Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Direction Générale des Études Technologiques


# SUPPORT DE COURS

## Algorithmique et structures des données 2

**Enseignants :**

Ahlem Tayachi

Fatma Laribi

Maali Ben Khadija

Nesrine Ben Hadj Hammouda

Sameh Mansouri

AU 2009-2010
(Semestre 2)

# Présentation du cours

## Fiche matière

- **Domaine de formation** : Sciences et technologies
- **Mention** : Technologie de l'informatique (TI)
- **Parcours** : Tronc Commun
- **Semestre** : S2
- **Unité d'enseignement** : Programmation et structures dynamiques
- **Volume horaire** : 45 heures (22,5 cours – 22,5 TD)
- **Coefficient** : 2

## Résumé du cours

Le cours d'algorithmique et structure de données est destiné aux étudiants de première année du tronc commun de l'option Technologies de l'Informatique.
Ce cours définit les enregistrements et la façon de les utiliser afin de pouvoir maîtriser par la suite les structures de données dynamiques.
Il introduit ensuite les types de données abstraits pour pouvoir les implémenter dans la représentation des données et les différentes opérations associées.
Après avoir spécifier le type de donnée abstrait, ce cours présente les structures linéaires simples, telles que les listes, les piles et les files pour terminer par des structures pour des structures plus complexes telles que les graphes et les arbres.
Enfin, ce cours introduit la notion de complexité algorithmique en donnant un aperçu sur la complexité de quelques algorithmes.
Mentionnons que ce cours nécessite comme pré-requis l'algorithmique et structures de données 1.

## Objectifs généraux

- Ce cours vise à approfondir les compétences acquises par l'apprenant en algorithmique.
- Savoir écrire des algorithmes se rapportant à des structures dynamiques : listes, piles, files, arbres, etc.
- Etre capable de transformer un schéma itératif simple en un schéma récursif.

## Pré-requis

- Algorihtmique1
- Programmation1

## Evaluation

- Interrogation Orale
- Un devoir surveillé d'une heure
- Examen Final écrit de 2 heures sur tout le programme

## Moyens Pédagogiques

- Exposé informel
- Tableau
- Support du cours
- Travaux dirigés

## Méthodologie

- Le cours s'articule autour de 2 types d'activités, les cours intégrés et les séances de travaux dirigés.
- Les séries d'exercices sont distribués aux étudiants et corrigées en classe.

# SOMMAIRE

## Chapitre8: Complexité des algorithmes



## DS et examen

# Chapitre 1 :
# Les enregistrements

## Vue d'ensemble

Cette leçon présente la définition et l'utilisation des enregistrements.

### Objectifs

Ce chapitre a pour objectifs de permettre aux étudiants d'acquérir les connaissances de base se rapportant aux objectifs spécifiques suivants:

- Etre capable d'utiliser et créer des nouveaux types de données selon le besoins du programme (les enregistrements).
- Savoir écrire un enregistrement

### Pré-requis

- Algorithmique 1

### Durée

- 3 H

## Eléments de contenu

1. Les types
2. Définition d'un enregistrement
3. Utilisation d'un enregistrement
4. Les types abstraits
5. Exercices d'application
6. Solutions des exercices

# 1.1 Les types

## 1.1.1 Les types de base

- Toute variable utilisée dans un algorithme doit avoir un type qui caractérise l'ensemble de valeur qu'elle peut prendre dans cet algorithme, ce type peut être un type de base (prédéfinit) ou un type composé qui est définit par l'utilisateur.

- Les types de base se divisent en 2 catégories : les types simples et les types complexe.

## 1.1.2 Les types simples

Les types de simple qui peuvent être utilisé pour représenter les variables d'un algorithme sont :

- **Entier** : la variable peut prendre des valeurs entières positives ou négatives, pour utiliser ce type on utilise le mot clé « entier »

- **Ree**l : la variable peut prendre des valeurs réelles, pour utiliser ce type on utilise le mot clé « reel »

- **Caractere** : la variable peut prendre tout caractère qu'on peut saisir avec un clavier standard, pour utiliser ce type on utilise le mot clé « reel »

- **Booleen** : la variable a 2 valeurs VRAI ou FAUX, pour utiliser ce type on utilise le mot clé « booleen »

## 1.1.3 Les types complexes

Les types simples servent pour définir les types complexes qui permettent de représenter des structures de données plus compliqués, ces types sont :

**- Tableau** :

- la variable est formée par l'union de plusieurs cases mémoire qui contiennent chacune le même type.
- la déclaration d'un tableau se fait comme suit : « nom : **Tableau** [i_deb .. i_fin] de **type ».**
- **i_deb et i_fin** représentent l'indice de début et l'indice de fin du tableau.

**- Chaîne de caractère** :

- C'est la concaténation de plusieurs caractères qui finissent par le marqueur de fin de chaîne. C'est ainsi un tableau de caractères qui contient comme dernier caractère un caractère spécial pour représenter le marqueur de fin de chaîne.
- La déclaration se fait comme suit : « nom : **chaine** [ Max ] »
- Max est le nombre maximum de caractère.

# 1.2 Définition d'un enregistrement

## 1.2.1 Utilité

Contrairement aux types de base les types composés sont définit par l'utilisateur. Les **types de bases** servent pour définir de nouveaux **types composés**. Ces nouveaux types servent pour représenter des structures de données plus complexes qu'on ne peut pas représenter par les types de bases.

*Exemple :*

Si on veut représenter les données relatives à une personne telle que le nom et l'âge dans une seule variable ou une seule structure de donnée, on ne peut pas faire ça avec les types de bases et particulièrement avec les tableaux. On peut utiliser un nouveau type appelé **enregistrement** qui contient 2 champs, un pour représenter le nom de type chaîne de caractère et le 2ème pour représenter l'age de type entier. On remarque qu'on ne peut pas représenter la personne par un tableau car elle est formée de 2 types différents.

Ainsi on définit le type personne par un enregistrement comme suit :

Personne = **Enreg**

nom : **chaîne** [30]

age : **entier**

**FinEnreg**

## 1.2.2 Définition

L'enregistrement est une structure de données (ou un type) composée, qui est formé par plusieurs autres structures de données de nature différentes. Il permet ainsi de les regrouper dans une seule structure de données.

## 1.2.3 Syntaxe

```
Nom = Enreg
    Champ1 : type
```

…

Champs N : type

**FinEnreg**

On désigne par :

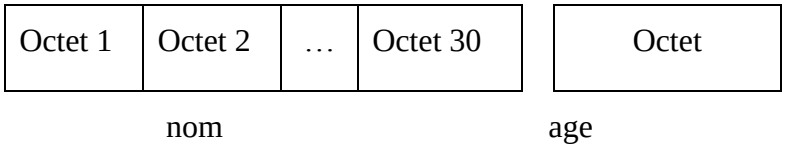
- **Nom** : représente le nom de l'enregistrement.

- **Champs 1…N** : les noms des champs qui constituent l'enregistrement.

- **Type** : est le type de chaque champ.

### 1.2.4 Représentation

Les **enregistrements** sont représentés en mémoire sous forme de suite de zones contiguës qui servent chacune à représenter les différents champs, ces zones sont de taille différentes puisque chaque champ a un type différent. Par exemple l'**enregistrement** personne sera représenter en mémoire comme suit :




## 1.3 Utilisation d'un enregistrement

### 1.3.1 Déclaration

La déclaration d'un enregistrement se fait comme étant **un nouveau type** définit par l'utilisateur dans l'algorithme, il doit être déclaré dans la partie **Type** avant d'être utilisé comme type de variable dans la partie **Var**.

### 1.3.2 Accès

L'enregistrement qui est une structure de données composé de plusieurs champs n'est pas accessible dans sa totalité, donc pour faire un accès à un enregistrement on accède à tous les champs un par un. Et pour utiliser un champ on écrit le **nom de la variable** de type enregistrement suivi de **« point »** suivi **du champ** auquel on veut accéder :
**Nom_enregistrement.champ_desire**

### 1.3.3 Exemple

On veut écrire un algorithme qui lit le nom, le prénom et la moyenne générale d'un étudiant puis il affiche si cet étudiant a réussi ou non selon que sa moyenne est >= 10 ou non. Utiliser un enregistrement pour représenter l'étudiant.

```
Algorithme Resultat
Type
  Etudiant = Enreg
      nom : chaîne [30]
      prenom : chaîne [30]
      MG : reel
      R : caractere
  FinEnreg
Var
  E : Etudiant
Debut
  Lire (E.nom)
  Lire (E.prenom)
  Lire (E.MG)
  Si (E.MG >= 10) Alors
      E.R ← 'A'
      Ecrire ("Admis ")
  Sinon
      E.R ← 'R'
      Ecrire ("Refusé ")
  FinSi
Fin
```

## 1.4 Les types abstraits

Les enregistrements permettent de représenter des structures de données complexes et formées par des types non homogènes. Mais ils ne présentent pas une abstraction au niveau des structures de données de l'algorithme. Pour résoudre ce problème on fait recours à une généralisation du type enregistrement par les types abstraits. Ce qui donne une indépendance vis-à-vis d'une implémentation particulière et les données sont considérées d'une manière abstraite.

### 1.4.1 Définition

Un type abstrait permet de représenter les données, les opérations faites sur ces données et les propriétés de ces opérations, dans une même structure de données.

### 1.4.2 Exemple :

Si on veut représenter un Compte Bancaire par un type abstrait il sera composé des données et des opérations concernant les Comptes Bancaires :

- **Les données seront :**
    - Une variable pour représenter le numéro du compte
    - Une variable pour représenter le solde
    - Une variable pour représenter le nom du propriétaire
- **Les opérations sur les comptes bancaires:**
    - Une fonction pour créer un nouveau compte
    - Une fonction pour consulter un compte
    - Une procédure pour débiter un compte
    - Une procédure pour créditer un compte
- **Les propriétés des opérations :**
    - Pour débiter il faut que le solde soit positif
    - Seuls les salariés peuvent créer des comptes.

### 1.4.3 Signature d'un type abstrait

La signature d'un type abstrait de données est définie par :

- Les noms des types utilisés pour représenter les données (réels, entiers, …).
- Les noms des opérations et leurs profils.

La signature décrit la syntaxe du type abstrait mais elle ne le définie pas réellement. Ainsi pour l'exemple précédent la signature est :

- **Les types des données :**
    - entier
    - reel
    - chaine[20]
- **les opérations et leurs profils :**

- Creer_compte : chaine[20] ➔ entier
- Consulter _compte : entier ➔ reel
- Debiter : entier * reel
- Crediter : entier * reel

## 1.5 Exercices d'application

**Exercice 1 : Saisie d'une fiche**

Une **fiche** contient les coordonnées suivantes:

-Nom (chaîne de caractères de longueur 10)

-Prénom (chaîne de caractères de longueur 10)

-Age (entier)

-Note (réel)

Ecrire un algorithme puis un programme C qui permet de créer la structure ci-dessus, saisir une fiche et l'afficher.

**Exercice 2 : Somme et produit de deux nombres complexes**

Ecrire un algorithme puis un programme C qui lit deux nombres complexes **C1** et **C2** et qui affiche ensuite leur somme et leur produit.

On utilisera les formules de calcul suivantes :

- $(a + bi)+(c + di) = (a + c) + (b + d)i$
- $(a + bi)\times(c + di) = (a\times c - b\times d) + (a\times d + b\times c)i$

**Exercice 3 : Coordonnées d'un point**

Créer une structure **Point** qui contient les champs suivants :

- **num : entier**
- **x et y : reel**

Ecrire un programme C qui permet de saisir 4 points, les ranger dans un tableau puis les afficher.

**Exercice 4 : Tableau des employés**

Ecrire un programme C qui permet de créer un tableau **TabEmp** qui contiendra les informations sur les 10 employés d'une entreprise (Matricule, Nom, Salaire, Etat_Civil), le remplir puis afficher le nombre d'employés dont le salaire est compris entre 500 et 700 D.

