements sont mémorisés les uns par rapport aux autres dans le fichier. Elle est déterminée par la nature des opérations que l'on peut effectuer sur le fichier.

Il s'agit toujours d'une vue logique du fichier et pas encore de la manière dont ils sont écrits sur le support.

#### L'organisation séquentielle

Les fichiers organisés séquentiellement sont appelés **fichiers séquentiels**. Dans ce cas, les enregistrements sont rangés les uns à la suite des autres, consécutivement.

Les opérations se résument, essentiellement, en la lecture de l'enregistrement suivant ou l'écriture d'un nouvel enregistrement en fin du fichier séquentiel.

Bien souvent, pour affirmer le caractère séquentiel du fichier, il n'est pas possible de mélanger les lectures et les écritures. Le fichier est alors un fichier séquentiel en lecture ou un fichier séquentiel en écriture.

Le fichier séquentiel en lecture est initialement positionné sur le premier enregistrement logique. Chaque opération de lecture (lire ou read) transfère dans une zone interne au programme un enregistrement du fichier, et prépare le fichier pour le positionner sur l'enregistrement suivant. On dispose souvent d'une opération complémentaire qui permet de savoir s'il y a encore des enregistrements à lire dans le fichier (fin\_de\_fichier ou end\_of\_file), et éventuellement une opération permettant le retour au début du fichier (rembobiner ou rewind).

Le fichier séquentiel en écriture peut être initialement vide, ou positionné après le dernier enregistrement déjà dans le fichier. Chaque opération d'écriture (écrire ou write) rajoute un enregistrement dans le fichier depuis une zone interne au programme, et positionne le fichier après cet enregistrement. L'écriture d'un enregistrement se fait donc toujours après ceux qui sont déjà dans le fichier.

Il est à noter que la caractéristique essentielle d'un fichier séquentiel est que les enregistrements sont lus dans l'ordre où ils ont été écrits. Aucune conversion des données n'est appliquée, soit parce qu'elle est faite par le programme, soit parce qu'elle n'est pas nécessaire.

Il est à remarquer que certains langages tels que le C permettent d'accéder au fichier aussi bien en lecture qu'en écriture malgré que les fichiers soient de type séquentiel. Pour ce faire, il faut faire en sorte d'ouvrir le fichier en mode binaire ("wb" comme mode d'accès lors du fopen) de manière à considérer les enregistrements comme un ensemble d’octets.

#### L'organisation relative (ou fichier à accès direct)

Un fichier en organisation relative se compose d'un certain nombre d'enregistrements de taille fixe. Chaque enregistrement porte un **numéro relatif au début du fichier**, ce numéro allant de 1 (ou 0) à N.

Une fonction est associée au fichier qui, à partir du numéro d'un enregistrement, calcule la position de l'enregistrement dans le fichier.

#### L'organisation chaînée

L'organisation chaînée permet de prévoir un ou plusieurs **liens logiques** entre les enregistrements.

#### L'organisation indexée

Dans cette organisation, chaque enregistrement est identifié par une clé et à chaque clé est associé un numéro d'enregistrement qui renvoie à l'enregistrement correspondant. Pour faire le lien entre la clé et le numéro de l'enregistrement, une table des clés, appelée **index**, est utilisée.

#### L'organisation calculée

La position de l'enregistrement dans le fichier est le résultat d'un **calcul sur la clé**. Pour cela on utilise les fonctions de **hachage**.

#### L'organisation arborescente

Il s'agit du même type d'organisation que l'organisation chaînée, à la différence que la liaison entre les enregistrements se fait suivant une **logique arborescente**.

## Le lien entre la représentation logique et physique

### Représentation interne d'un fichier

La représentation interne d'un fichier logique dans le programme est une structure de données (appelée parfois *bloc de contrôle* *de données* ou encore *Data Control Bloc*). Elle contient quatre types d'informations :

* Les *attributs de la liaison* permettent de savoir quel est l'état de la liaison (ouverte ou fermée), les variables internes de gestion de la liaison ainsi que les opérations autorisées sur le fichier ;
* Le *descripteur de l'objet externe* permet de localiser l'objet externe. S'il s'agit d'un périphérique, il faut connaître sa nature et son adresse. S'il s'agit d'un fichier physique, il faut connaître en plus la localisation de l'objet externe sur le périphérique qui le supporte ;
* Les *procédures d'accès* sont en fait la concrétisation des opérations du fichier sur l'objet externe qui lui est relié. Il faut noter que, vu du programmeur, il s'agit toujours d'un ensemble d'opérations bien définies, mais l'implantation proprement dite de ces opérations dépend de la nature de l'objet externe et de son support.
* Les *tampons d'entrées-sorties* permettent d'assurer l'interface entre les enregistrements logiques et les enregistrements physiques. Nous avons dit que les opérations du fichier manipulaient des enregistrements logiques tels que les définit le programmeur. Par contre, les entrées-sorties physiques ont des contraintes de taille et d'organisation qui dépendent du support. Les tampons permettent de regrouper plusieurs enregistrements logiques par enregistrement physique lors des écritures, et de les dégrouper lors de la lecture. Ils servent aussi à améliorer les performances du transfert entre le support externe et la mémoire centrale.

Certaines des informations énoncées ci-dessus sont statiques, c'est-à-dire connues à la compilation. Ce sont celles qui sont directement liées à la nature du fichier tel que l'utilise le programmeur. La plupart sont en fait dynamiques car elles dépendent du fichier et de l'objet externe avec lequel il est relié.

### L'établissement de la liaison

Un fichier (logique) peut être vu comme un objet interne au programme. Pour pouvoir effectuer des opérations sur le fichier, il faut qu'il soit relié à un objet externe. Il n'est pas intéressant que cette liaison soit prise en compte par l'éditeur de liens pour diverses raisons qui ne seront pas énoncées ici.

La liaison est établie par l'ouverture et est supprimée par la fermeture du fichier. Deux opérations sont donc nécessaires :

* L'opération d'*ouverture* (on dit aussi *open*) sur un fichier permet d'établir la liaison avec un objet externe suivant les informations contenues dans sa définition ;
* L'opération de *fermeture* (on dit aussi *close*) sur un fichier ouvert, permet de rompre temporairement ou définitivement la liaison avec l'objet externe.

Les opérations autres que l'ouverture (lecture, écriture, etc.) ne peuvent être appliquées que sur des fichiers ouverts, puisqu'elles ont pour conséquence des accès à un objet externe qui doit donc avoir été associé au fichier.

Il y a toutefois une exception à cette règle d'établissement dynamique de la liaison. La plupart des systèmes établissent trois liaisons spécifiques préalablement à l'exécution du programme, pour les trois fichiers standards :

* SYSIN pour l'entrée des données provenant de l'utilisateur ;
* SYSOUT pour la sortie des données vers l'utilisateur ;
* SYSERR pour la sortie des messages d'erreurs vers l'utilisateur.

## Formats d'enregistrement physiques

Comme il a été précisé dès le début de ce chapitre, les enregistrements physiques ne correspondent pas nécessairement à un enregistrement logique. La structure des enregistrements physiques dépend de l'environnement matériel et logiciel.

Parmi les divers formats possibles, en voici quelques-uns :

#### Stream ou continu

Les enregistrements sont délimités par des caractères spéciaux qui permettent de les dissocier comme le Carriage Return (ENTER).

*Exemple :*

Enregistrement

Enregistrement

Enregistrement

…

#### Longueur fixe

Tous les enregistrements ont la même longueur. Cette longueur est spécifiée à l'ouverture du fichier ou lors de l'écriture ou de la lecture d'un enregistrement.

*Exemple :*

Enregistrement

Enregistrement

Enregistrement

…

35 bytes

35 bytes

35 bytes

#### Longueur variable

La longueur d'un enregistrement varie d'un enregistrement à l'autre. Chacun d'eux contient donc une zone de donnée de longueur variable et une zone appelée Count Field précisant la longueur de la zone de donnée correspondante.

Le champ Count Field est en général géré par le système, mais ce n'est pas le cas en langage C.

*Exemple :*

Enregistrement

Enregistrement

Enregistrement

…

45 bytes

30 bytes

Count

Field

Zone de donnée

Enregistrement

35 bytes

avec

#### Format bloqué

Un bloc est l'unité de transfert entre le support et la mémoire centrale. Celui-ci peut contenir un ou plusieurs enregistrements. Le nombre d'enregistrements qu'il peut contenir est appelé le facteur de blocage ou de groupage.

Les blocs peuvent être en format fixe ou variable.

*Exemple :*

Enregistrement

Enregistrement

Enregistrement

Bloc

#### Format étendu - spanned

En sachant que le fichier est organisé par blocs, un enregistrement peut occuper plusieurs blocs.

Il est donc nécessaire d'utiliser un champ qui permette de savoir si le bloc est le premier occupé par l'enregistrement ou si c'est le dernier.

*Exemple :*

Flag

Zone de donnée

Bloc 1

Flag

Zone de donnée

Bloc 3

…

## En programmation …

De plus amples détails quant à l’utilisation des fichiers en programmation seront donnés au cours. Les notations assosiées à leur manipulation seront également précisées lors de la présentation de quelques exemples.

# Les Piles et les Files

## Introduction

Ce sont des **ensembles ordonnés d'éléments**, mais qui ne permettent l'accès qu'à un seul élément à la fois.

On utilise souvent le nom générique de pile pour les piles et les files, un seul nom existant en anglais : stack.

* Les **piles**, stack **LIFO** (Last In First Out) correspondent à une pile d'assiettes : on prend toujours l'élément supérieur, le dernier empilé.
* Les **files**, stack **FIFO** (First In First Out) correspondent aux files d'attente : on prend toujours le premier élément, donc le plus ancien.

Les piles et les files servent notamment à mémoriser des informations en attente de traitement. Elles permettront une clarification des algorithmes quand effectivement on n'a pas besoin d'accéder directement à tous les éléments. Elles sont souvent associées à des algorithmes récursifs.

Comme il n'y a pas de structures spécifiques à ces ensembles ordonnés prévues dans les langages de programmation, il faut les créer de toutes pièces.

* Pour les **piles**, on utilisera un ***tableau*** unidimensionnel en cas de piles de hauteur maximale prévisible (la hauteur de la pile est mémorisée par une variable entière), ou une ***liste*** en cas de longueur très variable.
* Pour les **files**, l'utilisation d'un ***tableau*** nécessite deux variables : la position du premier et celle du dernier. La suppression du premier élément ne se fait pas par décalage des suivants mais en incrémentant la variable indiquant le premier. La gestion de ce tableau est alors un peu plus complexe que pour les piles. L'utilisation d'une ***liste*** pour une file par contre est aussi simple que pour une pile.