## 1.6 Solutions des exercices

**Exercice 1 :**

```
Algorithme Saisie_Fiche
Type
   fiche=enreg
            nom : chaine[10]
            prenom : chaine[10]
            age : entier
            note : reel
   FinEnreg
Var
   f : fiche
Début
   Lire (f.nom)
   Lire (f.prenom)
   Lire (f.age)
   Lire (f.note)
   Ecrire (f.nom, f.prenom,f.age,f.note)
Fin
```

**Exercice 2 :**

```
Algorithme CalculComplexe
Type
   Complexe=enreg
            Re : Réel
            Im : Réel
   FinEnreg
Var
   C1, C2, S, P : Complexe
Début
   Ecrire (‘’ Partie réelle du premier nombre :’’)
```

```
    Lire (C1.Re)
    Ecrire (‘’ Partie imaginaire du premier nombre :’’)
    Lire (C1.Im)
    Ecrire (‘’ Partie réelle du deuxième nombre :’’)
    Lire (C2.Re)
    Ecrire (‘’ Partie imaginaire du deuxième nombre :’’)
    Lire (C2.Im)

    S.Re = C1.Re + C2.Re
    S.Im = C1. Im + C2.Im
    Ecrire (‘’Somme=’’, S.Re, ‘’+’’, S.Im, ‘’i’’)

    P.Re = (C1.Re * C2.Re) – (C1. Im * C2.Im)
    P.Im = (C1.Re * C2.Im) + (C1. Im * C2.Re)
    Ecrire (‘’Produit=’’, P.Re, ‘’+’’, P.Im, ‘’i’’)
Fin
```

**Exercice 4 :**

```
Algorithme Personnel
Constantes
    n = 50
Types
    Employé=Struct
            Matricule : Entier
            Nom : Chaîne
            Sal : Réel
            Etat_Civil : Caractère
    Fin Struct
    Tab = Tableau [1..n] de Employé
Variables
    TabEmp : Tab
    I, nb : Entier
```

```
Procédure Remplir (Var T :Tab)
Début
   Pour i de1 à n faire
      Ecrire (‘’ Matricule de l’employé :’’)
      Lire (T[i].Matricule)
      Ecrire (‘’ Nom de l’employé :’’)
      Lire (T[i].Nom)
      Ecrire (‘’ Salaire de l’employé :’’)
      Lire (T[i].Sal)
      Ecrire (‘’ Etat civil de l’employé :’’)
      Lire (T[i].Etat_Civil)
   Fin pour
Fin
Fonction Compter (T : Tab) : Entier
Début
   Compter ← 0
   Pour i de 1 à n faire
      Si (T[i].Sal >=500) ET (T[i].Sal <=700) alors
         Compter ← Compter + 1
      Fin si
   Fin pour
Fin
Début
   Remplir (TabEmp)
   Nb ← Compter (TabEmp)
   Ecrire ( nb, ‘’employés touchent entre 500 et 700 Dt’’)
Fin
```

# Chapitre 2 : Les pointeurs

## Vue d'ensemble

Cette leçon présente la définition et l'utilisation des pointeurs.

### Objectifs

Ce chapitre a pour objectifs de permettre aux étudiants d'acquérir les connaissances de base se rapportant aux objectifs spécifiques suivants:

- Etre capable d'utiliser les pointeurs.
- gérer un ensemble fini d'éléments dont le nombre varie au cours de l'exécution du programme.
- Savoir écrire un algorithme en utilisant des variables dynamiques à l'aide du pointeur.

### Pré-requis

- Algorithmique 1
- Les enregistrements

### Durée

- 3 H

## Eléments de contenu

1. Rappel
2. Les variables dynamiques
3. Les pointeurs
4. Exercices d'applications
5. Solutions des exercices

# 2.1 Rappel

## 2.1.1 Mémoire centrale (MC)

La mémoire centrale mémorise les données en cours de traitement par l'unité de traitement (UT). Elle contient les instructions et les données du programme en cours. L'UT lit une instruction, les opérandes de cette instruction, calcule le résultat puis l'écrit dans la mémoire centrale. En entrée, l'UT lit une donnée depuis une unité d'Entré\sortie et l'écrit dans la MC et inversement pour une sortie.

## 2.1.2 Structure de la mémoire centrale

La mémoire centrale est composée de cellules mémoire. Une cellule est un ensemble de 8 circuits électroniques. Chaque circuit est capable de soutenir dans un de deux états :

**Haut** représente le bit **1**
**Bas** représente le bit **0**

Une cellule est donc un ensemble de 8 bits dit un octet (**8 bits = 1 octet).** Les cellules de la mémoire sont numérotées de **0** à **n-1**, avec n est le nombre de cellules qu'elle contient.
Pour mémoriser plus d'informations, on utilise deux cellules adjacentes (un mot) ou 4 cellules adjacentes (un double mot).
Une variable est en faite une petite zone mémoire issue de n octets que l'on s'alloue et dans laquelle les informations sont rangées.

## 2.1.3 La notion d'adresse

Lors de la compilation d'un programme, l'ordinateur réserve dans sa mémoire une zone mémoire pour chaque variable déclarée. C'est à cet endroit que la valeur de la variable est stockée. Il associe alors au nom de la variable l'adresse de stockage. Ainsi, pendant le déroulement du programme, quand il rencontre un nom de variable, il va chercher à l'adresse correspondante la valeur en mémoire.

En langage C, il est possible de mettre une valeur dans une variable à l'aide de l'opérateur = ou de la fonction **scanf( ).** L'adresse mémoire est accessible en faisant précéder la variable de l'opérateur **&.**

Exemple :

```
char  car = 'c';
printf ("adresse de car %lx", &car);
```

# 2.2 Les pointeurs

## 2.2.1 Définition

Un pointeur est une variable qui a pour valeur l'adresse mémoire d'une autre variable sur laquelle il pointe. Le pointeur est déterminé par le type sur lequel il pointe (variable, premier élément d'un tableau, ….).

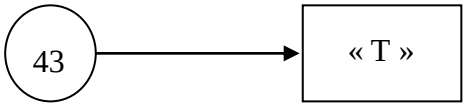
## 2.2.2 Déclaration d'un pointeur

En algorithmique, on déclare un pointeur en précédant son type de base par le caractère « ^ ».
*Syntaxe :*

**Nom_pointeur : ^ type**

*Exemple 1 :*

**P : ^caractère** → déclarer un pointeur appelé **P** sur une variable de type caractère.




- La variable pointeur **P** a pour valeur l'adresse « 43 ». Elle pointe sur l'espace mémoire « **P^** » à l'adresse **43** dont le contenu est le caractère « **T** ».

- La déclaration d'une variable de type pointeur a pour effet la réservation d'une case mémoire qui va contenir l'adresse de la variable pointé.

*Exemple 2 :*

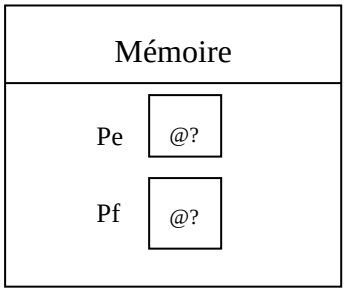
Si on déclare 2 pointeurs **Pe** et **Pf**

- Pe pointe vers une variable de type entier ;

-Pf pointe vers une variable de type réel,

→Ceci implique la création de deux variables qui **contiennent respectivement l'adresse d'un entier et l'adresse d'un réel** comme suit :

Pe : ^entier
Pf : ^reel




## 2.2.3 Création d'un pointeur

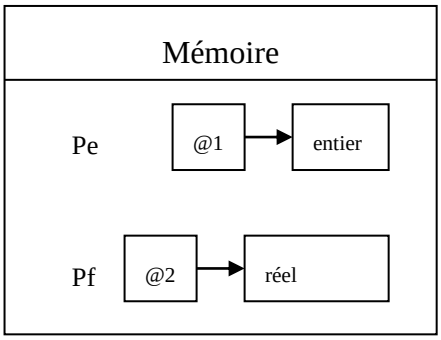
Pour utiliser la variable pointé **il faut d'abord la créer**, ce qui a pour effet de réserver un bloc en mémoire capable de stocker la valeur de cette variable. Pour créer une variable pointé, on utilise l'instruction **Allouer**.

*Syntaxe :*

**Allouer (**nom_pointeur**)**

*Exemple :*

Pour créer la variable pointé par **Pe** on écrit : **Allouer** (Pe), et pour créer la variable pointé par **Pf** on écrit : **Allouer** (Pf).




- L'instruction **Allouer** (P) :
  - Réserve un bloc mémoire de la taille adéquate pour contenir la valeur de la variable pointée par P.
  - Récupère l'adresse de ce bloc et la met dans la variable P.

## 2.2.4 Utilisation d'un pointeur

Pour accéder à la variable pointé par un pointeur appelé P on utilise l'opérateur ^.

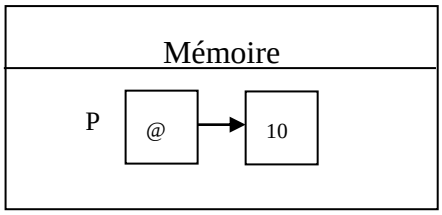
*Syntaxe :*

**P^ :** désigne la variable pointée par P.

Exemple :

L'algorithme suivant stocke la valeur 10 dans une variable de type pointeur.

```
Algorithme Pointeur_element
Var
  P : ^entier
Debut
 Allouer (P)
 P^ ← 10
Fin
```



### 2.2.5 Suppression d'une variable pointé

Lorsqu'une variable pointée n'a plus d'utilité, il est possible de la supprimer et de rendre disponible l'espace mémoire qu'elle occupe. L'instruction qui réalise cette tâche est **« Liberer ».**

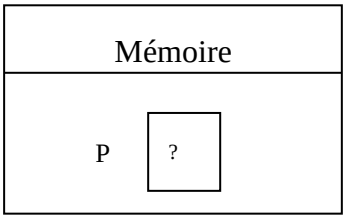
*Syntaxe :*

**Liberer (nom_pointeur)**

*Exemple :*

L'exemple suivant montre l'effet de l'instruction « **Liberer »** sur la mémoire.

```
Algorithme exemple
Var
  P : ^entier
Debut
 Allouer (P)
 P^ ← 10
 Liberer (P)
 P^ ← 4 {erreur}
Fin
```




- « **Liberer (P) »** supprime la variable pointé par **P** et libère l'espace mémoire occupé par cette variable. Par contre la variable pointeur « **P** » n'est pas supprimée mais son contenu n'est plus l'adresse de la variable pointé mais une adresse non significative.

Remarque :

- L'utilisation de la variable pointé après l'instruction **Liberer** provoque une erreur.

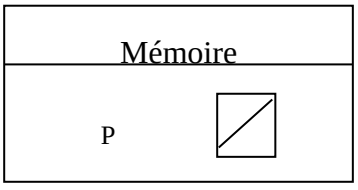
### 2.2.6 Arithmétiques sur les pointeurs

#### 2.2.6.1 Initialisation d'un pointeur

Pour initialiser un pointeur on lui affecte une valeur constante appelé « **Nil** ». Cette constante signifie que le pointeur ne pointe sur rien.

*Syntaxe :*

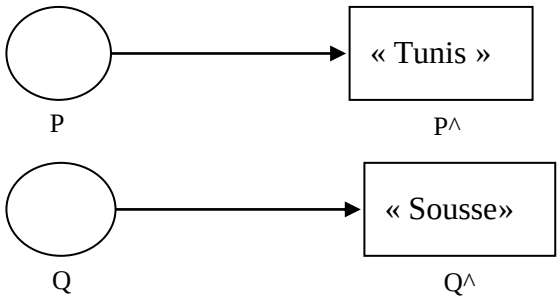
**P ← Nil**



*Exemple :*

On peut aussi initialiser un pointeur à partir d'un autre pointeur comme le montre l'exemple suivant :

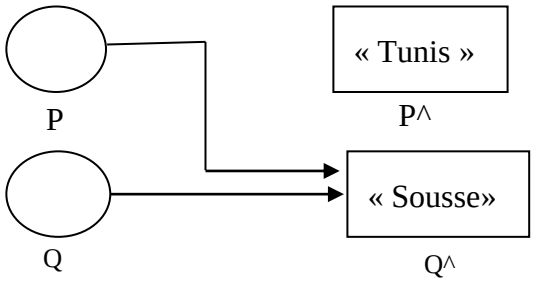
- Supposons qu'on soit dans la situation initiale illustrée par la figure a :



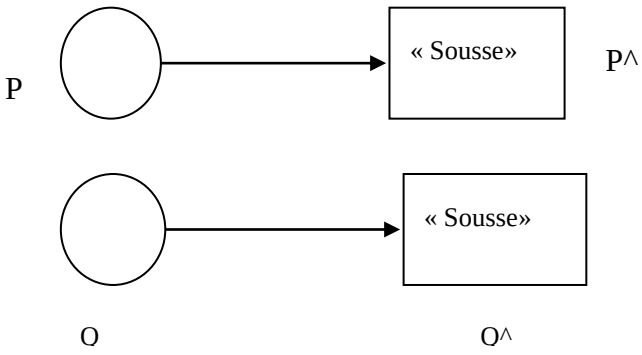

Si l'instruction : **P←Q** est exécuté, nous passerons à la nouvelle situation illustrée par la figure b.

Dans ce cas, on a : **P^ = Q^ = « Sousse »**

- Si on modifie la valeur de **Q^**, **P^** sera également modifié et restera égal à Q^.




- Par contre, si l'instruction : **P^←Q^** est exécuté. On passe à la nouvelle situation illustrée par la figure c.



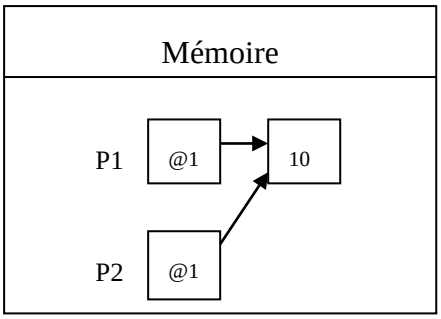
Dans ce cas, on a également **P^ = Q^ = « Sousse » ;** Mais si l'on modifie la valeur de Q^, P^ ne sera pas modifié.

### 2.2.6.2 Affectation d'un pointeur à un autre

Le contenu d'une variable pointeur peut être recopié dans une autre variable pointeur grâce à l'instruction d'affectation, à condition que les 2 pointeurs soit de même type.

*Exemple :*

Les pointeurs P1 et P2 pointent sur un entier. L'instruction **« P1 ← P2 »** permet de copier l'adresse contenu dans P2 dans le pointeur P1. Donc P1 et P2 vont pointer vers la même variable.




### 2.2.6.3 L'opérateur d'adresse

L'opérateur d'adresse permet de récupérer l'adresse en mémoire d'une variable il se note « **&**». Comme le pointeur contient l'adresse d'une variable pointeur on peut utiliser l'opérateur d'adresse pour affecter à un pointeur l'adresse d'une autre variable et ainsi il va pointer vers cette nouvelle variable. La syntaxe est la suivante : « **pointeur ← & variable** »

### 2.2.6.4 Comparaison de pointeurs

- On peut comparer 2 pointeurs entre eux avec l'opérateur = ou <> à condition qu'ils ont le même sous type. Et on peut comparer la valeur **Nil** à n'importe quel pointeur quelque soit son sous-type.
- On peut aussi comparer les valeurs des variables pointées par 2 pointeurs.

### 2.2.6.5 Exemple

```
Algorithme Adresse
Var
  P : ^entier
  A : entier
Debut
```

```
Allouer (P)
A ← 10
P ← & A
Ecrire (A)  {10}
Ecrire (P^) {10}
Liberer (P)
Fin
```

### 2.2.6.6 Tableau récapitulatif

| | Algorithmique | | Langage C | |
|---|---|---|---|---|
| | **Syntaxe** | **Exemple** | **Syntaxe** | **Exemple** |
| **Déclaration pointeur** | nomPointeur :^type | P :^entier | type* nomPointeur | int * P ; |
| **Création pointeur** | Allouer (^type) | Allouer(p) | (type*)malloc(sizeof(type)) | P=(int*)malloc(sizeof(int)) ; |
| **Suppression pointeur** | Libérer (^type) | Libérer(p) | delete (type *) | delete(p) ; |

## 2.2.7 Les tableaux et les pointeurs

En déclarant un tableau, on définit automatiquement un pointeur (on définit en fait l'adresse du premier élément du tableau).

- ***Les tableaux à une dimension:***

Les écritures suivantes sont équivalentes:

**- En algorithmique :**

**tableau: ^entier** **int tableau[10];** **déclaration**

**Allouer (tableau)**

**tableau^** **tableau[0]** **le 1er élément**

**(tableau+i)^** **tableau[i]** **un autre élément**

| | | |
|---|---|---|
| **tableau** | **& tableau[0]** | **adresse du 1er élément** |
| **( tableau + i)** | **& (tableau[i])** | **adresse d'un autre élément** |

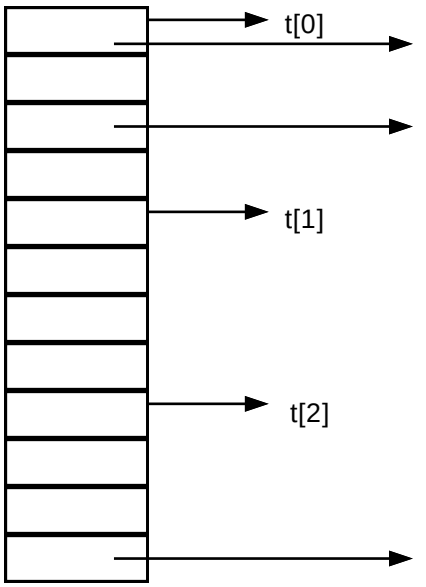
- ***Les tableaux à plusieurs dimensions:***

Un tableau à plusieurs dimensions est un pointeur de pointeur.

Exemple: **int t[3][4];** t est un pointeur de 3 tableaux de 4 éléments ou bien de 3 lignes à 4 éléments.

Les écritures suivantes sont équivalentes:

| | | |
|---|---|---|
| **t[0]** | **&t[0][0]** | **t :adresse du 1er élément** |
| **t[1]** | **&t[1][0]** | **adresse du 1er élément de la 2e ligne** |
| **t[i]** | **&t[i][0]** | **adresse du 1er élément de la ième ligne** |
| **t[i]+1** | **&(t[i][0])+1** | **adresse du 1er élément de la ième +1 ligne** |




REPARTITION DES ELEMENTSEN MEMOIRE

Le tableau a été déclaré ainsi: int t[3][4];

### 2.2.8 Les chaines des caractères et les pointeurs

Les chaînes de caractères sont des **tableaux de caractères**. Leur manipulation est donc analogue à celle d'un tableau à une dimension:

## 2.3 Exercices d'application

### Exercice 1

Compléter le tableau en indiquant les valeurs des différentes variables au terme de chaque instruction du programme suivant (on peut aussi indiquer sur quoi pointent les pointeurs) :

| Programme | A | B | C | P1 | P1^ | P2 | P2^ |
|---|---|---|---|---|---|---|---|
| A, B, C : entier<br>P1, P2 :^entier ; | - | - | - | - | - | - | - |
| A ←1, B ← 2, C ← 3 | | | | | | | |
| P1 ← &A, P2 ← &C | | | | | | | |
| P1^ ← (P2^) ++ | | | | | | | |
| P1 ← P2 | | | | | | | |
| P2 ← &B | | | | | | | |
| P1^ - ←P2^ | | | | | | | |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| P2^= (P2^ )++ | | | | | | | |
| P1^ * ← P2^ | | | | | | | |
| A = ++ P2^ * P1^ ; | | | | | | | |
| P1 = &A ; | | | | | | | |
| P2^ = P1^ /= ^P2 ; | | | | | | | |

## Exercice 2

Soit P un pointeur qui pointe sur un tableau A dont ses valeurs sont {12 , 23, 34 ,45,56,67, 78, 89, 90}

P : ^entier

P=A ;

Quelle valeur fournissent ces expressions :

a) P^+2
**b)** (p+2)^
c) P+1
d) & A [4] +1
**e)** A+3
f) P + (P^ - 10)
g) (P + (P+9)^ -A [8])^

## Exercice 3

On déclare un tableau de 10 entiers A. Ecrire un algorithme et un programme permettant de remplir et d'afficher ce tableau.

Déclarer un pointeur pointant vers l'élément 7 de A. Incrémenter le pointeur c'est-à-dire le faire pointer vers la case suivante. Incrémenter la valeur pointée. Vérifier par l'affichage des résultats.

Afficher le tableau après modifications.

### Exercice 4

Ecrire, de deux façons différentes, un algorithme et un programme en C qui lit 10 nombres entiers dans un tableau avant d'en rechercher le plus grand et le plus petit :

1. en utilisant uniquement le formalisme tableau,
2. en utilisant le formalisme pointeur à chaque fois que cela est possible.

## 2.4 Solutions des exercices

### Exercice 1

| Programme | A | B | C | P1 | P1^ | P2 | P2^ |
|---|---|---|---|---|---|---|---|
| A, B, C : entier<br>P1, P2 :^entier ; | - | - | - | - | - | - | - |
| A ←1, B ← 2, C ← 3 | 1 | 2 | 3 | - | - | - | - |
| P1 ← &A, P2 ← &C | 1 | 2 | 3 | &A | 1 | &C | 3 |
| P1^ ← (P2^) ++ | 3 | 2 | 4 | &A | 3 | &C | 4 |
| P1 ← P2 | 3 | 2 | 4 | &C | 4 | &C | 2 |
| P2 ← &B | 3 | 2 | 4 | &C | 4 | &B | 2 |
| P1^ - ←P2^ | 3 | 2 | 2 | &C | 2 | &B | 2 |
| P2^= (P2^ )++ | 3 | 3 | 2 | &C | 2 | &B | 3 |
| P1^ * ← P2^ | 3 | 3 | 6 | &C | 6 | &B | 3 |
| A = ++ P2^ * P1^ ; | 24 | 4 | 6 | &C | 6 | &B | 4 |
| P1 = &A ; | 24 | 4 | 6 | &A | 24 | &B | 4 |
| P2^ = P1^ /= ^P2 ; | 6 | 6 | 6 | &A | 6 | &B | 6 |

### Exercice 2

a) P^+2 : **14**

**b)** (p+2)^ : **23**

c) P+1 : **&A [2]**

d) & A [4] +1 : **&A [5]**

**e)** A+3 : **&A [4]**

f) P + (P^ - 10) : **&A [3]**

g) (P + (P+9)^ -A [8])^ : **23**

**Exercice 3**

```
Algorithme Pointeur_tableau
Variables
  A : tableau [1..10] de entier
  i : entier
  c :^entier
Début
Allouer(c)
Pour i de 1 à 10 faire
 Ecrire (« donner les éléments du tableau »)
 Lire (A[i])
Fin pour
Pour i de 1 à 10 faire
  Ecrire (A[i])
Fin pour
c←&A[7]
c←A[7+1]
c^ ← c^ +1
ecrire (« l'adesse et le contenu de c sont respectivemnt2 », c, c^).
Pour i de 1 à 10 faire
  Ecrire (A[i])
Fin pour
Liberer(c)
Fin
```

**Exercice4 :**

**a/**

```
Algorithme min_max
Const  NVAL =10
Var
  i,min,max,min_p,max_p :entier
  t : tableau [1..NVAL] de entier
  t :^entier
Début
Pour i de 1 à NVAL faire
  Lire(t[i])
Fin pour
  ecrire(« formalisme tableau »)
  max=min=t[0]
pour i de1 à NVAL faire
  si (t[i]>max) alors
       max←t[i]
  si (t[i]<min) alors
        min←t[i]
 finsi
 ecrire(« val max »,max)
 ecrire(« val min »,min)
 ecrire (« formalisme pointeur »)
 Allouer(t)
 max_p=min_p=t
  pour i de 1 à NVAL faire
 si ((t+i)^ > max_p) alors
    max_p=(t+i)^
 si ((t+i)^ < min_p) alors
       min_p=(t+i)^
```

```
finsi
ecrire(« val_max », max_p)
ecrire(« val_min », min_p)
liberer(t)
Fin
```

**b/**

```
Algorithme min_max
Const  NVAL =10
Var
  i,min,max,min_p,max_p :entier
  t : tableau [1..NVAL] de entier
  t :^entier
Début
Pour i de 1 à NVAL faire
  Lire(t[i])
Fin pour
  ecrire(« formalisme tableau »)
  max=min=t[0]
pour i de1 à NVAL faire
  si (t[i]>max) alors
       max←t[i]
  si (t[i]<min) alors
        min←t[i]
 finsi
 ecrire(« val max »,max)
 ecrire(« val min »,min)
 ecrire (« formalisme pointeur »)
 Allouer(t)
 max_p=min_p=t
  pour i de 1 à NVAL faire
 si ((t+i)^ > max_p) alors
```

```
    max_p=(t+i)^
  si ((t+i)^ < min_p) alors
       min_p=(t+i)^
finsi
ecrire(« val_max », max_p)
ecrire(« val_min », min_p)
liberer(t)
Fin
```

# Chapitre 3 : Les listes

## Vue d'ensemble

Ce Chapitre présente la définition et l'utilisation des listes

### Objectifs

Ce chapitre a pour objectifs de permettre aux étudiants d'acquérir les connaissances de base se rapportant aux objectifs spécifiques suivants:

- Etre capable d'utiliser les listes.
- Connaître une nouvelle structure de données, sa création et surtout les opérations de base sur une liste chaînée
- Savoir écrire un algorithme en utilisant la structure de données listes.

### Pré-requis

- Algorithmique 1
- Les enregistrements
- Les pointeurs

### Durée

- 3 H

## Eléments de contenu

1. Introduction
2. Définition
3. Opérations sur les listes
4. Représentations
5. Algorithmes de quelques opérations sur les listes

## 3.1 Introduction

Le principal problème des données stockées sous forme de tableaux est que celles-ci doivent être ordonnées : le "suivant" doit toujours être stocké physiquement derrière. Imaginons gérer une association. Un tableau correspond à une gestion dans un cahier : un adhérent par page. Supposons désirer stocker les adhérents par ordre alphabétique. Si un nouvel adhérent se présente, il va falloir trouver où l'insérer, gommer toutes les pages suivantes pour les réécrire une page plus loin, puis insérer le nouvel adhérent. Une solution un peu plus simple serait de numéroter les pages, entrer les adhérents dans n'importe quel ordre et disposer d'un index : une feuille où sont indiqués les noms, dans l'ordre, associés à leur "adresse" : le numéro de page. Toute insertion ne nécessitera de décalages que dans l'index. Cette méthode permet l'utilisation de plusieurs index (par exemple un second par date de naissance). La troisième solution est la liste chaînée : les pages sont numérotées, sur chaque page est indiquée la page de l'adhérent suivant, sur le revers de couverture on indique l'adresse du premier. Ainsi, l'insertion d'un nouvel adhérent se fera avec le minimum d'opérations. Nous intéressons dans ce chapitre de cette nouvelle structure de données, sa création et surtout les opérations de base sur une liste chaînée.

## 3.2 Définition

Les listes font parties des structures de données abstraites très couramment utilisées dans les programmes. Une structure de donnée abstraite est un type de donnée défini par l'utilisateur sous forme de deux composantes une pour représenter les données (les notations) et une pour représenter les opérations d'accès aux données et ses caractéristiques.