## Les piles

### Qu'est-ce qu'une pile ?

#### Définition

Une pile est un **ensemble ordonné d'éléments**, initialement vide. Les éléments sont empilés les uns sur les autres de telle manière que le seul élément accessible est celui du **sommet** de la pile.

La manière de procéder pour stocker et récupérer un élément dans une pile se fait selon le principe du "Last In, First Out – **LIFO**", c'est-à-dire que le dernier élément inséré dans la pile est le premier à pouvoir être récupéré.

#### Opérations

Les opérations sur une pile sont les suivantes :

* Initialiser la pile ;
* Vérifier si la pile est vide ;
* Obtenir la valeur au sommet de la pile ;
* Désempiler : retirer la valeur au sommet de la pile ;
* Empiler : insérer une valeur au sommet de la pile (s'il reste de la place).

#### Représentation

La seule structure déjà vue permettant de représenter une pile est le **tableau à simple indice**. Une pile contenant au maximum T éléments de type numérique sera représentée comme suit :

20

5

10

15

?

?

?

?

?

?

Pile[0]

Pile[1]

Pile[2]

Pile[3]

Pile[4]

Pile[5]

Pile[6]

Pile[7]

Pile[8]

Pile[9]

?

?

?

?

?

?

15

10

5

20

Pile[0]

Pile[1]

Pile[2]

Pile[3]

Pile[4]

Pile[5]

Pile[6]

Pile[7]

Pile[8]

Pile[9]

***Sommet***

Pile

Cell

Nbr

T \*

OU

### Algorithmes

#### Empiler

Si on considère l'algorithme d'empilement comme étant un module, celui-ci reçoit alors plusieurs arguments : la pile, sa taille, le sommet actuel ainsi que l'élément à empiler obtenu dans le module appelant. Il devra renvoyer la pile modifiée ainsi que le nouveau sommet.

* **Observation** 
  + Données : le tableau représentant la pile, la taille du tableau (T) ;
  + Entrée : aucune ;
  + Sortie :

message s'il y a débordement.

* **Diagramme** 
  + Opérations : empilement, sortie ;
  + Variables :
  + Pile, Sommet, Element ;
  + Initialisation : aucune;
  + Diagramme d'actions : Empiler.

┌─\* Empiler

│

│┌─ **if (***Sommet* < T-1**)**

││

││ *Sommet* ++

││ *Pile*[*Sommet*] = *Element*

││

│├─ **else**

││

││ **Sortir** "Il n'y a plus de place dans la pile !"

│└──

└──

#### Désempiler

Si on considère l'algorithme de désempilement comme étant un module, celui-ci reçoit alors plusieurs arguments : la pile et le sommet actuel. Il devra renvoyer la pile modifiée, le nouveau sommet ainsi que l'élément désempilé.

* **Observation** 
  + Données :

le tableau représentant la pile, la taille du tableau (T) ;

* + Entrée : aucune ;
  + Sortie : message s'il n’y a plus d’élément.

┌─\* Désempiler

│

│┌─ **if (***Sommet* < 0**)**

││

││ **Sortir** "Il n'y a plus d'élément dans la pile !"

││

│├─ **else**

││

││ *Element* = *Pile*[*Sommet*]

││ *Sommet* --

│└──

└──

* **Diagramme** 
  + Opérations :
  + désempilement, sortie ;
  + Variables :
  + Pile, Sommet, Element ;
  + Initialisation : aucune;
  + Diagramme d'actions :
  + Désempiler

### Exemples[[2]](#footnote-2)

* La **pile d'appel** (souvent appelée « la pile » tout court, parfois « pile système ») est, sur la plupart des architectures de microprocesseurs, une pile particulière dans laquelle sont poussées tout ou partie des paramètres d'appel des procédures ou fonctions, ainsi que l'adresse de retour. Par ailleurs, on y crée un espace pour des variables locales. La pile est ainsi formée de **cadres de piles** (stack frames) comprenant pour chaque procédure en cours d'appel imbriqué ses paramètres, ses variables locales et son point de retour. La pile système sert à sauver le contexte (valeur des registres, point de retour...) lors des exceptions et des interruptions.
* **Focus** est le premier **processeur** 32 bits au monde. Il utilise une architecture basée sur la **pile**, possède 220 instructions (une partie en 32 bits, d'autres en 16 bits), un modèle de mémoire utilisant des segments[[3]](#footnote-3), et aucun [registre mémoire](http://fr.wikipedia.org/wiki/Registre_%28informatique%29) d'usage général visible pour le programmeur.

## Les files

### Qu'est-ce qu'une file ?

#### Définition

Une file est un **ensemble ordonné d'éléments**, initialement vide. Les éléments sont insérés à une extrémité (la **queue**) et sont récupérés à l'autre extrémité (la **tête**) de la file.

La manière de procéder pour stocker et récupérer un élément dans une file se fait selon le principe du "First In, First Out – **FIFO**", c'est-à-dire que le premier élément inséré dans la file est le premier à pouvoir être récupéré.

#### Opérations

Les opérations sur une file sont les suivantes :

* Initialiser la file ;
* Vérifier si la file est vide ;
* Piquer : obtenir la valeur de l'élément de tête ;
* Défiler : supprimer l'élément de tête ;
* Enfiler : ajouter un élément en queue de liste.

#### Représentation

La seule structure déjà vue permettant de représenter une file est le **tableau à simple indice**. D'autres représentations sont possibles, par exemple une liste chaînée.

Voici une file contenant au maximum T éléments :

File

Cell

Nbr

T \*

?

?

10

15

5

25

20

?

?

?

File[0]

File[1]

File[2]

File[3]

File[4]

File[5]

File[6]

File[7]

File[8]

File[9]

Queue

Tête

### Algorithmes

#### Enfiler

Si on considère l'algorithme d'enfilement comme étant un module, celui-ci reçoit alors plusieurs arguments : la file, sa taille, la queue ainsi que l'élément à empiler obtenu dans le module appelant. Il devra renvoyer la file modifiée ainsi que la nouvelle queue.

* **Observation** 
  + Données : le tableau représentant la file, la taille du tableau (T) ;
  + Entrée : aucune ;
  + Sortie : aucune.
* **Diagramme** 
  + Opérations : enfilement, sortie ;
  + Variables : *File*, *Queue*, *Element* ;
  + Initialisation : aucune;
  + Diagramme d'actions : Enfiler.

┌─\* Enfiler

│

│┌─ **if (***Queue =* T**)**

││

││ *Queue* = 0

││

│├─ **else**

││

││ *Queue* ++

│└──

│ *File*[*Queue*] = *Element*

└──

#### Défiler

Si on considère l'algorithme de défilement comme étant un module, celui-ci reçoit alors plusieurs arguments : la file et la tête actuelle. Il devra renvoyer la file modifiée, la nouvelle tête ainsi que l'élément défilé.

* **Observation** 
  + Données : le tableau représentant la file, la taille du tableau (T) ;
  + Entrée : aucune ;
  + Sortie : aucune.
* **Diagramme** 
  + Opérations : défilement, sortie ;
  + Variables : *File*, *Tête*, *Element* ;
  + Initialisation : aucune;
  + Diagramme d'actions : Défiler.

┌─\* Défiler

│ *Element* = *TabPile*[*Tête*]

│

│┌─ **if (***Tête* = T**)**

││

││ *Tête* = 0

││

│├─ **else**

││

││ *Tête ++*

│└──

└──

### Exemples[[4]](#footnote-4)

* Les **serveurs d'impression**, qui doivent traiter les requêtes dans l'ordre dans lequel elles arrivent, et les insèrent dans une file d'attente (ou une queue).
* Une **file d'attente de message** est une technologie de programmation utilisée pour la communication interprocessus ou la communication de serveur à serveur. En anglais, on parle de « Message-Oriented Middleware » ou « Mom ».

Les files d'attente de message fournissent des liaisons asynchrones normalisées. Elles ont pour but que l'expéditeur et le récepteur du message ne soient pas contraints de s'attendre l'un l'autre. Des messages placés dans la file d'attente sont stockés, jusqu'à ce que le destinataire les recherche. L'expéditeur n'a pas à attendre que le récepteur commence à traiter son message, il poste son information et peut passer à autre chose.

Beaucoup de réalisations de files d'attente de message sont créées pour les besoins internes des systèmes d'exploitation. Elles sont indispensables pour la synchronisation ou le travail multitâche des processus afin de pouvoir accorder du temps-machine à chaque tâche, sans en privilégier aucune.

D'autres réalisations de file d'attente permettent une communication entre différents systèmes informatiques, connectant plusieurs applications, ou plusieurs systèmes d'exploitation. Ces systèmes de synchronisation de messages fournissent typiquement une fonctionnalité de persistance pour s'assurer que les messages ne soient pas perdus en cas d'échec du système. On parle d'[Intergiciel orienté message](http://fr.wikipedia.org/w/index.php?title=Intergiciel_orient%C3%A9_message&action=edit" \o "Intergiciel orienté message), c'est-à-dire un gestionnaire de files d'attente.

# Les Tableaux à double indice

## Pourquoi des tableaux à plusieurs dimensions ?

Les cellules d'un tableau à une dimension sont indicées de manière consécutive. La représentation graphique du tableau se fait à l'aide d'une suite de cellules les unes contre les autres. Cependant, il est possible de disposer les cellules d'un tableau selon des grilles (tableaux à deux dimensions), des cubes (tableaux à trois dimensions), etc.

En effet, l'algorithmique étant à notre service pour gérer des problèmes divers, il est nécessaire de prévoir une modélisation plus complexe que la simple juxtaposition de cellules.

En prenant l'exemple de la modélisation d’un jeu de dames, et du déplacement des pions sur le damier. Avec les outils que nous avons abordés jusqu'ici, le plus simple serait évidemment de modéliser le damier sous la forme d’un tableau. Chaque case du damier étant représentée par une cellule du tableau. La valeur de la cellule modélisant une case vide serait, par exemple, 0, et celle d'une case occupée par un pion serait 1.

Chaque cellule du tableau correspondrait à une case du damier : les cellules d'indice 0 à 7 pour la première ligne, 8 à 15 pour la deuxième ligne, et ainsi de suite jusqu’à avoir représenté les 64 cases du damier.

Sur base de cette représentation, on peut écrire un algorithme permettant de simuler le jeu de dames, mais le fait d'utiliser un tableau à simple indice ne facilite pas la clarté de l’algorithme. Il serait évidemment plus simple de modéliser un jeu de dame à l'aide d'un damier. D'où, l'algorithmique offre la possibilité d'utiliser des tableaux dans lesquels les valeurs ne sont pas repérées par un seul, mais par **deux indices**. Il s'agit alors d'un **tableau à deux dimensions**.

Il est intéressant de noter qu'il n'y a pas de limite au nombre de dimensions. Si vous avez compris le principe des tableaux à deux dimensions, il n’y a aucun problème à passer au maniement de tableaux à trois dimensions, quatre dimensions ou plus. Ainsi, l'accès à un élément dans un tableau à trois dimensions se fait à l'aide de 3 trois indices.