Une liste stocke un ensemble de données et permet de parcourir cet ensemble du début à la fin dans l'ordre. C'est une suite finie (éventuellement vide) d'éléments, Lorsque elle n'est pas vide, son premier élément est appelé **tête** et la suite constituée des autres éléments est appelée **queue**.

**Remarque :** Les listes servent pour représenter des structures de données dont la taille change souvent ou pour stocker des données dans l'ordre en ayant la possibilité de les mettre à jour quand il y a des changements.

## 3.3 Opérations sur les listes

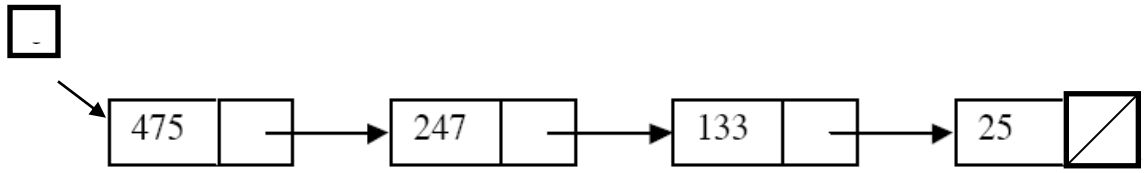
Il existe un certain nombre d'opérations classiques sur les listes.

- Le test si la liste est vide.
- ajouter un élément:
    - au début de la liste
    - à la fin de la liste
    - à un rang donné

- rechercher un élément:
  - résultat booléen
  - résultat = rang de la première occurrence
  - résultat = nombre d'occurrences
- enlever un élément:
  - caractérisé par sa valeur
  - caractérisé par son rang
- calculer la longueur d'une liste (nombre d'éléments)

# 3.4 Représentation

## 3.4.1 Représentation graphique




## 3.4.2 Représentation des données

La structure de donnée qui permet de représenter une liste chaînée dans le cas général est déclarée comme suit :

```
Type
  Cellule = enreg
      valeur : type
      suivant : ^Cellule
  FinEnreg
  Liste : ^Cellule
```

Ainsi le type **Liste** est un pointeur vers le type **Cellule** qui est un enregistrement formé de deux champs : un qui contient la valeur de l'élément donc le type peut être un type quelconque et le 2ème champ contient un pointeur vers la cellule suivante. C'est-à-dire il contient l'adresse de la cellule suivante. La dernière cellule ne pointe vers rien donc elle doit contenir la valeur **Nil.**

Le type **Liste** contient un pointeur vers le type Cellule qui contient l'adresse du 1er élément de la liste.

## 3.5 Algorithmes de quelques opérations sur les listes

Pour simplifier on va travailler sur une liste dont les valeurs sont de type entier. Alors la définition du type liste sera comme suit :

```
Type
  Cellule = enreg
        valeur  : entier
        suivant : ^Cellule
  FinEnreg
  Liste : ^Cellule
```

- **ListeVide :**

La 1ère fonction de base qu'on doit écrire pour pouvoir manipuler des données de type liste est la fonction ListeVide () qui prend en entrée une liste et retourne vrai si elle est vide et faux sinon.

```
Fonction  ListeVide (liste1 : Liste) : Booleen
Debut
 Si (liste1 = Nil) Alors
  Retourner (Vrai)
 Sinon
  Retourner (Faux)
 FinSi
Fin
```

- **PosFin :**

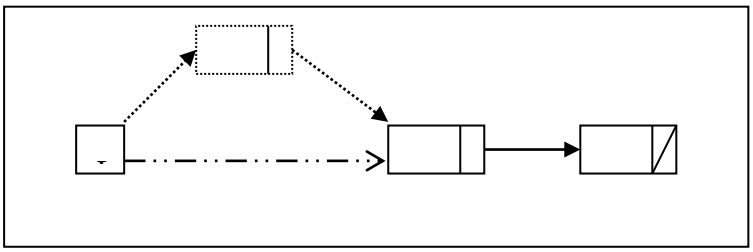
La fonction PosFin () prend en entrée une liste et retourne un pointeur vers le dernière élément de la liste.

```
Fonction  PosFin (liste1 : Liste) : Liste
Var listeAux : Liste
Debut
listeAux ← liste1
 Si (listeAux <> Nil) Alors
   Tantque (listeAux^.suivant <> Nil) Faire
     listeAux ← listeAux^.suivant
    Fintanque
 FinSi
 Retourner (listeAux)
Fin
```

- **AjouterTete :**

La fonction AjouterTete () prend en entrée une valeur à ajouter en tête d'une liste donnée et retourne un pointeur vers le début de la nouvelle liste.
Cette fonction doit créer une nouvelle variable dynamique pour contenir la nouvelle cellule, puis remplir le champ valeur et le champ suivant de cette cellule ensuite mettre l'adresse de cette cellule dans le pointeur vers la tête de liste et retourner cette valeur.

```
Fonction  AjouterTete (v : entier, L : Liste) : Liste
Var C : ^Cellule
Debut
 Allouer (C)
 C^.valeur ← v
 C^.suivant ← L
 L ← C
 Retourner (L)
Fin
```



- **Ajouter un élément à la fin de la liste**

La fonction **Ajouter_fin()** prend en entrée une valeur à ajouter à la fin d'une liste donnée et retourne un pointeur vers le début de la nouvelle liste.

```
procedure  ajouter_fin(liste1 : Liste, e : entier)
   liste2 : Liste;
debut
oSi (liste1= Nil) Alors        liste1 ← new couple(e, null)
sinon
 /* parcours de la liste vers la fin */
            Liste2 ← PosFin(liste1)
            /*ajout de l'élèment e à la fin de la liste */
            liste2.suivant ← new couple(e, null)
fisi        /* nous pouvons ajouter le retour en tête de la liste */
fin ajouter_fin;
```

- **langueur d'une Liste**

Cette fonction calcule le nombre des éléments de la liste. Elle prend comme argument à l'entrée une liste et retourne comme sortie un entier présentant la langueur de la liste.

```
function longueur(liste1 : Liste) : entier
            x : entier
            listeAux : Liste
début
            x ← 0
            listeAux ← liste1
            Tantque (listeAux^.suivant <> Nil) Faire
                        x ← x + 1
                        listeAux ← listeAux^.suivant
           Fintanque
           retourner x;
fin longueur
```

- **chercher un élément dans la liste**

Cette fonction permet de rechercher la position d'un élément dans une liste en supposant qu'il existe, sinon, la fonction retourne faux. Ses paramètres d'entrée sont alors la liste et l'élément recherché, et son paramètre de sortie étant un booleén.

```
function appartient(x : entier; liste1 : Liste) : booleén
  liste2 : Liste
début
        liste2 ← liste1
        tantque (liste2 != null && liste2^.valeur <>x)
        liste2 ← liste2^.suivant
        fin tantque
                si (liste2^.valeur <>x) alors
                retourner (vrai)
                sinon
                retourner (faux)
                fin si
fin appartient;
```

# Chapitre 4 : Les Piles et les Files

## Vue d'ensemble

Ce Chapitre présente la définition et l'utilisation des Piles et des files

### Objectifs

Ce chapitre a pour objectifs de permettre aux étudiants d'acquérir les connaissances de base se rapportant aux objectifs spécifiques suivants:

- Etre capable d'utiliser les piles et les files.
- Connaître une nouvelle structure de données, sa création et surtout les opérations de base sur une pile et une file
- Savoir écrire un algorithme en utilisant les structure de données piles et files.

### Pré-requis

- Algorithmique 1
- Les enregistrements
- Les pointeurs

### Durée

- 6 H

## Eléments de contenu

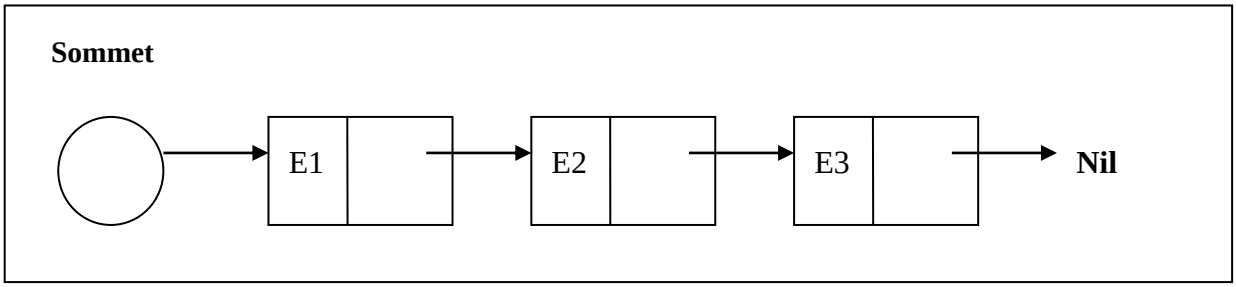
1. Introduction
2. Les piles
3. Les files

# 4.1 Les piles

## 4.1.1 Présentation

Une pile est une suite de cellules allouées dynamiquement (liste chaînée) où l'insertion et la suppression d'un élément se font toujours en tête de liste.

L'image intuitive d'une pile peut être donnée par une pile d'assiettes, ou une pile de dossiers à condition de supposer qu'on prend un seul élément à la fois (celui du sommet). On peut résumer les contraintes d'accès par le principe « dernier arrivé, premier sorti » qui se traduit en anglais par : **L**ast **I**n **F**irst **O**ut (figure 1).




La structuration d'un objet en pile s'impose lorsqu'on mémorise des informations qui devront être traitées dans l'ordre inverse de leur arrivée. C'est le cas, par exemple, de la gestion des adresses de retour dans l'exécution d'un programme récursif.

En supposant que les éléments de la pile sont des entiers, celle-ci se déclare de la façon suivante :

```
Types
Pile = ^Cellule
Cellule = Enreg
            Elem : Entier
            Suiv : Pile
          FinEnreg
Var
P : Pile
```

### 4.1.2 Manipulation d'une pile

D'un point de vue manipulation, les contraintes d'accès sont matérialisées par les procédures et les fonctions suivantes :

- **Procédure Initialiser (Var P : Pile)** : crée une pile vide P.

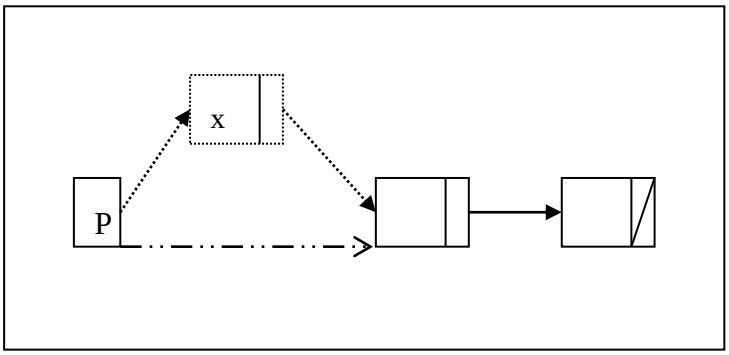
```
Procédure Initialiser (Var P : Pile)
Début
   P ← Nil
Fin
```

- **Fonction Pile_Vide (P : Pile) : Booléen** : renvoie la valeur vrai si la pile est vide.

```
Fonction Pile_Vide (P : Pile) : Booléen
Début
   Pile_Vide ← (P = Nil)
Fin
```

- **Procédure Empiler (x : Entier ; Var P : Pile)** : ajoute l'élément x au sommet de la pile.

```
Procédure Empiler (x : Entier ; Var P : Pile)
Variables
   Q : Pile
Début
   Allouer (Q)
   Q^.Elem ← x
   Q^.Suiv ← P
   P ← Q
Fin
```



- **Procédure Dépiler (Var x : Entier ; Var P : Pile)** : dépile le sommet de la pile et le met dans la variable x.