Le principal obstacle au maniement systématique de ces tableaux à plus de trois dimensions est que le programmeur, quand il conçoit son algorithme, a pour habitude de dessiner les structures de données qu'il utilise afin de se les représenter.

Or, il est facile d’imaginer concrètement un tableau à une dimension, cela reste faisable pour deux dimensions, cela devient plus complexe pour les tableaux à trois dimensions, mais les tableaux à plus de trois dimensions sont impossibles à visualiser. En effet, l’esprit humain a des difficultés à se représenter les choses dans l’espace, et est souvent perdu dans l’hyperespace (espace comptant plus de trois dimensions). Donc, pour des raisons uniquement pratiques, les tableaux à plus de trois dimensions sont rarement utilisés.

## Définition

**Rappel :** Un **tableau** est un ensemble de valeurs (de même type) portant le même nom de variable, ces valeurs étant repérées par un nombre, appelé indice.

Un **tableau à double indice** est un **tableau** dont chaque valeur est repérée par **deux indices**.

En réalité, la création d'un tableau à deux dimensions permet l'accès à plus d'une valeur à partir d'un même indice.

En effet, le premier indice permet de repérer ce qu'on appellera une **ligne**, c'est-à-dire un sous-ensemble de cellules contiguës du tableau. Le second indice permet l'accès à une cellule de cette ligne, c'est-à-dire à une **colonne**.

Le nombre de colonnes doit être le même pour chaque ligne et réciproquement. Il s'agira donc bien d'une **matrice**.

Tout comme pour les tableaux à simple indice, l’indice d'un tableau à double indice commence à **0**.

L'accès à une cellule du tableau se fait via ces deux indices :

*NomTab* **[** *indiceLigne* **]** **[** *indiceColonne* **]**.

Un tableau *Mat* de *N* éléments par **ligne** et *M* éléments par **colonne** peut être représenté graphiquement de la manière suivante :

***N* = 5** et ***M* = 10**

Colonnes

0

1

2

3

4

5

6

7

8

9

Mat[4]

Mat[3]

Mat[2]

Mat[1]

Mat[0]

?

?

?

?

?

?

?

?

?

?

?

?

?

?

?

?

?

?

?

?

?

?

?

?

?

?

?

?

?

Lignes

?

?

?

?

?

?

?

?

?

?

?

?

?

?

?

?

?

?

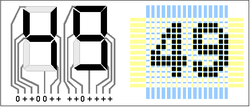
?

?

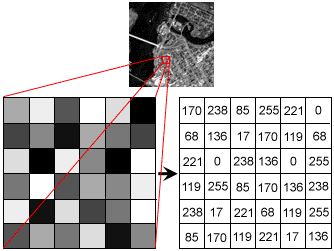
Mat **[**2**][**4**]**

## Exemples d'utilisation

L'utilisation des matrices en informatique a permis notamment le passage de l'affichage par segments à **l'affichage par pixels**[[5]](#footnote-5).



Plus précisément, une **image numérique**[[6]](#footnote-6) est une matrice de pixels dont chacun comporte une information de couleur et de luminance. Ces points sont agencés comme dans un tableau en lignes et en colonnes et sont définis par leur position d'une part, et leur valeur d'autre part. Cette matrice forme une image lorsqu'elle est vue à une distance appropriée.

*Exemple :*

Chaque valeur correspond à une couleur et est traduite pour être visualisée en fonction des codes de couleurs ou de gris. Par exemple pour une gamme allant du noir (0) au blanc (255), le pixel codé 75 sera gris foncé. Mais, si l'on affiche le même pixel, codé 75, sur une gamme de couleurs, celui-ci conservera sa valeur initiale mais deviendra coloré à l'écran, par exemple orange. Ce n'est plus alors le fichier image d'origine qui va déterminer la visualisation de l'image, mais bien la configuration de l'ordinateur et de l'écran utilisés pour traiter l'image. Une même image peut ainsi apparaître totalement différente d'un écran à l'autre.

L'organisation des informations en matrices de points codés va permettre un grand nombre de calculs issus des mathématiques matricielles, non seulement sur les codes mêmes des pixels, mais aussi sur leur spatialisation, c'est-à-dire sur leur position par rapport aux autres points.

Cependant, si la plupart des images nous parviennent codées sur 3 fois 8 bits (un octet), soit 3 fois 256 couleurs potentielles (16 777 216 couleurs possibles), d'autres formats d'images (notamment les formats pour l'internet par exemple) peuvent être codés sur un nombre plus réduit de bits, ce qui réduit le nombre de couleurs.

## Manipulations des tableaux à double indice

### Afficher le contenu d'un tableau à double indice

*Exemple :* On a relevé pour 10 villes différentes, la température moyenne pour chaque mois d’une année.

Comme tous les éléments sont de même type (numérique), un tableau à double indice est tout à fait indiqué pour mémoriser ces informations.

Ce tableau, appelé *TabTemp*, se compose de 10 lignes et de 12 colonnes.

On a donc

TabTemp par ville par mois Température

(10\*) (12\*)

Ce tableau ayant été garni au préalable, on désire écrire le diagramme d’actions qui permet d'afficher son contenu.

* **Observation** 
  + Données : le tableau, le nombre de lignes et le nombre de colonnes ;
  + Entrée : aucune ;
  + Sorties : les valeurs contenues dans le tableau.
* **Diagramme** 
  + Opérations : parcours, sortie ;
  + Variables : *TabTemp*, *CptL*, *CptC* ;
  + Initialisations : *CptL* = 0, *CptC* = 0 ;
  + Diagramme d'actions : SortieTabTemp.

┌─\* SortieTabTemp

│

│ *CptL* = 0

│

│╔═ **do while** **(***CptL* < 10**)**

│║

│║ *CptC* = 0

│║

│║╔═ **do while** **(***CptC* <12**)**

│║║

│║║ **Sortir** TabTemp**[***CptL***][***CptC***]**, "**¬**"

│║║ *CptC*++

│║╙─

│║ **Sortir** "**↵**"

│║ *CptL*++

│╙─

└──

### Afficher le total d'une ligne

*Exemple :* Soit un tableau d'entiers *TabEnt* comportant *NbLig* lignes et *NbCol* colonnes, écrire le diagramme d’actions qui permet d'afficher le total des éléments d'une ligne entrée par l'utilisateur.

* Observation
  + Données : le tableau *TabEnt*, *NbLig* et *NbCol* ;
  + Entrée : le numéro de la ligne ;
  + Sortie : le total des éléments de la ligne.
* Diagramme
  + Opérations : parcours de la ligne, somme, sortie ;
  + Variables : *TabEnt*, *Tot*, *CptC*, *NumLig* ;
  + Initialisations : *Tot* = 0, *CptC* = 0 ;
  + Diagramme d'actions : SortieTotLigne.

┌─\* SortieTotLigne

│

│ **Sortir** "Entre le numéro de la ligne : "

│ **Obtenir** *NumLig*

│

│ *Tot*  = 0

│ *CptC* = 0

│╔═ **do while** **(***CptC* < NbCol**)**

│║

│║ *Tot*  += *TabEnt***[***NumLig***][***CptC***]**

│║ *CptC* ++

│╙─

│ **Sortir** *Tot*

└──

### Afficher le minimum d'une colonne

*Exemple :* Soit un tableau d'entiers *TabEnt* comportant *NbLig* lignes et *NbCol* colonnes, écrire le diagramme d’actions qui permet d'afficher le plus petit élément d'une colonne entrée par l'utilisateur.

* Observation
  + Données : le tableau *TabEnt*, *NbLig* et *NbCol* ;
  + Entrée : le numéro de la colonne ;
  + Sortie : le minimum des éléments de la colonne.
* Diagramme
  + Opérations : parcours de la colonne, minimum, sortie ;
  + Variables : *TabEnt*, *Min*, *CptL*, *NumCol* ;
  + Initialisation : *CptL* = 0, *Min* = 0 ;
  + Diagramme d'actions : SortieMinCol.

┌─\* SortieMinCol

│

│ **Sortir** "Entre le numéro de la colonne : "

│ **Obtenir** *NumCol*

│

│ *Min*  = HV

│ *CptL* = 0

│╔═ **do while** **(***CptL* < NbLig**)**

│║

│║┌─ **if** **(***Min >* *TabEnt***[***CptL***][***NumCol***])**

│║│

│║│ *Min* = *TabEnt***[***CptL***][***NumCol***]**

│║└──

│║

│║ *CptL*++

│╙─

│ **Sortir** *Min*

└──

### Afficher la position d'un élément donné

*Exemple :* Soit un tableau d'entiers *TabEnt* comportant *NbLig* lignes et *NbCol* colonnes, écrire le diagramme d’actions qui permet d'afficher la position (ligne et colonne) d'un élément entré par l'utilisateur. Si ce nombre n'est pas présent dans le tableau, sortir "Élément non trouvé !".

* Observation
  + Données : le tableau *TabEnt*, *NbLig* et *NbCol* ;
  + Entrée : le nombre recherché ;
  + Sortie : la position du nombre ou un message.
* Diagramme
  + Opérations : recherche, sortie ;
  + Variables : TabEnt, CptL, CptC, NbrLu, Trouve ;
  + Initialisation : *CptL* = 0, *CptC* = 0, *Trouve* = FAUX ;
  + Diagramme d'actions : SortiePosition.

┌─\* SortiePosition

│

│ **Sortir** "Entre un nombre : "

│ **Obtenir** *NbrLu*

│

│ *Trouve* = FAUX

│ *CptL* = 0

│╔═ **do while** **(***CptL* < NbLig **AND** *Trouve* = FAUX**)**

│║

│║ *CptC* = 0

│║

│║╔═ **do while** **(***CptC* <NbCol **AND** TabEnt **[***CptL***][***CptC***]** ≠ *NbrLu***)**

│║║

│║║*CptC*++

│║╙─

│║┌─ **if** **(***CptC* < NbCol**)**

│║│

│║│ *Trouve* = VRAI

│║├─ **else**

│║│

│║│ *CptL*++

│║└──

│╙─

│┌─ **if** **(***Trouve* = VRAI**)**

││

││**Sortir** "[", *CptL*, " ; ", *CptC*, "]"

│├─ **else**

││

││**Sortir** "Elément non trouvé !"

│└──

└──

## Tableau de structure ou Tableau à double indice ?

Dans certains cas, deux solutions s'offrent au programmeur pour stocker les informations à traiter en mémoire centrale.

En effet, prenons l'exemple suivant :

On désire calculer la moyenne, pour chaque cours, des cotes de janvier des étudiants de premières TI. Pour ce faire, il est nécessaire de stocker, pour chacun d'eux, les différentes cotes sur 20 qu'il a obtenues.