**Procédure** Dépiler (Var x : Entier ; Var P : Pile)

**Début**

**Si** NON (Pile_Vide(P)) **Alors**

x ← P^ .Elem

Q ← P

P ← P^ .Suiv

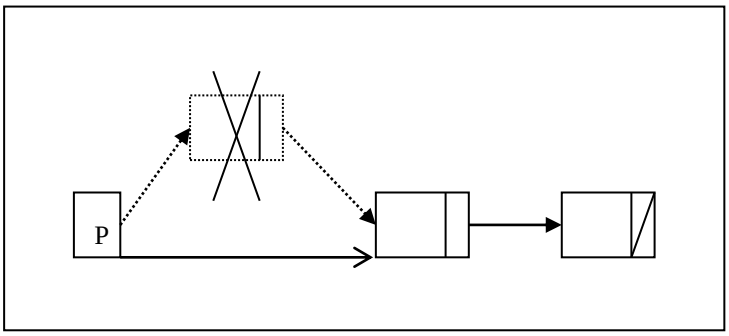
Libérer (Q)

**Sinon**

Ecrire (“Impossible, la pile est vide”)

**FinSi**

**Fin**



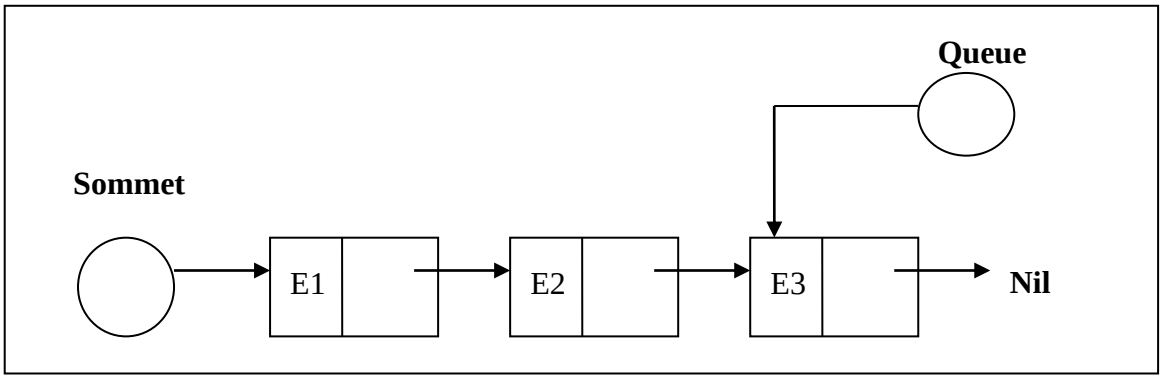

# 4.2 Les Files

## 4.2.1 Présentation

Une file est une suite de cellules allouées dynamiquement (liste chaînée) dont les contraintes d’accès sont définies comme suit :

- On ne peut ajouter un élément qu’en dernier rang de la suite.
- On ne peut supprimer que le premier élément.

L’image intuitive d’une file peut être donnée par la queue devant un guichet lorsqu’il n’y a pas de resquilleurs. On peut résumer les contraintes d’accès par le principe « premier arrivé, premier sorti » qui se traduit en anglais par : **F**irst **I**n **F**irst **O**ut.




La structuration d’un objet en file s’impose en particulier lorsqu’une ressource doit être partagée par plusieurs utilisateurs : il y a formation d’une *file d’attente.*

Un exemple est donné par le spooler d'impression qui est un programme qui reçoit, traite, planifie et distribue les documents à imprimer dans un système multiprogrammé.

En supposant que les éléments de la file sont des entiers, celle-ci se déclare de la façon suivante :

```
Types
        Liste = ^Cellule
        Cellule = Enreg
                    Elem :Entier
                    Suiv : Liste
                  FinEnreg
        File =  Enreg
                Tête : Liste
                Queue : Liste
                FinEnreg
Var
F : File
```

### 4.2.2 Manipulation d'une file

D'un point de vue manipulation, les contraintes d'accès sont matérialisées par les procédures et les fonctions suivantes :

- **Procédure Initialiser (Var F : File)** : crée une file vide F.

```
Procédure Initialiser (Var F : File)
Début
        F.Tête ← Nil
        F .Queue ← Nil
Fin
```

- **Procédure Ajouter (x : Entier ; Var F : File)** : ajoute l'élément x à la fin de la file.

```
Procédure Ajouter (x : Entier ; Var F : File)
Var
        P : Liste
Début
```

```
        Allouer (P)
        P^ .Elem ← x
        P^ .Suiv ← Nil
        Si (F .Queue <> Nil) Alors
                F .Queue^. Suiv ← P
      Sinon
                F.Tête ← P
        FinSi
        F .Queue ← P
Fin
```

→ Dans le cas où la file est vide, comme la queue, la tête de la file doit également pointer vers le nouvel élément.

- **Procédure Extraire (Var x : entier ; Var F : File)** : extrait le sommet de la file et le met dans la variable x.

```
Procédure Extraire (Var x : entier ; Var F : File)
Var
      P : Liste
Début
        Si (F.Tête = Nil) Alors
                Ecrire (« impossible, la file est vide »)
        Sinon
                P ← F.Tête
                x ← F.Tête^.Elem
                F.Tête ← F.Tête^. Suiv
                Libérer (P)
         FinSi
Fin
```

**Remarque**

Dans les langages qui n'offrent pas le type pointeur, les structures de pile ou file peuvent être implantées sous forme de tableaux.

Ainsi, une pile d'entiers peut être déclarée de la façon suivante :

**Constantes**

n = 100

**Types**

Indice : 1..n

Cellule = Enreg

Elem : entier

Suiv : Indice

FinEnreg

Pile = Tableau [Indice] de cellule

**Variables**

L : Pile

# Chapitre 5 : La Récursivité

## Vue d'ensemble

Cette leçon présente la définition et l'utilité de la récursivité.

### Objectifs

Ce chapitre a pour objectifs de permettre aux étudiants d'acquérir les connaissances de base se rapportant aux objectifs spécifiques suivants:

- Etre capable de transformer un schéma itératif simple en un schéma récursif.
- Savoir écrire un algorithme récursif.

### Pré-requis

- Algorithmique 1

### Durée

- 6 H

## Eléments de contenu

1. Rappel
2. Algorithmes récursives

# 5.1 Rappel

## 5.1.1 Les procédures

### 5.1.1.1 Définition

Une procédure est un sous algorithme réalisant un traitement sur une partie des données d'un algorithme principal. Elle permet de décomposer un algorithme en sous algorithme ou module plus simple et donc de simplifier la lecture et le suivi d'un algorithme. Une procédure est définie avant le code de l'algorithme principal puis elle est appelée par son nom suivi de ses paramètres s'ils existent.

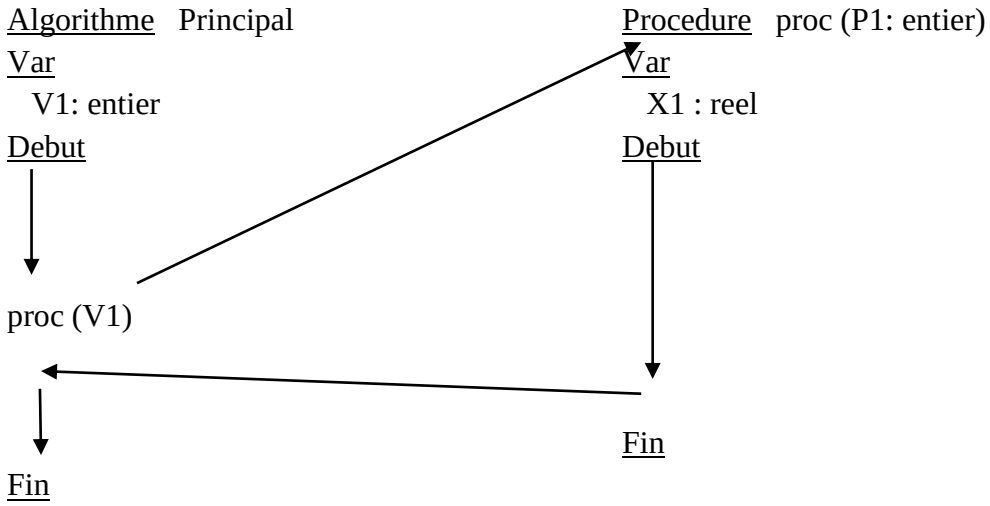
***Syntaxe***

| | |
|---|---|
| Procedure nom (P1: type, …, Pn : type)<br>Var<br>{Variable}<br>Debut<br><Traitement><br>Fin | Algorithme Principal<br>Var<br>{Variable}<br>Debut<br>…<br>nom (P1, …, Pn)<br>…<br>Fin |

Remarque :
- nom est le nom de la procédure.
- type est un type de donnée simple ou composé.
- P1 … Pn sont les paramètres de la procédure.
- Une procédure peut être appelée à partir de l'algorithme principal ou d'une autre procédure.

*Schéma d'exécution*

L'appel de la procédure se fait en utilisant son nom suivi de ses paramètres séparés par des virgules et entourés par des parenthèses. Quand on appelle une procédure le contrôle se trouve automatiquement transféré au début de la procédure. Quand toutes les instructions de la procédure ont été exécutées le contrôle retourne à l'instruction qui suit immédiatement l'instruction d'appel de la procédure.




### 5.1.1.2 Les variables et les paramètres

- ***Variable locale***

Une variable est dite locale si elle est définie dans la partie de déclaration des variables propre à la procédure. Elle n'est accessible que dans la procédure où elle a été définie. Dans l'exemple précédent X1 est une variable locale à la procédure proc.

- ***Variable globale***

Une variable est dite globale si elle est définie au niveau de l'algorithme principal qui appelle la procédure c'est à dire une variable utilisée par la procédure et qui n'est pas déclaré dans cette procédure. Une variable globale peut être utilisé n'importe où, à l'intérieur ou à l'extérieur d'une procédure on parle donc de visibilité et on dit qu'elle est visible par l'algorithme principal et par toutes les autres procédures utilisées. Dans l'exemple précédent V1 est une variable globale.

L'échange d'information entre la procédure et l'algorithme peut se faire à travers les variables globales. Cette méthode peut changer le contenu de la variable à l'intérieur de la procédure ce qui peut affecter l'exécution de l'algorithme principal. Pour résoudre ce problème on fait recours à l'emploie des paramètres qui offrent une meilleur approche pour l'échange d'information entre la procédure et le l'algorithme principal. Le transfert d'une donnée se fait entre un paramètre effectif et un paramètre formel.

- ***Paramètre formel***

Un paramètre formel est un paramètre défini dans la déclaration de la procédure c'est-à-dire dans l'entête de procédure. Dans l'exemple précédent P1 est un paramètre formel à la procédure proc.

- ***Paramètre effectif***

Un paramètre effectif est un paramètre utilisé pendant l'appel de la procédure. Dans l'exemple précédent V1 est un paramètre effectif pour la procédure proc. Lors de l'appel de la procédure et avant de commencer son exécution les paramètres formels sont initialisés par les valeurs des paramètres effectifs. Ainsi dans l'exemple précédent P1 est initialisé avec la valeur de V1.

*Exemple*

| | |
|---|---|
| Procedure **max** (x : entier, y : entier)<br>Var<br>m : entier<br>Debut<br>Si x>y Alors<br>m ← x<br>Sinon<br>m ← y<br>FinSi<br>Ecrire ("le max est :", m)<br>Fin | Algorithme Maximum<br>Var<br>A, B : entier<br>Debut<br>Lire (A, B)<br>**max** (A, B)<br>Fin |

On désigne par
- A, B: 2 paramètres effectifs
- x, y: 2 paramètre formels
- A, B : variables globales
- m: variable locale

### 5.1.1.3 Passage des paramètres

- ***Passage des paramètres par valeur***

Le mode de transfert de paramètre entre l'algorithme appelant et la procédure appelé définit un moyen d'échange de donnée entre eux. Dans le passage des paramètres par valeur qui représente le 1er mode, les paramètres formels de la procédure reçoivent les **valeurs** des paramètres effectifs, ainsi avant l'exécution de la procédure, ses paramètres formels sont initialisés avec les valeurs des paramètres effectifs. Tous changement des paramètres formels dans la procédure ne change pas les valeurs des paramètres effectifs associés. Dans l'exemple précédent le mode de passage des paramètres pour la procédure max est **par valeur**, les paramètres formels sont x et y, ils sont initialisés par les valeurs de A et B.

- ***Passage des paramètres par variable***

Contrairement au mode de passage des paramètres par valeur, dans le mode de passage des paramètres par variable toutes modification dans la valeur du paramètre formel passé par variable, implique une modification de la valeur du paramètre effectif correspondant. Ainsi ce

2ème mode est plus général que le 1er il prend les caractéristiques du 1er et ajoute la possibilité de modifier les variables de l'algorithme appelant à partir de la procédure.

Pour définir un passage de paramètre par variable on ajoute le mot clé « **var** » dans la définition des paramètres de la procédure avant les paramètres pour lesquels on veut définir ce mode.

Un paramètre passé par variable ne peut être qu'une variable il ne peut pas être une constante ou une expression.

*Exemple*

```
Procedure   max (x : entier, var y : entier)
Var
  m : entier
Debut
  Si x>y Alors
    m ← x
  Sinon
    m ← y
  FinSi
  Ecrire ("le max est :", m)
  x ← 0
  y ← 0
Fin
```

```
Algorithme     Maximum
Var
   A, B : entier
Debut
   Lire (A, B) {2,4} {ceci est un
commentaire}
   max (A, B)
   Ecrire (A, B)
   Ecrire (x, y)
Fin
```

## 5.1.2 Les fonctions

### 5.1.2.1 Définition

La fonction possède la même définition que la procédure mais avec seulement 2 différences. D'abord une fonction possède une valeur de retour. Ensuite la fonction est utilisée dans l'algorithme principal comme une variable contrairement à la procédure qui est utilisé comme une instruction.

*Syntaxe*

```
Fonction   nom_fonction (P1: type, …, Pn :
type) : type
Var
   {Variable}
Debut
     <Traitement>
  Retourner (val_retour)
Fin
```

```
Algorithme  Principal
Var
   vf : type
   …
Debut
…
vf  ← nom_fonction (P1, …, Pn)
…
Fin
```

**Remarque :** Le type de la fonction doit être le même que le type de la variable vf utilisé dans l'algorithme principal pour recevoir la valeur de retour de la fonction.