Les deux solutions possibles pour stocker ces informations sont :

* + Un tableau de structure comprenant l'identifiant (nom et prénom) de l'étudiant et un tableau de cotes ;
  + Un tableau à double indice contenant les cotes des étudiants, chaque ligne correspondant à un étudiant.

### Tableau de structure

La première manière de procéder est de mémoriser non seulement les cotes des étudiants mais aussi leur identifiant.

#### Structure

TabEtud

CellEtud

NbEtud \*

NomPrenom

TabCote

CellCote

NbCours \*

Cote

#### Représentation graphique

On représenterait les informations de la manière suivante :

Balasko Josiane

10

12

14

15

11

10

12

11

15

10

12

Blanc Michel

6

14

10

12

12

14

9

7

12

14

16

Clavier Christian

12

10

13

11

10

8

18

17

14

13

15

Chazel Marie-Anne

4

13

8

17

9

11

14

15

10

9

10

Jugnot Gérard

8

15

9

13

7

13

13

12

10

11

9

Lhermitte Thierry

5

17

12

14

8

10

11

10

13

12

10

**…**

TabEtud[0]

TabEtud[1]

TabEtud[2]

TabEtud[3]

TabEtud[4]

TabEtud[5]

TabEtud[0].TabCote[1]

#### Algorithmes

* Observation
  + Données : le tableau *TabEtud*, *NbEtud*, *NbCours* ;
  + Entrée : aucune ;
  + Sortie : aucune.
* Diagramme
  + Opérations : calcul des moyennes ;
  + Variables : TabEtud, CptEt, CptCrs, TabMoy ;
  + Initialisation : Initialisation : *CptEt* = 0, *CptCrs* = 0, *TabMoy* ;
  + Diagramme d'actions : MoyenneCours.

┌─\* MoyenneCours

│

│ *CptCrs* = 0

│╔═ **do while** **(***CptCrs* <NbCours**)**

│║ *TabMoy***[***CptCrs***]** = 0

│║ *CptCrs* ++

│╙─

│ *CptEt* = 0

│╔═ **do while** **(***CptEt* < NbEtud**)**

│║

│║ *CptCrs* = 0

│║

│║╔═ **do while** **(***CptCrs* <NbCours**)**

│║║ *TabMoy***[***CptCrs***]** += *TabEtud***[***CptEt***].***TabCote***[***CptCrs***]**

│║║ *CptCrs* ++

│║╙─

│║ *CptEt* ++

│╙─

│ *CptCrs* = 0

│╔═ **do while** **(***CptCrs* <NbCours**)**

│║ *TabMoy***[***CptCrs***]** /= NbEtud

│║ *CptCrs* ++

│╙─

└──

On remarque rapidement que, dans cet algorithme, l'identifiant de l'étudiant n'est pas utilisé. D'où, en utilisant une structure, on mémorise une information inutile qui est cet identifiant. En effet, dans le cas de notre traitement, cet identifiant ne sert à rien puisqu'il s'agit de calculer la moyenne par cours et non par étudiant !

### Tableau à double indice

Cette deuxième manière de procéder consiste à mémoriser uniquement les cotes des étudiants en associant à chacun une ligne du tableau, chaque cellule de cette ligne représentant une des cotes de celui-ci.

#### Structure

TabCote

Ligne

NbEtud \*

Cote

Colonne

NbCours \*

#### Représentation graphique

On représenterait les informations de la manière suivante :

TabCote**[**0**][**1**]**

TabCote[0]

10

12

14

15

11

10

12

11

15

10

12

12 14

16

14

12

7

9

12

10

14

6

TabCote[1]

TabCote[2]

4

13

8

17

9

11

14

15

10

9

10

TabCote[3]

12

10

13

11

10

8

18

17

14

13

15

TabCote[4]

8

15

9

13

7

13

13

12

10

11

9

TabCote[5]

5

17

12

14

8

10

11

10

13

12

10

#### Algorithmes

* Observation
  + Données : le tableau *TabCote*, *NbEtud*, *NbCours* ;
  + Entrée : aucune ;
  + Sortie : aucune.
* Diagramme
  + Opérations : calcul des moyennes ;
  + Variables : TabCote, CptEt, CptCrs, TabMoy;
  + Initialisation : *CptEt* = 0, *CptCrs* = 0, *TabMoy* ;
  + Diagramme d'actions : SortiePosition.

┌─\* SortiePosition

│

│ *CptCrs* = 0

│╔═ **do while** **(***CptCrs* <NbCours**)**

│║ *TabMoy***[***CptCrs***]** = 0

│║ *CptCrs* ++

│╙─

│ *CptEt* = 0

│╔═ **do while** **(***CptEt* < NbEtud**)**

│║

│║ *CptCrs* = 0

│║╔═ **do while** **(***CptCrs* <NbCours**)**

│║║ *TabMoy***[***CptCrs***]** += *TabCote***[***CptEt***][***CptCrs***]**

│║║ *CptCrs* ++

│║╙─

│║ *CptEt* ++

│╙─

│ *CptCrs* = 0

│╔═ **do while** **(***CptCrs* <NbCours**)**

│║ *TabMoy***[***CptCrs***]** /= NbEtud

│║ *CptCrs* ++

│╙─

└──

Cette solution est donc plus appropriée si l'algorithme se limite à ce genre de calcul sans devoir tenir compte de l'identifiant de l'étudiant. En effet, il permet de ne pas utiliser de place mémoire inutile. De plus, la notation sera plus simple puisqu'il n'y aura pas besoin de spécifier le champ TabCote, celui-ci étant remplacé par une ligne du tableau à double indice.

### Conclusion

En fonction des informations nécessaires au traitement, le programmeur doit choisir la représentation qui correspond le mieux.

Le but de la « modélisation informatique » n’est pas de compliquer la démarche mais de l’adapter au mieux aux besoins précis de l’utilisateur.

On veillera ainsi à ne stocker que les données utiles à la gestion du problème en évitant ainsi de faire des doubles emplois. Ceci d'ailleurs est l’un des objectifs principaux de l’élaboration de base de données.

# Les listes chaînées

## Pourquoi des listes chaînées ?[[7]](#footnote-7)

En organisant d'une certaine manière les données, on permet un traitement automatique de ces dernières plus efficace et rapide. Le fait d'utiliser une organisation de données appropriée à un traitement peut également faire baisser de manière significative la complexité d'une application et ainsi participer à faire baisser le taux d'erreurs.

Dans le cadre de ce cours, seules les organisations de données **séquentielles** sont abordées. Elles permettent de ranger des objets dans un ordre arbitraire. Parmi celles-ci, on retrouve les **tableaux**, les **piles** et les **files**, ainsi que les **listes**.

Les tableaux ont la particularité d'être indicés. On peut ainsi accéder à chaque élément du tableau directement par son indice. Cependant, lorsqu'on veut ajouter ou supprimer un élément, il est nécessaire d'avoir recours à des décalages afin de limiter la perte de données ou la perte de place en mémoire. De plus, lors de l'allocation – la réservation – de la mémoire, un tableau se voit allouer un espace correspondant au nombre maximum d'éléments qu'il peut contenir. Cet espace lui est alloué une fois pour toutes, c'est-à-dire de manière statique. D'où, si le traitement nécessite une place mémoire plus importante, il est nécessaire d'allouer un nouvel espace, plus grand, et d'y recopier, une à une, les informations présentes dans le tableau déjà existant.

Le choix d'une implémentation particulière dépend d'un certain nombre de compromis, comme l'occupation mémoire ou les performances requises pour diverses opérations de base : itération, ajout d'un élément (au début, à la fin ou encore dans un emplacement quelconque de la collection), recherche d'un élément, suppression d'un élément, décompte du nombre d'éléments, etc.

Il semble alors nécessaire de trouver une autre technique permettant de mieux s'adapter à certains types de traitement. Cette nouvelle organisation des données est appelée **liste chainée**.

Contrairement à un tableau, la taille d'une liste chaînée n'a comme limite que celle de la mémoire disponible. En effet, la mémoire est allouée de manière dynamique. L'organisation des données est telle que chaque élément peut pointer, suivant le type de liste chaînée, vers un ou plusieurs éléments de la liste. Ainsi, pour augmenter la taille d'une liste chaînée, il suffit de créer un nouvel élément et de faire pointer certains éléments déjà présents vers ce nouvel élément.

Il existe plusieurs types de listes chaînées :

* Les **listes simplement chaînées** : chaque élément dispose d'un pointeur sur l'élément suivant de la liste. Le parcours se fait dans un seul sens ;
* Les **listes doublement chaînées** : chaque élément dispose de deux pointeurs, un sur l'élément suivant et un autre sur le précédent. Le parcours peut se faire alors dans les deux sens.
* Les **listes chainées cycliques** : il s’agit d’une liste chaînée simple ou double qui forme une boucle. Dès que l'on a atteint la "fin" de la liste et que l'on désire continuer, on se retrouve sur le "premier" élément de la liste.
* Etc.

En résumé, voici un tableau comparant point par point les caractéristiques des tableaux à celles des listes chainées :

|  |  |  |
| --- | --- | --- |
| **Propriétés** | **Liste Chainée** | **Tableau** |
| Allocation de la mémoire | Allocation **dynamique**.  Permet d’allouer la mémoire seulement lors de l’exécution du programme et non lors de la compilation.  Utile lorsqu'on ne connait pas le nombre d'éléments à traiter. | Allocation **statique**.  Permet d'allouer la mémoire lors de la compilation sans pouvoir modifier sa taille par la suite.  Utile lorsqu’on connaît d’avance le nombre d'éléments. |
| Espace mémoire nécessaire | Chaque élément comprenant les données ainsi que un ou plusieurs pointeurs, la **taille** de celui-ci est **plus élevée**.  Lorsque le nombre d'éléments est élevé, l'espace utilisé peut devenir beaucoup plus grand que celui nécessaire à un tableau. | Chaque élément ne contient que les informations à mémoriser. Un élément prend alors **moins** **de place** en mémoire.  Lorsque le nombre d'éléments est élevé, la place mémoire se limite au strict nécessaire. |
| Accès à un élément | Comme les éléments ne sont pas indicés, l'accès à un élément se fait à l'aide d'un **parcours de la chaîne**.  Même si l’on connaît l'emplacement de l'élément dans la chaîne, il est nécessaire de la parcourir pour l'atteindre. | Comme les éléments sont indicés, un **accès direct** à l'information est possible.  Lorsque l’on connait l'emplacement de l’élément recherché, l'accès à celui-ci est plus rapide que dans une liste chaînée. |
| Manipulations | L'ajout et la suppression d'un élément se font grâce à la **création ou la suppression d'un chaînon**.  L'**insertion** ou la **suppression** d'un ou plusieurs éléments est **plus rapide** que dans un tableau. | L'ajout ou la suppression d'un élément nécessite soit un **décalage** à droite, soit un décalage à gauche des autres éléments du tableau.  L'**insertion** ou la **suppression** d'un ou plusieurs éléments est **plus lourde** que dans une liste chaînée. |

## Définition

Une liste est un **ensemble ordonné d’éléments**. Chaque élément peut comporter un certain nombre de champs ou être lui-même une liste.

Les éléments de la liste, appelés **chaînons**, comportent des champs de données et un champ appelé **pointeur** contenant l’adresse de l’élément suivant.

Le début de la liste est déterminé par un **pointeur de début de liste** retenant l’adresse du premier élément de la liste.

*Exemple :*

PtrDeb

Données

Pointeur

Données

Pointeur

Données

Pointeur

Chaînon

## Notations

Par convention, on utilise, comme pointeurs, les noms de variable suivants :

***PtrDeb***  variable contenant l’adresse du premier élément de la liste ;

***Ptr***  variable contenant l’adresse du chaînon courant ;

***Ptr🡪Données***  permet d’accéder au contenu des données.

***Ptr🡪PtrSuiv***  variable contenant l’adresse du chaînon suivant le chaînon courant ;

***Ptr🡪PtrPrec*** variable contenant l’adresse du chaînon précédent le chaînon courant (uniquement dans les listes doublement chaînée) ;

***PtrNouv***  variable contenant l’adresse d'un nouveau chaînon ;

## Fonctionnement

On accède aux éléments de la liste en passant d'un chaînon à l'autre via le pointeur vers le chaînon suivant le chaînon courant – ***Ptr🡪PtrSuiv***, et ce à partir du pointeur indiquant le début de la liste – ***PtrDeb***.

Les éléments de la liste ne sont pas nécessairement placés consécutivement en mémoire. Ils **peuvent être disséminés** un peu partout. C'est pourquoi un langage de programmation doit disposer d’un mécanisme permettant d’**allouer** un espace mémoire **de manière dynamique** pour l’ajout de nouveaux éléments. Il doit également permettre de **libérer** cet espace lors de la suppression d'un élément.

## Opérations sur les listes

Les opérations pouvant être réalisées sur les listes chaînées sont les suivantes :

* Placement sur le premier élément de la liste ;
* Placement sur le dernier élément de la liste (si liste doublement chaînée) ;
* Placement sur l'élément suivant l'élément courant si c'est possible ;
* Placement sur l'élément précédent l'élément courant si c'est possible (si liste doublement chaînée) ;
* **Vérifier** si la liste est vide ou non ;
* **Recherche** d'un élément de la liste ;
* **Mise-à-jour** d'un élément de la liste ;
* **Ajout** d'un élément dans la liste ;
* **Suppression** d'un élément de la liste ;
* **Comptage** du nombre d'éléments de la liste.

## Exemples d'utilisation

### Gestion de la mémoire par listes chainées[[8]](#footnote-8)

Une méthode pour mémoriser l'occupation de la mémoire consiste à gérer une liste chaînée des segments libres et occupés, un segment étant un processus ou un espace libre entre deux processus.

Chaque entrée spécifie, pour une zone libre ou pour un processus, son adresse de départ, sa longueur et un pointeur sur l'entrée suivante.

Cette liste peut être triée sur les adresses. Ce tri permet de mettre la liste à jour facilement lorsqu'un processus se termine ou est déplacé sur le disque.

### Gestion de l'accès à un fichier séquentiel[[9]](#footnote-9)

Un fichier séquentiel consiste en blocs d’octets enregistrés sur un support tel que ruban, disque… La dimension de ces blocs est dictée par les caractéristiques du support. Ces blocs sont lus (lecture physique) dans un tampon en mémoire.

Un bloc contient un certain nombre d’enregistrements qui sont des unités d’information logiques pour l’application (un étudiant, un client, un produit, etc.) souvent de longueur et de contenu uniformes. Ils sont triés selon une clé, normalement un code (code d’étudiant, numéro produit, etc.).

Les opérations sur les fichiers séquentiels se résument, essentiellement, en la lecture de l'enregistrement suivant ou l'écriture d'un nouvel enregistrement en fin du fichier séquentiel. Un fichier séquentiel peut être accessible soit en lecture, soit en écriture.

Le fichier séquentiel en lecture est initialement positionné sur le premier enregistrement logique. Chaque opération de lecture transfère dans une zone interne au programme un enregistrement du fichier, et prépare le fichier pour le positionner sur l'enregistrement suivant. On dispose souvent d'une opération complémentaire qui permet de savoir s'il y a encore des enregistrements à lire dans le fichier, et éventuellement une opération permettant le retour au début du fichier.

Le fichier séquentiel en écriture peut être initialement vide, ou positionné après le dernier enregistrement déjà dans le fichier. Chaque opération d'écriture rajoute un enregistrement dans le fichier depuis une zone interne au programme, et positionne le fichier après cet enregistrement. L'écriture d'un enregistrement se fait donc toujours après ceux qui sont déjà dans le fichier.

Il est à noter que la caractéristique essentielle d'un fichier séquentiel est que les enregistrements sont lus dans l'ordre où ils ont été écrits.

### Piles et files

Les piles et les files peuvent être implémentées sous la forme de listes. Leurs propriétés restent les mêmes, mise à part l'allocation de la mémoire qui devient dynamique.

## Exercice introductif sur les listes chaînées

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 681 |  | NumAbon | NomAbon | PtrSuiv |
|  |  |  |  | 217 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5268 |  | NumAbon | NomAbon | PtrSuiv |
|  |  |  |  | 729 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 4023 |  | NumAbon | NomAbon | PtrSuiv |
|  |  |  |  | 681 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2918 |  | NumAbon | NomAbon | PtrSuiv |
|  |  |  |  | 3400 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 4321 |  | NumAbon | NomAbon | PtrSuiv |
|  |  |  |  | 5268 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 729 |  | NumAbon | NomAbon | PtrSuiv |
|  |  |  |  | 4023 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 217 |  | NumAbon | NomAbon | PtrSuiv |
|  |  |  |  | 63 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 3400 |  | NumAbon | NomAbon | PtrSuiv |
|  |  |  |  | NULL |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 63 |  | NumAbon | NomAbon | PtrSuiv |
|  |  |  |  | 2918 |

1. A partir de l’adresse de PtrDeb = 4321 et des PtrSuiv, effectuer correctement les chaînages.
2. Sachant que la liste est triée par ordre alphabétique sur le NomAbon, compléter les chaînons avec les noms d’abonnés suivants : Bertrand, Collignon, Dubois, Evrard, Forain, Gérard, Mortiaux, Pierard, Walraf.

Écrire la 1ière instruction de remplissage.

1. Écrire le DA qui permet de remplir les numéros des abonnés (NumAbon) sachant qu’elles correspondent à l’ordre alphabétique (numéros commençant à partir de 1200).
2. Si Ptr = 4023, donner la valeur de

Ptr🡪PtrSuiv :

Ptr🡪NomAbon :

Ptr🡪RefAbon :

1. Supprimer l’abonné ayant la référence 1205 et recréer les liens.
2. Insérer un nouvel abonné au bon endroit (par *Exemple :* Moulin).

## Manipulations des listes chaînées

Ci-après se trouvent les différents diagrammes d'actions qui permettent de réaliser les principales opérations sur les listes :

* **Recherche** d'un élément de la liste ;
* **Mise-à-jour** d'un élément de la liste ;
* **Ajout** d'un élément dans la liste ;
* **Suppression** d'un élément de la liste ;
* **Comptage** du nombre d'éléments de la liste.

Tous ces diagrammes sont réalisés sur base d'un énoncé commun décrit au point suivant.

### Énoncé commun

On désire permettre la gestion des inscriptions à une *newsletter*.

Lors de l'inscription, chaque abonné doit encoder les informations qui le concernent, c’est-à-dire :

* son numéro d'abonné ;
* son nom ;
* sa nationalité.

Chaque chaînon de la liste représente donc un abonné.

La **structure** d'un chainon est : La **représentation graphique** d'un chaînon est :

NumAbon

NomAbon

NatAbon

***PtrSuiv***

Skieur

NumAbon

NomAbon

NatAbon

***PtrSuiv***

### Recherche d’un élément de la liste

De même que pour les tableaux, on peut travailler avec une liste qui est soit non triée, soit triée.

#### Liste non triée

*Exemple :* Soit une liste chaînée non triée de abonnés, dont le pointeur de début est *PtrDeb*, écrire le diagramme d’actions qui permet de rechercher un abonné dont le numéro d’abonné, *NumAbonLu*, a été entré par l'utilisateur au clavier.

* Observation
  + Donnée : le pointeur de début de la liste ;
  + Entrée : le numéro de l’abonné recherché ;
  + Sortie : aucune.
* Diagramme
  + Opération : recherche dans de la liste ;
  + Variables : *PtrDeb*, *NumAbonLu*, *Ptr* ;
  + Initialisation : *Ptr* = *PtrDeb* ;
  + Diagramme d'actions : Recherche.

┌─\* Recherche

│

│ **Sortir** " Entrer le numéro d'abonné recherché : "

│ **Obtenir** *NumAbonLu*

│

│ *Ptr* = *PtrDeb*

│╔═ **do while** ( (*Ptr* ≠ NULL) **AND** (*NumAbonLu* ≠ *Ptr*🡪*NumAbon*) )

│║

│║ *Ptr* = *Ptr*🡪*PtrSuiv*

│╙─

└──

Si l'abooné ne fait pas partie de la liste, le pointeur *Ptr* aura la valeur NULL.

#### Liste triée

*Exemple :* Soit une liste chaînée d’abonnés, triée en ordre croissant sur leur numéro d’abonné, dont le pointeur de début est *PtrDeb*, écrire le diagramme d’actions qui permet de rechercher un abonné dont le numéro, *NumAbonLu*, a été entré par l'utilisateur au clavier. Si l'abonné ne fait pas partie de la liste, afficher le message " L'abonné recherché ne fait pas partie de la liste ! ".

* Observation
  + Donnée : le pointeur de début de la liste ;
  + Entrée : le numéro de l’abonné recherché ;
  + Sortie : aucune.
* Diagramme
  + Opération : recherche dans de la liste triée;
  + Variables : *PtrDeb*, *NumAbonLu*, *Ptr* ;
  + Initialisation : *Ptr* = *PtrDeb* ;
  + Diagramme d'actions : Recherche.