*Exemple*

| | |
|---|---|
| Fonction **max** (x : entier, y : entier) : entier<br>Var<br><br>Debut<br>Si x>y Alors<br>Retourner (x)<br>Sinon<br>Retourner (y)<br>FinSi<br>Fin | Algorithme Maximum<br>Var<br>A, B, M : entier<br>Debut<br>Lire (A, B) {2,4}<br>M ← **max** (A, B)<br>Ecrire ("le max est :", M)<br>Fin |

## 5.2 Algorithmes récursifs

### 5.2.1 Définitions récursives

Une définition récursive est définie en fonction d'un objet ou d'une action de même nature mais de complexité moindre.

*Exemple*

On peut définir le factoriel d'un nombre « **n** » positif de deux manières:

- définition non récursive (ou itérative):

  n ! = n * n-1 * ... 2 * 1

- définition récursive:

$$\begin{cases} n ! = n * (n-1) ! \\ \text{et } 0 ! = 1 \end{cases}$$

Une définition récursive est composée de deux partie: une partie strictement récursive et une partie non récursive (base) servant de point de départ à l'utilisation de la définition récursive.

## 5.2.2 Fonctions récursives

### 5.2.2.1 Définition

Comme pour les définitions récursives, les fonctions récursives sont des fonctions qui sont appelées depuis leur propre corps de fonction, soit directement soit indirectement, à travers une ou plusieurs fonctions relais. Si la fonction P appelle directement P, on dit que la récursivité est directe. Si P appelle une fonction P1, qui appelle une fonction P2 et qui enfin appelle P, on dit qu'il s'agit d'une récursivité indirecte.

Si la fonction P appelle dans son propre corps la fonction P, il est logique de penser que l'exécution ne s'arrêtera jamais. Il est donc primordial qu'une des branches de la fonction P permette de stopper la récursivité.

*Exemple*

On veut calculer la somme des n premiers entiers positifs. Une définition récursive de cette somme serait:

$$\begin{cases} \text{Somme } (n) = n + \text{somme } (n-1) \\ \text{Et somme}(1) = 1 \end{cases}$$

Ce qui se traduit par la fonction récursive suivante:

```
Fonction  somme (n : entier) : entier
Var
  m : entier
Debut
  Si (n = 1)  alors
    somme ← 1
  Sinon
    m ← somme (n-1)
    somme ← n + m
  FinSi
Fin
```

Il est clair que sans le test « Si (n = 1) » cette fonction ne s'arrêterait jamais.

### 5.2.2.2 Exécution d'une fonction récursive

Supposant qu'on appelle la fonction somme (4). Puisque $4 \neq 1$, cette fonction va s'appeler elle-même avec la valeur 3 (somme (3)). Ce nouvel appel se termine en renvoyant une valeur

(dans cet exemple la valeur retournée est 6 = 3 + 2 + 1). Cette valeur sera ajoutée à la valeur 4 et l'appel de somme (4) retournera la valeur 10 (4 + 6).
L'appel somme (3) suivra le même processus et appellera somme (2) qui lui-même appellera à somme (1) qui retourne 1.

**somme**(4)

**somme**(3)

**somme**(2)

**somme**(1)

retourne 1

retourne 2 + 1

retourne 3 + 3

**retourne 4 + 6**

### 5.2.2.3 Variables locales et paramètres des fonctions récursives

Lorsqu'une fonction récursive définit des variables locales, un exemplaire de chacune d'entre elles est créé à chaque appel récursif de la fonction.
Dans notre exemple somme (4), la variable locale m est créée 4 fois. Ces variables sont détruites au fur et à mesure que l'on quitte la fonction comme toute variable locale d'une fonction.

Il en est de même des paramètres des fonctions.

somme(4)

création du 1er m

somme(3)

création du 2ème m

somme(2)

création du 3ème m

somme(1)

création du 4ème m

destruction de m

retourne 1

destruction de m

retourne 2 + 1

destruction de m

retourne 3 + 3

destruction de m

retourne 4 + 6

## 5.3 Exercices d'application

**Exercice 1 :**

Ecrire une fonction récursive qui permet de calculer le produit de deux entiers k et x selon la formule suivante : K*x=x+ (k-1)*x

On prendra soin de trouver les cas particuliers (pour que la récursivité s'arrête)

**Exercice2:**

Pour calculer le PGCD de deux entiers positifs a et b, nous allons utiliser l'algorithme d'Euclide.

Celui-ci est résumé comme suit :

Si l'un des entiers est nul, alors le PGCD est égal à l'autre.

Sinon le PGCD de a et b est égale au PGCD de a et de b-a (sous l'hypothèse que b>=a)

Ecrire la fonction récursive correspondante.

**Exercice 3 :**

1. Nous cherchons à réaliser une fonction qui à partir de deux entiers positifs a et b, renvoie le quotient de la division de a par b .On cherche à réaliser cela sans utiliser les opérateurs DIV ou MOD,

Ecrire le programme de cette division sous forme récursive.

2. Même question, mais pour le reste de la division de a par b

**Exercice 4 :**

Ecrire une fonction récursive qui prend en entrée un entier positif N et retourne la valeur de N factoriel (N !), en utilisant la définition récursive suivante :

$$\begin{cases} N ! = N * (N-1) ! \\ 0 ! = 1 \end{cases}$$

**Exercice 5 :**

Ecrire une fonction récursive qui permet de calculer les n termes de la suite de Fibonacci sachant que :

$$\begin{cases} F(0) = 1 \\ F(1) = 1 \\ \text{Pour tout } N \geq 2 : F(N) = F(N-1) + F(N-2) \end{cases}$$

**Exercice 6 :**

On désire réaliser les tâches suivantes :

1- Saisir un entier N (0 <N <20).

2- remplir un tableau T par N entiers positifs.

3- Inverser le tableau T.

4- Afficher le nouveau tableau T.

Exemple :

Si on donne N=7 et T

1 10 9 6 11 3 8

Après inversion on obtient

8 3 11 6 9 10 1

Ecrire le programme principal et les sous programmes nécessaires pour réaliser ces tâches.

Les algorithmes de la tâche 1,2 et 4 doivent être itératifs et celui de la tache 3 récursif.

## 5.4 Solutions des exercices

**Exercice1 :**

```
Fonction produit(k :entier,x :entier) :entier
var
p :entier
debut
Si (k=1) alors
        P←x
sinon
        si(k=0) alors
                p←0
        sinon
                p←x+produit(k-1,x)
        FinSi
FinSi
returner(p)
Fin
```

**Exercice 2**

```
Foncion  pgcd(a :entier,b :entier) :entier
var
        r :entier
debut
        Si ((a=0) ou (b=0))alors
                R←a+b;
        sinon

                Si (b>=a) alors
                        R←pgcd(a,b-a);
                sinon
                        r←pgcd(b,a-b);
                FinSi

        FinSi
          returner(r)
Fin
```

**Exercice 3**

```
Fonction quotient(a :entier,b :entier) :entier
var
        q :entier
debut
        si (a<b) alors
                q←0;
        sinon
                q←1+quotient(a-b,b);
        finSi
          returner(q)
Fin
Fonction reste(a :entier,b :entier) :entier
var
        r :entier
```

```
debut
        si (a<b)alors
                r←a;
        sinon
                r←reste(a-b,b)
        finSi
        returner(r)
Fin
```

**Exercice 4**

```
Fonction Factoriel(n :entier):entier

var
F: entier
Debut
        Si(n=0) Alors
                Retourner(1)
        Sinon
                F ← Factoriel(n-1)
                Retourner(n*F)
        FinSi
Fin
```

**Exercice 5**

```
Fonction Fibonacci(n :entier):entier
var
F1,F2: entier
Debut
        Si(n=0 ou n=1) Alors
                Retourner(1)
        Sinon
                F1 ← Fibonacci(n-1)
```

```
            F2 ← Fibonacci(n-2)
            Retourner(F1+F2)
        FinSi
Fin
```

## Exercice 6

```
Procedure saisir(var n:entier)

debut
ecrire("taper un entier entre 1..20")
lire(n)

Si (n>20) ou (n<0)
        alors saisir(n)
fin

procedure remplir(var a: tableau[1..20] de entier;n:entier)
var i:entier

debut

lire(a[n])

Si a[n]<0 Alors
        remplir(a,n)
sinon si n>1 alors
        remplir(a,n-1)
finSi
fin

procedure affiche(a: tableau[1..20] de entier;n:entier)
var i:entier

debut

ecrire(a[n])

Si n>1 Alors
        affiche(a,n-1)
```

```
finSi
fin

procedure inverser(var t: tableau[1..20] de entier;d,f:entier)
var aux:entier

debut
Si d <f alors
        aux←t[d]
        t[d]←t[f]
        t[f]←aux
        inverser(t,d+1,f-1)
FinSi
Fin

//Algorithme principlal
Algorithme inversttrecursif
var
        t: tableau[1..20] de entier
        n:entier
debut

saisir(n)
remplir(t,n)
affiche(t,n)
inverser(t,1,n)
affiche(t,n)
Fin
```

# Chapitre 6 : Les arbres et les graphes

## Vue d'ensemble

Ce Chapitre présente la définition et l'utilisation des  arbres et des graphes

### Objectifs

Ce chapitre a pour objectifs de permettre aux étudiants d'acquérir les connaissances de base se rapportant aux objectifs spécifiques suivants:

- Etre capable d'utiliser les arbres et les graphes.
- Connaître une nouvelle structure de données, sa création et surtout les opérations de base sur un arbre et un graphe
- Savoir écrire un algorithme en utilisant se rapportant à des structures dynamiques de données arbres et graphes.

### Pré-requis

- Algorithmique 1
- Les enregistrements
- Les pointeurs
- La récursivité

### Durée

- 6 H

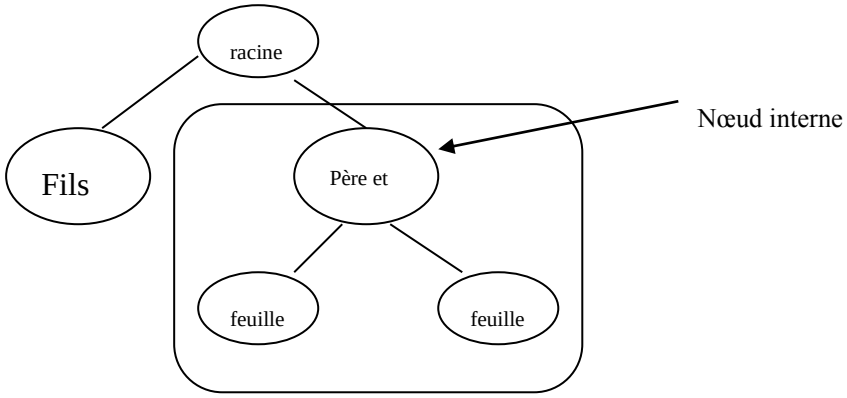
## Eléments de contenu

1. Les arbres
2. Les graphes

# 6.1Les Arbres

## 6.1.1 Définition

Un arbre est un ensemble de nœuds avec une relation de hiérarchie entre Père et Fils




**Remarque :**

- Un **arbre** est :

  - soit un arbre vide

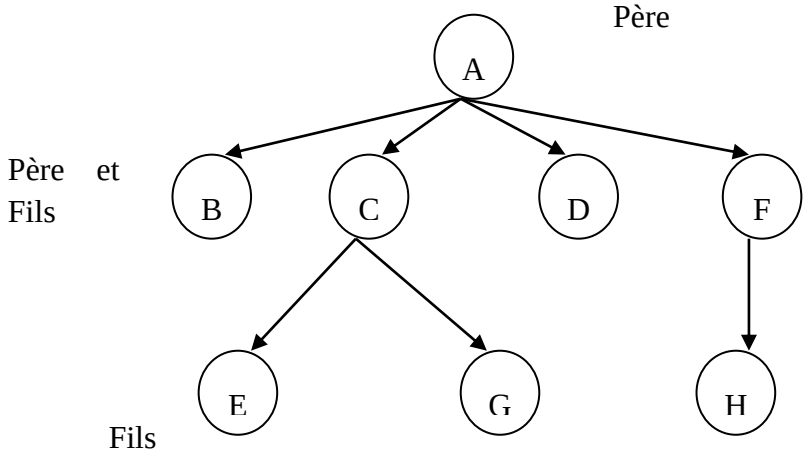
  - soit un nœud racine et un ensemble de (sous)-arbres T1 ,…, Tn de sorte à ce que la racine de Ti est connecté par un lien direct à **la racine.**

- la **racine** est un nœud sans père

- la **feuille** est un nœud sans fils

- le **nœud interne** est un nœud admettant au moins un fils

*Exemple*



**Opérations sur les arbres**

Il existe un certain nombre d'opérations classiques sur les arbres :

- Tester si l'arbre est vide ou non
- Créer un arbre ou une feuille
- Supprimer un nœud ou la totalité de l'arbre
- Accéder à une valeur qui se trouve dans la racine.

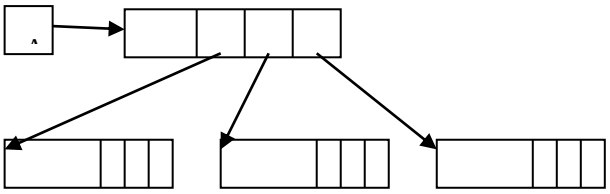
## 6.1.2 Implémentation

Un arbre peut être représenté par un ensemble de nœuds, chaque nœud contient une valeur et un tableau de N case qui contient des pointeurs pour les N fils du nœud. **N** représente le nombre maximal de fils que peut avoir un nœud dans l'arbre.

Ainsi il se définit comme suit :

```
Type
  Nœud = Enreg
       valeur : type
       fils : Tableau [1..N] de ^Nœud
  FinEnreg
  Arbre = ^Nœud
Var
  A : Arbre
```

La figure ci-dessous représente un arbre avec N= 3, donc chaque nœud doit avoir au maximum 3 fils.



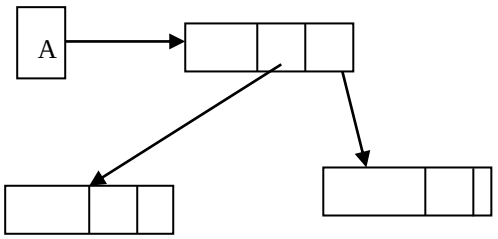
## 6.1.3 Les fonctions d'accès

Nous allons étudier un cas particulier des arbres : les **arbres binaires.** Un arbre binaire est un arbre où tout nœud a zéro ou deux fils.

La représentation de la structure de données arbre binaire est constituée de 2 parties une pour représenter un nœud qui contient une valeur, un pointeur vers le nœud gauche appelé aussi

fils gauche et un pointeur vers le nœud droit ou le fils droit ; et une autre partie qui constitue la représentation de l'arbre sous forme d'un pointeur vers un nœud qui est la racine. On travaille sur les valeurs de type entier.

```
Type
  Nœud = enreg
       valeur          : entier
       fils_gauche     : ^Nœud
       fils_droit      : ^Nœud
  FinEnreg
  Arbre_binaire = ^Nœud
```




- **ArbreBinVide**

Teste si un arbre binaire est vide ou non.

```
Fonction  ArbreBinVide (A : Arbre_binaire) : Booleen
Debut
 Si (A = Nil) Alors
  Retourner (Vrai)
 Sinon
  Retourner (Faux)
 FinSi
Fin
```

- **CreerFeuille**

Cette fonction crée un nœud contenant une valeur et retourne son adresse en mémoire.

**Fonction** CreerFeuille (v : entier) : Arbre_binaire

**Var**

F : Arbre_binaire

**Debut**

Allouer (F)

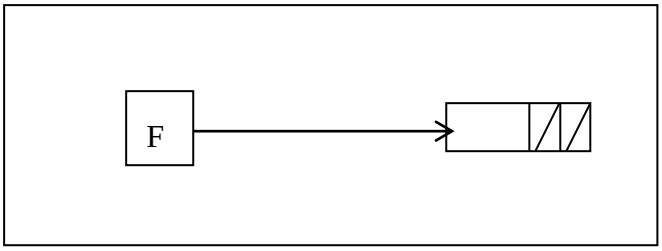
F^.valeur ← v

F^.fils_gauche ← Nil

F^.fils_droit ← Nil

Retourner (F)

**Fin**




- **CreerArbreBin**

Cette fonction crée un arbre binaire à partir d'un sous arbre droit et un sous arbre gauche existants et d'une valeur à mettre dans la racine enfin elle retourne l'adresse de la racine.

**Fonction** CreerArbreBin (v : entier, fg : Arbre_binaire, fd : Arbre_binaire) : Arbre_binaire

**Var**

R : Arbre_binaire

**Debut**

Allouer (R)

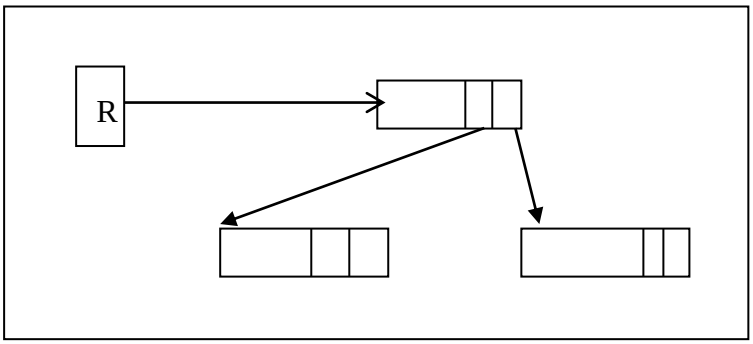
R^.valeur ← v

R^.fils_gauche ← fg

R^.fils_droit ← fd

Retourner (R)

**Fin**



- **SupprimeArbreBin**

Cette procédure permet de supprimer tous les nœuds d'un arbre binaire dont l'adresse de sa racine est fournie en paramètre.

```
Procedure  SupprimeArbreBin (A : Arbre_binaire)
Debut
 Si ( ArbreBinVide(A) = Faux ) Alors
    SupprimeArbreBin (A^.fils_gauche)
    SupprimeArbreBin (A^.fils_droit)
    Liberer(A)
 FinSi
Fin
```

- **Accès à la racine**

Cette fonction récupère la valeur contenue dans le nœud racine

```
Fonction Racine (A :Arbre_Binaire) : entier
Debut
    Si (A<>Nil)
          Retourner (A^.valeur)
   Sinon
       Ecrire(« Erreur »)
  Fin Si
Fin
```

## 6.1.4 Niveau et hauteur d'un arbre :

- Le niveau d'un nœud v est le nombre de ses ancêtres

  - si v est la racine alors son niveau est 0

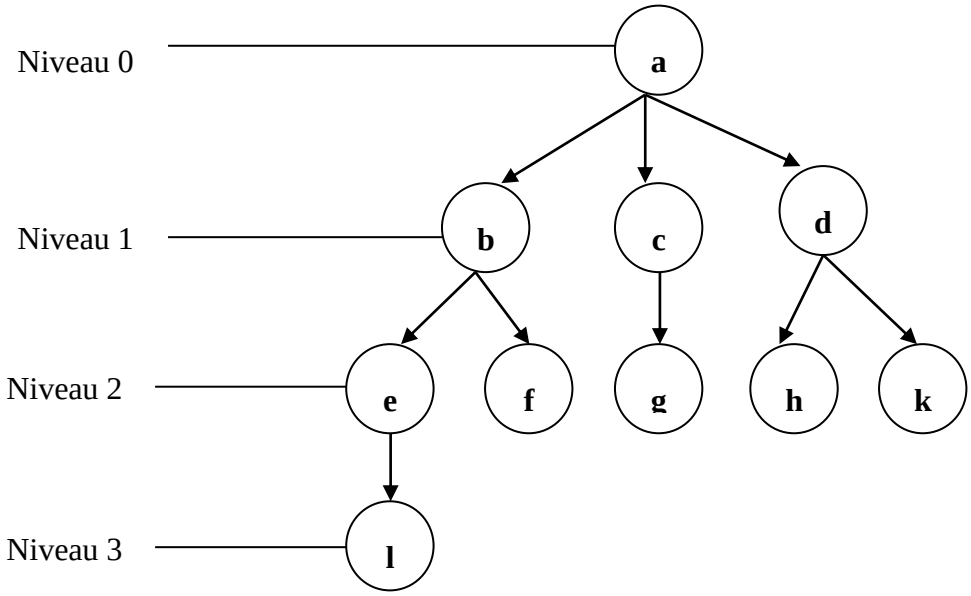
  - sinon le niveau de v est : 1+ niveau du père de v

- La hauteur d'un nœud v est définie par la définition suivante :

- si v est un nœud feuille alors sa hauteur est 0

- sinon la hauteur de v est : 1+ maximum des hauteurs de ses fils

- La hauteur d'un arbre est la hauteur de sa racine




- Nœud « b » :

- niveau=1

- hauteur=2

- hauteur de l'arbre=3

## 6.1.5 Parcours d'un arbre

### 6.1.5.1 Types de parcours

- Parcourir un arbre revient à visiter ses nœuds

- Visiter un nœud revient à effectuer un certain traitement sur ce nœud qui dépend de l'application (affichage du contenu, modification d'un attribut, calcul d'une valeur, …)

- Il existe 2 types de parcours :

  - Le parcours en profondeur qui explore l'arbre branche par branche et peut être préfixé, infixé ou post fixé :
    - **Parcours préfixé** : on parcourt la racine puis le sous arbre gauche enfin le sous arbre droit.
    - **Parcours infixé**: on parcourt le sous arbre gauche puis la racine enfin le sous arbre droit.

- **Parcours postfixé**: on parcourt le sous arbre gauche puis le sous arbre droit enfin la racine.

- Le parcours en largeur qui explore l'arbre niveau par niveau en commence par la racine.

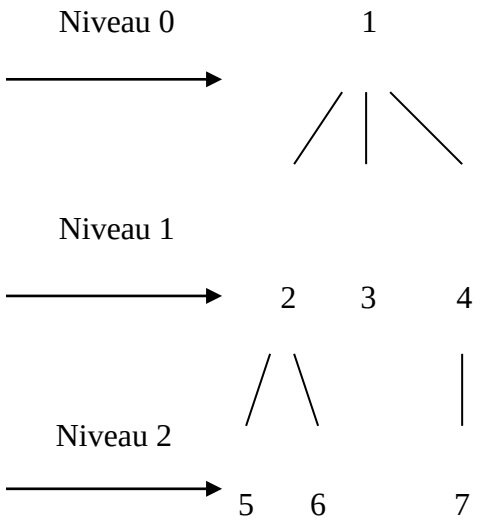
- ***Parcours Infixé***

Soit un arbre binaire dont les valeurs sont des entiers on veut écrire une procédure récursive qui fait un parcours infixé de cet arbre pour afficher les valeurs qui se trouvent dans tous les nœuds.

```
Procedure  Infixe (A : Arbre_binaire)
Var
Debut
 Si ( ArbreBinVide(A) = Faux ) Alors
   Infixe (A^.fils_gauche)
   Ecrire (A^.valeur)
   Infixe (A^.fils_droit)
 FinSi
Fin
```

*Exemple*

Si on prend l'arbre suivant :



- Le parcours préfixé donne : (1,2,5,6,3,4,7)

- Le parcours postfixé donne : (5,6,2,1,3,7,4,1)

- Le parcours infixé donne : (5,2,6,1,3,7,4)

- Le parcours en largeur donne : (1,2,3,4,5,6,7)

# 6.2 Les graphes

## 6.2.1 Définition

- Les graphes sont des structures de données qui permettent de représenter un grand nombre de données : des réseaux, des plans, des trajets, etc.
- Un graphe orienté est un couple G=(S, A) où :
    - S représente un ensemble de sommets,
    - A représente un ensemble d'arcs.

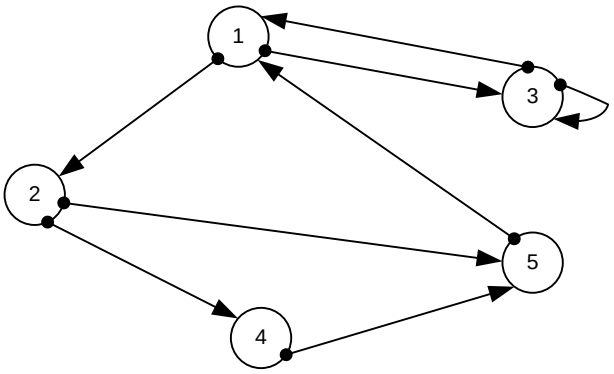
Soit a ∈ A, a= (u, v)

- **u** est appelé **origine** de a,
- **v** est appelé **extrémité** de a,
- u=prédécesseur (v)
- v=successeur (u)
- l'arc **a** est noté u→v

*Exemple*

S= {1, 2, 3, 4,5}

A= {(1,2), (1,3), (2,4), (2,5), (3,1), (3,3), (4,5), (5,1)}



## 6.2.2 Implémentation :

Pour implémenter un graphe, nous allons utiliser le mode de représentation par matrice d'adjacence.

Dans un graphe, il faux pouvoir bénéficier des fonctionnalités suivantes :

- créer un graphe
- initialiser un graphe
- ajouter un arc dans un graphe
- supprimer un arc dans un graphe
- ajouter un sommet dans le graphe
- trouver le successeur d'un sommet du graphe

- **Structure de données**

Pour implémenter la structure de données graphe selon la représentation matricielle, nous avons besoin de :

- un tableau à deux dimensions pour définir le graphe
- un type de données sommet, dans ce cas un sommet est un entier (indice de ligne ou de colonne de la matrice)
- un arc défini par deux sommets : une origine et une extrémité.

```
Type
Sommet=entier
Arc = Enreg
    Origine : Sommet
    Extrémité : Sommet
FinEnreg
Graphe =Enreg
    T : Tableau [1..Max][1..Max] de Sommet
    NS : entier
 FinEnreg
```

Avec NS=Nombre de sommets dans les graphes.