┌─\* Recherche

│

│ **Sortir** " Entrer le numéro d'abonné recherché : "

│ **Obtenir** *NumAbonLu*

│ *Ptr* = *PtrDebSki*

│

│╔═ **do while** **(** (*Ptr* ≠ NULL) **AND** (*NumAbonLu* > *Ptr🡪*NumAbon) **)**

│║

│║ *Ptr* = *Ptr🡪PtrSuiv*

│╙─

│┌─ **if** **(** (*Ptr* ≠ NULL) **AND** (*NumAbonLu <* *Ptr🡪*NumAbon) **)**

││

││ **Sortir** " L'abonné recherché ne fait pas partie de la liste ! "

│└──

└──

### Mise-à-jour d'un élément de la liste

*Exemple :* Soit une liste chaînée d'abonnéq, triée en ordre croissant sur leur numéro d’abonné, dont le pointeur de début est *PtrDeb*, écrire le diagramme d’actions qui permet de mettre à jour les informations concernant un abonné dont le numéro d'abbonné, *NumAbonLu*, a été entré par l'utilisateur au clavier. Le message "Modification effectuée !" est affiché si la mise-à-jour est concluante et "Échec !" si non.

* Observation
  + Donnée : le pointeur de début de la liste ;
  + Entrée : le numéro de l’abonné recherché ;
  + Sortie : aucune.
* Diagramme
  + Opérations : recherche, obtention, mise-à-jour ;
  + Variables : PtrDeb, NumAbonLu, Message, Ptr ;
  + Initialisation : *Ptr* = *PtrDeb* ;
  + Diagramme d'actions : Mise-a-jour.

┌─\* Mise-a-jour

│

│ **Sortir** " Entrer le numéro d'abonné recherché : "

│ **Obtenir** *NumAbonLu*

│

│ *Ptr* = *PtrDebSki*

│╔═ **do while** **(** (*Ptr* ≠ NULL) **AND** (*NumAbonLu* > *Ptr🡪*NumAbon) **)**

│║

│║ *Ptr* = *Ptr🡪PtrSuiv*

│╙─

│┌─ **if** **(** (*Ptr* ≠ NULL) **AND** (*NumAbonLu <* *Ptr🡪*NumAbon) **)**

││

││ *Message* = " Echec !"

│├─ **else**

││

││ **Sortir** " Entrer le nom de l'abonné : "

││ **Obtenir** *Ptr🡪NomAbon*

││

││ **Sortir** " Entrer la nationnalit de l'abonné : "

││ **Obtenir** *Ptr🡪NatAbon*

││

││ *Message* = " Modification effectuée !"

│└──

└──

### Ajout d’un élément dans une liste

*Exemple :* Soit une liste chaînée d'abonnés, dont le pointeur de début est PtrDeb, écrire le diagramme d’actions qui permet d'ajouter un abonné à cette liste.

L'ajout dans une liste chaînée peut s'effectuer de trois manières :

* On insère le nouveau chaînon en début de liste, avant le premier chaînon ;
* On insère le nouveau chaînon en fin de liste, après le dernier chaînon ;
* On insère le nouveau chaînon dans la liste, avant le dernier chaînon courant.

#### Allocation de mémoire

Dans tous les cas, avant d'ajouter un nouvel élément, il faut créer un nouveau chaînon c'est-à-dire qu'il faut lui allouer la place mémoire permettant de conserver les informations prévues.

* Observation
  + Données : aucune;
  + Entrée : aucune ;
  + Sortie : message permettant de savoir si l'allocation s'est bien déroulée.
* Diagramme
  + Opération : allocation de mémoire ;
  + Variable : *PtrNouv* ;
  + Initialisation : aucune ;
  + Diagramme d'actions : Allocation.

┌─\* Allocation

│

│ *PtrNouv* = adresse d’un nouveau chaînon

│ (allocation de l'emplacement par le système)

│

│┌─ **if** **(***PtrNouv* = NULL**)**

││

││**Sortir** " Mémoire insuffisante !"

│├─ **else**

││

││**Sortir** "Allocation de mémoire réussie !"

│└──

└──

#### Ajout en début de liste

* Observation
  + Donnée : le pointeur de début de la liste ;
  + Entrées : les informations concernant le nouvel abonné ;
  + Sortie : aucune.
* Diagramme
  + Opérations : Allocation, obtention et chaînage;
  + Variables : *PtrDeb*, *PtrNouv* ;
  + Initialisation : aucune ;
  + Diagramme d'actions : AjoutDébut.

┌─\* AjoutDébut

│

│ *PtrNouv* = adresse d’un nouveau chaînon

│ (allocation de l'emplacement par le système)

│

│┌─ **if** **(***PtrNouv* = NULL**)**

││

││**Sortir** " Mémoire insuffisante !"

│├─ **else**

││

││ **Sortir** " Entrer le numéro de l'abonné : "

││ **Obtenir** *PtrNouv 🡪NumAbon*

││

││ **Sortir** " Entrer le nom de l'abonné : "

││ **Obtenir** *PtrNouv 🡪NomAbon*

││

││ **Sortir** " Entrer la nationnalit de l'abonné : "

││ **Obtenir** *PtrNouv 🡪NatAbon*

││

││ *PtrNouv🡪PtrSuiv* = *PtrDeb*

││ *PtrDeb* = *PtrNouv*

│└──

└──

#### Ajout en fin de liste

* Observation
  + Donnée : le pointeur de début de la liste ;
  + Entrées : les informations concernant le nouvel abonné ;
  + Sortie : aucune.
* Diagramme
  + Opérations : Allocation, obtention et chaînage;
  + Variables : PtrDeb, PtrNouv, Ptr, PtrSauv ;
  + Initialisation : aucune ;
  + Diagramme d'actions : AjoutFin.

┌─\* AjoutFin

│

│

│ *PtrNouv* = adresse d’un nouveau chaînon

│ (allocation de l'emplacement par le système)

│

│┌─ **if** **(***PtrNouv* = NULL**)**

││

││**Sortir** " Mémoire insuffisante !"

│├─ **else**

││

││ **Sortir** " Entrer le numéro de l'abonné : "

││ **Obtenir** *PtrNouv 🡪NumAbon*

││

││ **Sortir** " Entrer le nom de l'abonné : "

││ **Obtenir** *PtrNouv 🡪NomAbon*

││

││ **Sortir** " Entrer la nationnalit de l'abonné : "

││ **Obtenir** *PtrNouv 🡪NatAbon*

││

││ *PtrNouv🡪PtrSuiv* = NULL

││

││┌─ **if** **(***PtrDeb* = NULL**)**

│││

│││ *PtrDeb* = *PtrNouv*

│││

││├─ **else**

│││

│││ *Ptr*  = *PtrDeb*

│││╔═ **do while** **(***Ptr* ≠ NULL**)**

│││║

│││║ *PtrSauv* = *Ptr*

│││║ *Ptr* = *Ptr🡪PtrSuiv*

│││╙─

│││ *PtrSauv🡪PtrSuiv* = *PtrNouv*

││└──

│└──

└──

#### Ajout dans la liste

┌─\* AjoutDans

│

│

│ *PtrNouv* = adresse d’un nouveau chaînon

│ (allocation de l'emplacement par le système)

│

│┌─ **if** **(***PtrNouv* = NULL**)**

││

││**Sortir** " Mémoire insuffisante !"

│├─ **else**

││

││ **Sortir** " Entrer le numéro de l'abonné : "

││ **Obtenir** *PtrNouv 🡪NumAbon*

││

││ **Sortir** " Entrer le nom de l'abonné : "

││ **Obtenir** *PtrNouv 🡪NomAbon*

││

││ **Sortir** " Entrer la nationnalit de l'abonné : "

││ **Obtenir** *PtrNouv 🡪NatAbon*

││

││

││ *Ptr* = *PtrDeb*

││╔═ **do while** **(***Ptr* ≠ NULL **AND** *PtrNouv🡪NumAbon*> *Ptr🡪NumAbon***)**

││║

││║ *PtrSauv* = *Ptr*

││║ *Ptr* = *Ptr🡪PtrSuiv*

││╙─

││

││

││

││

││

On utilise ce type d'insertion lorsqu'il s'agit d'une liste triée.

* Observation
  + Donnée : le pointeur de début de la liste ;
  + Entrées : les informations concernant le nouvel abonné ;
  + Sortie : aucune.
* Diagramme
  + Opérations : Allocation, obtention et chaînage;
  + Variables : PtrDeb, PtrNouv, Ptr, PtrSauv ;
  + Initialisation : aucune ;
  + Diagramme d'actions : AjoutDans.

**OU**

││

││

││┌─ if (PtrDeb = NULL)

│││

│││ PtrNouv🡪PtrSuiv = NULL

│││ PtrDeb = PtrNouv

│││

││├─ else

│││

│││┌─ if (Ptr = PtrDeb )

││││

││││ PtrNouv🡪PtrSuiv = PtrDeb

││││ PtrDeb = PtrNouv

││││

│││├─ else

││││

││││┌─ if (Ptr = NULL)

│││││

│││││ PtrSauv🡪PtrSuiv = PtrNouv

│││││ PtrNouv🡪PtrSuiv = NULL

│││││

││││├─ else

│││││

│││││ PtrSauv🡪PtrSuiv = PtrNouv

│││││ PtrNouv🡪PtrSuiv = Ptr

││││└──

│││└──

││└──

│└──

└──

### Suppression d’un élément de la liste

*Exemple :* Soit une liste chaînée d'abonnés non triée dont le pointeur de début est *PtrDeb*, écrire le diagramme d’actions qui permet de supprimer un abonné dont le numéro, *NumAbonLu*, a été entré par l'utilisateur au clavier.

* Observation
  + Donnée : le pointeur de début de la liste ;
  + Entrée : le numéro de l’abonné recherché ;
  + Sortie : aucune.
* Diagramme
* Opérations : Recherche, chaînage, libération;
* Variables : *NumAbonLu*, *PtrDeb*, *Ptr*, *PtrSauv* ;
* Initialisation : aucune ;
* Diagramme d'actions : Suppression.

┌─\* Suppression

│

│┌─ **if** **(***PtrDeb* = NULL**)**

││

││ **Sortir** "Liste vide !"

│├─ **else**

││

││ *Ptr*  = *PtrDeb*

││╔═ **do while** **(***Ptr* ≠ NULL **AND** *Ptr🡪NumAbon* ≠ *NumAbonLu***)**

││║

││║ *PtrSauv* = *Ptr*

││║ *Ptr* = *Ptr🡪PtrSuiv*

││╙─

││┌─ **if (***Ptr*= NULL**)**

│││

│││ **Sortir** "Elément absent !"

││├─ **else**

│││

│││┌─ **if** **(***Ptr* = *PtrDeb***)**

││││

││││ *PtrDeb = PtrDeb🡪PtrSuiv*

│││├─ **else**

││││

││││ *PtrSauv🡪PtrSuiv* = *Ptr🡪PtrSuiv*

│││└──

││└──

││ **Libérer** *Ptr*

│└──

└──

### Comptage du nombre d’éléments de la liste

*Exemple :* Soit une liste chaînée d'abonnés, triée ou non, dont le pointeur de début est *PtrDeb*, écrire le diagramme d’actions qui permet de compter le nombre d'abonnés de cette liste.

* Observation
  + Données : le pointeur de début de la liste ;
  + Entrée : aucune ;
  + Sortie : aucune.
* Diagramme
  + Opérations : Comptage ;
  + Variables : *PtrDeb*, *Ptr*, *Cpt* ;
  + Initialisation : aucune ;
  + Diagramme d'actions : Comptage.

┌─\* Comptage

│

│ *Ptr*  = *PtrDeb*

│ *Cpt*  = 0

│╔═ **do while** **(***Ptr* ≠ NULL**)**

│║

│║ *Cpt++*

│║ *Ptr* = *Ptr🡪PtrSuiv*

│╙─

└──

# Les Arbres

## Pourquoi des arbres ?

Une fois de plus, il s'agit de présenter une nouvelle manière d'organiser les données, afin de permettre un traitement automatique de ces dernières plus efficace et rapide. Le fait d'utiliser une organisation de données appropriée à un traitement peut comme il a déjà été signalé au chapitre précédent diminuer de manière significative la complexité d'une application.

Dans le cadre de ce cours, les organisations de données **séquentielles** ont déjà été abordées. Elles permettent de ranger des objets dans un ordre arbitraire. Parmi celles-ci, on retrouve les **tableaux**, les **piles** et les **files**, ainsi que les **listes chaînées**.

Une nouvelle technique consiste en l'organisation des données en mémoire de manière logique et hiérarchisée. Cela permet une consultation des informations plus rapide et une manipulation des données plus facile. Cette technique ne fait plus partie des organisations de données séquentielles, mais est considérée comme une organisation de données **récursive**.

## Quelques notions de la théorie des graphes

### Définition

**Graphe** Un graphe est un ensemble fini ou dénombrable de nœuds reliés par des arcs (graphe orienté) ou des arêtes (graphe non orienté).

Graphiquement, les sommets sont représentés par des points et les arêtes par des lignes reliant ces points.

**Graphe orienté** Un graphe orienté est un graphe dans lequel une direction est associée à chaque arc. Ceci permet de distinguer les deux extrémités de l'arc, l'une étant l'extrémité initiale et l'autre l'extrémité terminale.

Graphiquement, la direction de l'arc est représentée par une flèche orientée vers l'extrémité terminale.

**Graphe**

x3

x7

x1

x5

x2

x6

x4

u1

u5

u2

u6

u3

u7

u4

u8

u9

**Graphe orienté**

x1

x5

x2

x6

x3

x7

x4

u1

u5

u2

u6

u3

u7

u4

u8

u9

*Exemples :*

**Chemin** **Suite d'arcs** telle que l'extrémité terminale d'un arc est l'origine du suivant.

*Exemple :* Les arcs u2, u5, u7 et u4 forment un chemin.

**Arbre** Un arbre est un **graphe** tel que, pour aller d'un nœud à l'autre, on ne peut passer que par un seul chemin.

C'est donc un ensemble **fini d’éléments appelés nœuds**, organisé de manière hiérarchique. Deux nœuds sont **reliés par un arc**.

*Exemple :*

x1

x2

x5

x7

x4

x10

x9

x8

x6

x11

x3

**Nœud** Un nœud peut être père ou fils :

* + - un nœud fils ou **descendant** est un nœud situé immédiatement après un nœud ;
    - un nœud père ou **ascendant** est un nœud situé immédiatement avant un nœud.

**Arc** Un arc relie deux nœuds.

**Racine** La racine est le nœud ascendant de tous les nœuds de l'arbre.

*Exemple :* x1 est la racine du graphe.

**Sous-arbre** Les nœuds autres que la racine sont partitionnés en ensembles disjoints qui sont eux-mêmes des arbres. On les appelle sous-arbres de la racine.

**Feuille** Nœud qui n'est l'origine d'aucun arc. Il s'agit donc d'un nœud sans fils.

*Exemple :* x4, x8, x9, x10 et x11 sont les feuilles de l'arbre.

**Branche** Tout chemin allant de la racine à une feuille.

**Écart** Longueur du plus court chemin entre deux nœuds.

*Exemple :* L'écart entre x1 et x4 est de 2.

### Caractéristiques d'un arbre

Un arbre est notamment caractérisé par les trois éléments suivants :

* sa **taille**  le nombre de ses nœuds ;
* sa **hauteur**  la longueur de sa branche la plus longue ;
* son **degré**  le degré maximum de ses nœuds = nombre de fils possibles. *Exemple :* arbre binaire = 2, tertiaire = 3, n-aire = n.

## Exercice introductif sur les arbres

Racine :

Fils de A :

Père de I, J et K :

Branche la plus longue :

Feuilles :

Sous-arbres de :

A

C

F

Degré des nœuds :

A =

F =

H =

## Arbre binaire

### Définition

Un arbre binaire est un arbre dont chaque nœud possède au maximum deux descendants directs :

* Le fils gauche;
* Le fils droit.

*Exemples :*

### Représentation en mémoire

Un nœud se représente comme suit :

PGche

Données

PDrt

L’arbre binaire précédent est représenté en mémoire par le schéma suivant :

Racine

PGche

L

PDrt

PGche

O

PDrt

PGche

F

PDrt

PGche

T

PDrt

PGche

J

PDrt

PGche

B

PDrt

PGche

R

PDrt

PGche

C

PDrt

PGche

A

PDrt

## Arbre binaire de recherche

### Définition

Un arbre binaire de recherche est un arbre binaire dont chaque élément est composé d’un champ de données particulier qui permet d'identifier le nœud. Ce champ est appelé la **clé d’identification du nœud**.

La clé d’un nœud permet d’organiser l’arbre de telle sorte que :

* La clé est **supérieure à toutes celles de ses descendants de gauche** ;
* La clé est **inférieure à toutes celles de ses descendants de droite**.

*Exemple :*

### Représentation en mémoire

Un nœud se représente comme suit :

PGche

***Clé***

Données

PDrt

L’arbre binaire de recherche précédent est représenté en mémoire par le schéma suivant :

Racine

PGche

***25***

L

PDrt

PGche

***20***

F

PDrt

PGche

***36***

O

PDrt

PGche

***10***

B

PDrt

PGche

***23***

J

PDrt

PGche

***3***

A

PDrt

PGche

***17***

C

PDrt

PGche

***58***

T

PDrt

PGche

***42***

R

PDrt

💣 A partir d'ici, ce chapitre ne traitera que des **arbres binaires de recherche**.

## Notations

Par convention, on utilise, comme pointeurs, les noms de variable suivants :

* **Ptr** variable contenant l’adresse du nœud courant ;
* **Racine** variable contenant l’adresse du nœud racine ;
* **Ptr🡪PGche** variable contenant l’adresse du fils gauche ;
* **Ptr🡪PDrt** variable contenant l’adresse du fils droit ;
* **Ptr🡪Données** permet d’accéder au contenu des données ;
* **PtrNouv** variable contenant l’adresse d'un nouveau nœud.

## Opérations

Les opérations pouvant être réalisées sur les arbres sont les suivantes :

* Placement sur la racine ;
* Vérification de l’arbre (vide ou non) ;
* **Recherche** d'un élément de l'arbre ;
* **Ajout** d'un élément dans l'arbre ;
* **Suppression** d'un élément de l'arbre ;
* **Parcours** de l'arbre ;
* **Comptage** du nombre d'éléments de l'arbre.

## Exemples d'utilisation

### Dans la vie quotidienne

Il existe de nombreux exemples d'utilisation des arbres dans la vie quotidienne, dont :

* Les arbres généalogiques ;
* Les tableaux de tournoi ;
* Les organigrammes de société.

### En informatique

La gestion des fichiers par un système d'exploitation se fait de manière hiérarchique, et ce, en utilisant les arbres. En effet, à la base de l'arborescence se trouve un répertoire appelé la racine. Ce répertoire peut contenir des fichiers et des répertoires, qui eux-mêmes peuvent contenir la même chose. Si les fichiers et les répertoires sont placés de manière cohérente, la recherche de fichier est relativement aisée et rapide. C'est le cas grâce à cette structuration sous forme d'arbre.

Lorsqu'on utilise une application, il est fréquent que celle-ci permette l'ouverture de nouvelles fenêtres afin de permettre la visualisation de certaines informations. Cependant, comme lorsqu'on fait appel à un module à partir d'un autre, il faut garder les informations concernant la fenêtre dont on vient. Pour cela, le système d'exploitation utilise une technologie basée sur l'arborescence.

## Manipulation des arbres

Comme pour les listes chaînées, les nœuds d'un arbre ne sont pas nécessairement placés consécutivement en mémoire. Ils peuvent être disséminés un peu partout. On utilise donc à nouveau les instructions permettant d’allouer un espace mémoire de manière dynamique pour l’ajout de nouveaux nœuds. On a également recours à l'instruction qui permet de libérer cet espace lors de la suppression d'un nœud.

On accède aux éléments de l'arbre en passant d'un nœud à l'autre via le pointeur vers le fils droit – ***Ptr🡪PtrDrt*** – ou le fils gauche – ***Ptr🡪PtrGche*** – du nœud courant – ***Ptr*** –, et ce, à partir de la racine de l'arbre – ***Racine***.

Ci-après se trouvent les différents diagrammes d'actions qui permettent de réaliser les principales opérations sur les arbres :

* **Recherche** d'un élément de l'arbre ;
* **Ajout** d'un élément dans l'arbre ;
* **Suppression** d'un élément de l'arbre.

Ces trois diagrammes d'actions sont réalisés sur base d'un énoncé commun décrit au point suivant. Il est important de noter que ces diagrammes d'actions n'utilisent **pas le principe de la récursivité**.

### Énoncé commun

On désire permettre la gestion des inscriptions à une *newsletter*.

Lors de l'inscription, chaque abonné doit encoder les informations qui le concernent, c’est-à-dire :

* son numéro d'abonné ;
* son nom ;
* sa nationalité.

Chaque nœud de la liste représente donc un abonné.

La **structure** d'un nœud est : La **représentation graphique** d'un nœud est :

***PtrGche***

NumAbon

NomAbon

NatAbon

***PtrDrt***

Skieur

***PtrGche***

NumAbon

NomAbon

NatAbon

***PtrDrt***

### Recherche d’un élément

*Exemple :*

Soit un arbre d'abonnés, dont le pointeur de début est *Racine*, écrire le diagramme d’actions qui permet de rechercher un abonné dont le numéro, NumAbonLu, a été entré par l'utilisateur au clavier. On demande d'afficher un message de réussite ou d'échec.

* Observation
  + Donnée: la racine ;
  + Entrée : le numéro de l'abonné recherché ;
  + Sortie : message de réussite ou d'échec.
* Diagramme
  + Opération : recherche dans un arbre ;
  + Variables : RacineSki, NumAbonLu, Ptr ;
  + Initialisation : *Ptr* = *RacineSki* ;
  + Diagramme d'actions : Recherche.

┌─\* Recherche

│

│ *Ptr = Racine*

│

│╔═ **do while** **(** (*Ptr* ≠ NULL) **AND** (*NumAbonLu* ≠ *Ptr🡪*NumAbon) **)**

│║

│║┌─ **if** **(***NumAbonLu* < *Ptr🡪*NumAbon**)**

│║│

│║│ *Ptr* = *Ptr🡪 PtrGche*

│║├─ **else**

│║│

│║│ *Ptr = Ptr🡪 PtrDrt*

│║└──

│╙─

│┌─ **if** **(***Ptr = NULL***)**

││

││ Sortir "Absent”

│├─ **else**

││

││Sortir "Présent”

│└──

└──

### Ajout d’un élément

*Exemple :*

Soit un arbre d'abonnés, dont le pointeur de début est Racine, écrire le diagramme d’actions qui permet d'ajouter un abonné dans l'arbre.

* Observation
  + Donnée : la racine ;
  + Entrée : les informations concernant l'abonné à ajouter ;
  + Sortie : aucune.
* Diagramme

### Suppression d’un élément

*Exemple :*

Soit un arbre d'abonnés, dont le pointeur de début est Racine, écrire le diagramme d’actions qui permet de supprimer un abonné dont le numéro, *NumAbonLu*, a été entré par l'utilisateur au clavier.

* Observation
  + Donnée : la racine ;
  + Entrée : le numéro de l'abonné à supprimer ;
  + Sortie : aucune.
* Diagramme

## Parcours

### Introduction

Après avoir étudié les trois algorithmes permettant la manipulation des élément d'un arbre, il est nécessaire de bien comprendre la manière dont ceux-ci sont traités en général. Il s'agit donc de déterminer comment parcourir un arbre binaire de recherche.

### Récursivité

Le parcours d'un arbre binaire de recherche peut se fait de manière itérative, mais la façon la plus rapide et efficace de le faire est d'utiliser un **algorithme récursif**.

Un **algorithme récursif** est un algorithme **qui s'appelle lui-même**.

La récursivité est un domaine très intéressant de l'informatique, un peu abstrait, mais très élégant ; elle permet de résoudre certains problèmes d'une manière très rapide, alors que les résoudre de manière itérative prendrait beaucoup plus de temps et de structures de données intermédiaires.

La récursivité utilise toujours la pile du programme en cours (voir pile d'appel chapitre 6 point 6.3.4.). Pour rappel, supposons que l’on soit dans un module Mod1 dans lequel nous avons des variables locales, et que, dans ce module, nous faisions appel à un module Mod2 ; afin de pouvoir continuer l'exécution du Mod1 après avoir fini Mod2, il faut stocker quelque part les variables de Mod1 ; c'est le rôle de la pile d'appel. Tout ceci est géré de façon transparente pour l'utilisateur. Dans un algorithme récursif, toutes les variables locales sont stockées dans la pile d'appel, et empilées autant de fois qu'il y a d'appels récursifs. Donc la pile se remplit progressivement. Ensuite, les variables sont désempilées.

Un algorithme récursif comporte un module qui s'appelle lui-même, alors qu'une procédure non récursive ne comporte que des appels à d'autres modules.

Tout algorithme récursif comporte une instruction (ou un bloc d'instructions) nommée "point terminal" ou "point d'appui" ou "point d'arrêt", qui indique que le reste des instructions ne doit plus être exécuté. Ce point est atteint lorsque les paramètres de l'appel récursif, étant changés à chaque fois, l'ordinateur rencontre un ensemble de paramètres vérifiant le test d'arrêt, et que le module récursif atteint le "fond" (point terminal). Le fait de ne rien changer dans les paramètres impliquerait que l'ordinateur effectuerait un appel infini à ce module, ce qui se traduirait en réalité par un débordement de pile et l’arrêt de l'exécution de l'algorithme en cours. Ensuite, lorsque le point d'arrêt est atteint, les paramètres ainsi que les variables locales sont désempilées au fur et à mesure qu'on remonte les niveaux.

### Parcours d'un arbre binaire de recherche

Il existe trois manières de parcourir un arbre :

* Parcours préfixe (pré-ordre) : Racine – Gauche – Droite ;
* Parcours suffixe (post-ordre) : Gauche – Droite – Racine ;
* Parcours Infixe (in-odre) : Gauche – Racine –Droite ;

#### Parcours préfixe

On commence à la Racine, on parcourt l’arbre en visitant tous les nœuds dans le sens contraire des aiguilles d’une montre.

On n’affiche les nœuds que lors de leur première visite.

*Exercice :*

* Parcours :
* Affichage :

#### Parcours suffixe

On commence à la Racine, on parcourt l’arbre en visitant tous les nœuds dans le sens contraire des aiguilles d’une montre.

On affiche les nœuds que lors de leur dernière visite.

*Exercice :*

* Parcours :
* Affichage :

#### Parcours infixe

On commence à la Racine, on parcourt l’arbre en visitant tous les nœuds dans le sens contraire des aiguilles d’une montre.

On affiche l’arbre aplati, la racine apparaît entre les fils (c'est-à-dire par ordre croissant).

*Exercice :*

* Parcours :
* Affichage :

### Algorithme récursif du parcours infixe

┌─\* Parcours

│

│

│

└──

Parcours Infixe

↓ Racine

Parcours Infixe

↓ Ptr

┌─\* Parcours Infixe

│

│┌─ **if** **(***Ptr* ≠ NULL**)**

││

││

││

││Sortir Ptr🡪Donnée

││

││

││

│└──

└──

Parcours Infixe

↓ Ptr🡪PtrGche

Parcours Infixe

↓ Ptr🡪PtrDrt

# Conclusion

Ce syllabus constitue une introduction aux principes de bases de la programmation. Il aborde les diverses structures de données fréquemment utilisées ainsi que la manière de les manipuler.

En effet, le chapitre 1 a permis d'introduit le cours en décrivant ce qu'est la programmation, ainsi qu'en définissant la notion incontournable de variable.

Le chapitre 2 a permis de faire les premiers pas en présentant les différents cas d'alternatives. La rédaction d'une condition a également fait l'objet d'une analyse particulière.

Ensuite, un tour complet de la notion de répétitive a été réalisé au chapitre 3.

Les structures de contrôle étant maîtrisées, il était question, au chapitre 4, d'une première structure de données plus élaborée : les tableaux à simple indice. Un détour y a également été fait concernant les modules, afin de mieux structurer les algorithmes.

Pour pouvoir utiliser cette structure de données de manière efficace, le chapitre 5 a présenté diverses méthodes de tri.

Le chapitre 6 a permis de comprendre comment utiliser la notion de fichier en programmation.

Les piles et les files, deux exemples d'utilisation des tableaux à simple indices, ont été décrites au chapitre 7.

Les structures de bases ayant été vues, aussi bien au niveau des données qu'au niveau du contrôle du traitement de celles-ci, la première structure complexe a été abordée au chapitre 8 : les tableaux à double indice.

Le chapitre 9 a donné un aperçu complet de ce que sont les listes chaînées en passant par la notion de pointeur.

Enfin, le chapitre 10 cloture le cours en traitant des arbres ainsi que de la récursivité.

Il est cependant important de rappeler aux lecteurs que la programmation est un savoir procédural. La lecture de ce document ne suffit donc pas à sa maîtrise. Il est nécessaire de réaliser en parallèle des exercices visant à mettre en pratique les notions qui y sont abordées afin que ce savoir procédural devienne un savoir -faire.

Enfin, je tiens à remercier les personnes qui ont participé à la rédaction de ce syllabus par leur soutien et leurs relectures fréquentes : **Chantal Bertrand**, **Christine Charlier**, **Isabelle Charlier**, **Corinne Derwa**, **Natalie Dheur**, **Christiane Glime**, etc.

# Bibliographie

Ce syllabus est inspiré de plusieurs ouvrages et sites Internet :

* + Images pour programmer : Apprendre les concepts de base
  + Charles Duchateau – De Boeck Université
  + Initiation aux raisonnements de la programmation
  + Roland Lesuisse & Anne Borsu – Presses Universitaires de Namur
  + Initiation à la programmation
  + Claude Delannoy - Eyrolles
  + ALGORITHMIQUE ET PROGRAMMATION POUR NON-MATHEUX
  + Christophe Darmangeat – Université Paris 7.
  + <http://www.pise.info/algo/>
  + Encyclopédie informatique en ligne
  + <http://www.commentcamarche.net>
  + Encyclopédie générale en ligne
  + <http://www.wikimedia.org/be-portal/>

De plus, il a été réalisé sur base des documents suivants :

* + Notes de cours de **Christiane Glime** ;
  + Préparation de cours de Chantal Bertrand et Isabelle Charlier ;
  + Syllabus de remédiation de **Christine Charlier** et **Natalie Dheur**.

1. <http://deptinfo.cnam.fr/Enseignement/CycleA/AMSI/cours_systemes/07_notion_fichier/fichie.htm> [↑](#footnote-ref-1)
2. <http://fr.wikipedia.org/wiki/Pile_%28informatique%29> [↑](#footnote-ref-2)
3. La segmentation offre une vue de la mémoire plus consistante pour l'utilisateur. En effet, celui-ci ne considère pas la mémoire comme une suite de pages mais plutôt d'espaces dédiés à une utilisation particulière comme par exemple : le code d'un programme, les données, la pile, etc. La segmentation reflète cette organisation. Chaque objet logique est désigné par un segment. Dans un segment, l'adressage se fait à l'aide d'un déplacement. Le couple (segment, déplacement) se traduit en adresse mémoire par le biais d'une table de segments contenant deux champs : la base est l'adresse de début du segment et la limite la dernière adresse du même segment. [↑](#footnote-ref-3)
4. <http://fr.wikipedia.org/wiki/File> et <http://fr.wikipedia.org/wiki/Mom> [↑](#footnote-ref-4)
5. <http://fr.wikipedia.org/wiki/%C3%89cran_%C3%A0_cristaux_liquides> [↑](#footnote-ref-5)
6. <http://www.univ-tlse2.fr/sed/geographie/COURSTLD/LC17.html> [↑](#footnote-ref-6)
7. <http://fr.wikipedia.org/wiki/Structure_de_donn%C3%A9e> et <http://fr.wikipedia.org/wiki/Liste> [↑](#footnote-ref-7)
8. <http://lifc.univ-fcomte.fr/~mbala/chap3.htm> [↑](#footnote-ref-8)
9. <http://deptinfo.cnam.fr/Enseignement/CycleA/AMSI/cours_systemes/07_notion_fichier/fichie.htm> [↑](#footnote-ref-9)