Max : nombre maximal pour le nombre de sommets du graphe

- **Création du graphe**

Pour créer un graphe, revient à initialiser le nombre de sommet.

```
Fonction CreerGraphe(n :entier) :Graphe
Var G :Graphe
Debut
    G.NS← n
    Allouer (G.T, n*n)
    Retourner (G)
Fin
```

- **Initialisation du graphe**

Cette fonction permet d'initialiser les cases de la matrice à 0, le graphe une fois crée est initialisé, ensuite les arcs sont ajoutés au fur et à mesure des besoins de l'application.

```
Fonction InitialiserGraphe(G :Graphe) :Graphe
Var i, j : entier
Debut
   Pour i de 1 à G.NS faire
          Pour j de 1 à G.NS faire
               G.T[i,j]←0
          Fin Pour
   Fin Pour
   Retourner (G)
Fin
```

- **Ajout d'un arc**

Permet d'attacher un arc préalablement crée au graphe

```
Fonction AjouterArc (G :Graphe, a :Arc) :Graphe
Debut
     Si (a.origine >G.NS ou a.extrémité>G.NS ou  a.origine <1 ou a.extrémité<1) alors
          Ecrire (« Arc non défini »)
     Sinon
          G.T [a.origine , a.extrémité]← 1
     Fin Si
     Retourner (G)
Fin
```

- **Suppression d'un arc**

Permet de supprimer un arc existant du graphe

```
Fonction SupprimerArc (G :Graphe, a :Arc) :Graphe
Debut
    Si (a.origine >G.NS ou a.extrémité>G.NS ou  a.origine <1 ou a.extrémité<1) alors
        Ecrire (« Arc non défini »)
    Sinon
        G.T [a.origine, a.extrémité] ← 0
    Fin Si
    Retourner (G)
Fin
```

- **Ajout d'un sommet**

Cette fonction permet d'ajouter un sommet au graphe dans la limite de la taille prévue, et initialise les arcs du nouveau sommet à 0

```
Fonction AjouterSommet (G : Graphe, s : Sommet) :Graphe
Debut
    Si (s>Max) alors
        Ecrire (« Arc non défini »)
    Sinon
        G.NS=G.NS+1
        Pour i de 1 à G.NS faire
            G.T [s, i] ← 0
            G.T [i, s] ← 0
        Fin pour
    Fin Si
    Retourner (G)
Fin
```

- **Successeur d'un sommet**

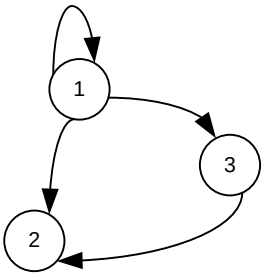
Recherche le premier successeur d'un sommet du graphe. Cette fonction retourne le premier successeur du sommet dans le graphe, et la valeur -1 s'il n'existe pas de successeur

```
Fonction Successeur (G : Graphe, s : Sommet) : Sommet
Var  Trouve : booléen
succ: Sommet
```

```
début
   si (s<1) ou (S>G.NS) alors
        Ecrire (« Sommet inexistant »)
   Sinon
      Trouve← faux
      Succ← 1
      Tant que (succ<G.NS) et (trouve<>vrai) faire
         Si (G.T[s,succ]=1) alors
               Trouve←vrai
         Sinon
               Succ←succ+1
         Fin Si
      Fin Tant que
      Si (trouve=vrai) alors
          Retourner (succ)
      Sinon
          Retourner (-1)
      Fin Si
Fin
```

*Exemple*

Nous allons créer le graphe suivant :




```
Algorithme Principal
Type
Sommet=entier
Arc = Enreg
    Origine : Sommet
    Extrémité : Sommet
FinEnreg
Graphe =Enreg
```

```
    T : Tableau [1..Max][1..Max] de Sommet
    NS : entier
 FinEnreg
Var
G1 : Graphe
A : Arc
Début
G1←CreerGraphe(3)
G1←InitialisationGraphe(G1)
//1ère arc
A.origine ← 1
A.extrémité ← 1
G1← AjouterArc(G1,A) ;

//2ème arc
A.origine ← 1
A.extrémité ← 2
G1← AjouterArc(G1,A) ;

//3ème arc
A.origine ← 1
A.extrémité ← 3
G1← AjouterArc(G1,A) ;

//4ème arc
A.origine ← 3
A.extrémité ← 2
G1← AjouterArc(G1,A) ;
Fin
```

# Chapitre 7 : Les algorithmes récursifs de recherche

## Vue d'ensemble

Ce Chapitre présente les algorithmes de recherche récursifs.

### Objectifs

Ce chapitre a pour objectifs de permettre aux étudiants d'acquérir les connaissances de base se rapportant aux objectifs spécifiques suivants:

- Etre capable d'utiliser les algorithmes de recherches récursifs sur des structures des données dynamique (liste, pile, file, arbre, graphe).
- Savoir écrire un algorithme de recherche sur une structure de données d'une façon récursive

### Pré-requis

- Algorithmique 1
- Les enregistrements
- Les pointeurs
- La récursivité
- Les listes
- Les arbres et les graphes

### Durée

- 6 H

## Eléments de contenu

1. Définition
2. Recherche dans une liste chaînée
3. Recherche dans un arbre binaire

# 7.1 Définition

Un **algorithme de recherche** est un type d'algorithme qui cherche, parmi une liste de données, les éléments satisfaisant certains critères.
En général, le critère de recherche porte sur la valeur d'une clé (ex : chercher un numéro de tel dans un ensemble de personnes (une personne=nom + adresse + tel...), rechercher un mot dans un dictionnaire (dictionnaire=mots+définitions+photo+...), ...). Cette clé peut aussi être simplement la valeur de l'objet (recherche d'un entier dans une liste d'entier, ...). Les algorithmes de recherche que nous allons présenter retournent un élément quelconque ayant une clé de la valeur cherchée. Il est parfois plus utile de retourner la place de cet élément dans l'ensemble des données (pour par exemple effectuer ensuite une suppression, ...).

# 7.2 Recherche dans une liste chainée

## 7.2.1 Recherche séquentielle

La recherche séquentielle est une méthode de recherche qui consiste à chercher un élément dans une liste d'éléments de même type. On commence du premier élément de la liste jusqu'à ce qu'on trouve l'élément. Si on ne le trouve pas, alors on aura parcouru toute la liste. Si on teste souvent une liste et si ces éléments sont répartis de manière aléatoire, le nombre moyen d'éléments à comparer pour savoir si un élément est dans la liste ou non est de : N/2 où N est le nombre d'éléments de la liste.

## 7.2.2 Recherche par valeur

La fonction recherche par valeur retourne un pointeur vers le premier élément trouvé dont la valeur est passée en paramètre, ou le pointeur *nil* si aucun élément n'a été trouvé

```
Fonction recherche_valeur (L: Liste, val:entier): Liste
Début
si (L=nil ) alors
   Retourner (nil)
sinon
   si (L^.valeur = val) alors
         Retourner(L)
   sinon
          Retourner recherche_valeur (L^.suivant,val)
   finsi
finsi
```

```
fin
```

### 7.2.3 Recherche par position

La fonction recherche par position retourne un pointeur vers l'élément ayant la position passée en paramètre. Si la liste contient moins d'éléments que le paramètre position, le pointeur nil est retourné.

```
Fonction recherche_postion (L: Liste, pos:entier): Liste
Début
si (L=nil ou pos=1) alors
   Retourner(L)
sinon
   Retourner recherche_position (L^.suivant , pos-1)
finsi
fin
```

### 7.2.4 Recherche dichotomique

La dichotomie consiste à diviser une liste de données en deux sous-listes. La recherche dichotomique consiste à décomposer une liste en deux sous-listes A et B puis à déterminer si l'élément recherché se trouve en A ou B. Puis on redécoupe en deux sous-listes C et D, la sous-liste où se trouve l'élément puis on détermine si l'élément recherché se trouve en C ou D. On répète l'opération tant qu'on n'a pas trouvé l'élément ou que les sous-listes restantes ne soient composées plus que d'un seul élément, puisqu'on n'a pas décomposé une liste d'un seul élément. Dans ce cas, on considère que l'élément n'est pas dans la liste recherchée.

La méthode par dichotomie est une des méthodes de recherche la plus rapide, le problème essentiel de la recherche dichotomique est la nécessité que les éléments de la liste soient triés (croissant ou décroissant).

### 7.2.5 Recherche dans une liste ordonnée

- **Principe :**

On regarde l'élément au milieu de la liste :

- S'il est égal à la valeur cherchée, c'est fini
- S'il est inférieur à la valeur cherchée, il ne reste à traiter que la moitié droite de la liste
- S'il est supérieur à la valeur cherchée, il ne reste à traiter que la moitié gauche de la liste

On continue ainsi la recherche en diminuant à chaque fois de moitié le nombre d'éléments de la liste restant à traiter.

Pour réaliser la recherche dichotomique dans une liste triée, la fonction Recherche_dicho est implémentée, elle prend 4 paramètres : l'élément x à cherchée, L : la tête de la liste chaînée, le début et la fin de la sous liste dans la quelle x est cherchée.

Dans l'algorithme principal, l'appel sera comme suit :

**Recherche_dicho(x, L, 1, longueur (L))**

Avec longueur (L) est une fonction qui renvoi la longueur de la liste chaînée.

```
Fonction Recherche_dicho(x : entier, L : Liste, debut : entier, fin:entier) : Booléen
 Var
   Rang, milieu, e : entier
 Début
    si debut<=fin alors
       milieu ← (fin+debut) div 2
       e ← ieme(milieu,L)
       si (x=e) alors
          // on a trouve l'element
          retourner VRAI
       sinon
          si (x<e) alors
             // il est peut être dans la moitié gauche
             retourner Recherche_dicho(x, L, debut, milieu-1)
          sinon              // il est peut être dans la moitié droite
             retourner Recherche_dicho(x, L, milieu+1, fin)
          finsi
        finsi
    sinon
       // la liste à traiter est vide
       retourner FAUX
    finsi
  fin
```

La fonction **ieme**(milieu, L) est une fonction qui retourne la valeur de la cellule ayant la position *milieu* de la liste *L*.

```
Fonction ieme(pos : entier, L : Liste) : entier
Var
P : Liste
i : entier
Début
   P ← L
   i ← 1
   Tantque (i < pos) faire
      P ← P^.suivant
      i ← i + 1
   fin tantque
   retourner P^.valeur
Fin
```

La fonction longueur (L) est une fonction qui calcule la longueur d'une liste chaînée L.

```
Fonction longueur (L : Liste) : entier
Var
P : Liste
nb : entier  //nb est le nombre d'élément de la liste
Début
   P ← L
   nb ← 0
   Tantque (P<>Nil) faire
     nb ← nb + 1
     P ← P^.suivant
   fin tantque
   Retourner (nb)
Fin
```

## 7.3 Recherche dans un arbre binaire

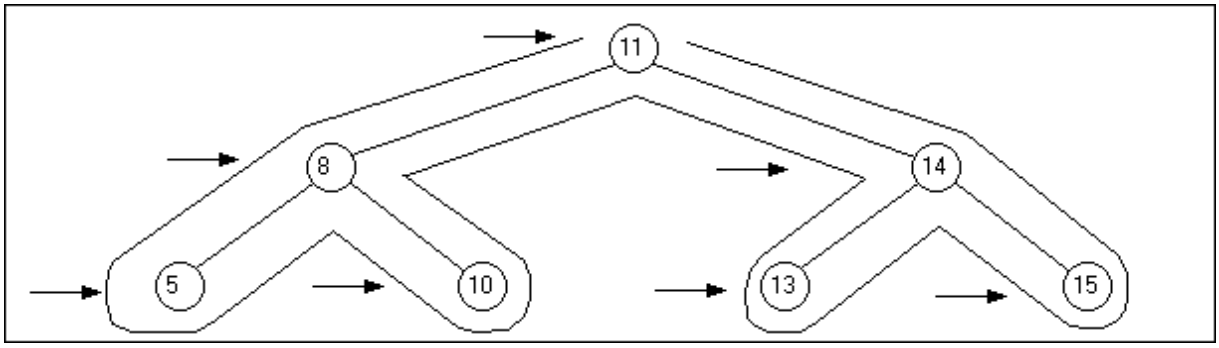
La recherche dans un arbre binaire d'un nœud ayant une valeur particulière est un procédé récursif. On commence par examiner la racine. Si sa valeur est la valeur recherchée, l'algorithme se termine et renvoie la racine. Si elle est strictement inférieure, alors elle est dans le sous-arbre droit, sur lequel on effectue alors récursivement la recherche. De même si la valeur de la racine est strictement supérieure à la valeur recherchée la recherche continue sur le sous-arbre gauche. Si on atteint une feuille dont la valeur n'est pas celle recherchée, on sait alors que la valeur recherchée n'appartient à aucun nœud. On peut la comparer avec la recherche par dichotomie qui procède à peu près de la même manière.

### 7.3.1 Parcours en profondeur

Le parcours en profondeur est un parcours récursif sur un arbre. Il existe trois ordres pour cette méthode de parcours.

#### 7.3.1.1 Parcours préfixé

Dans ce mode de parcours, on traite la racine, puis le sous-arbre gauche, puis le sous-arbre droit.




Ainsi, un parcours préfixé pour cet arbre va nous donner comme résultat :

➔ 11- 8 -5 -10 -14 -13 -15

La procédure récursive de la recherche préfixée est :

```
Procédure préfixé ( A : Arbre )
 Début
  si (A < > nil) alors
```

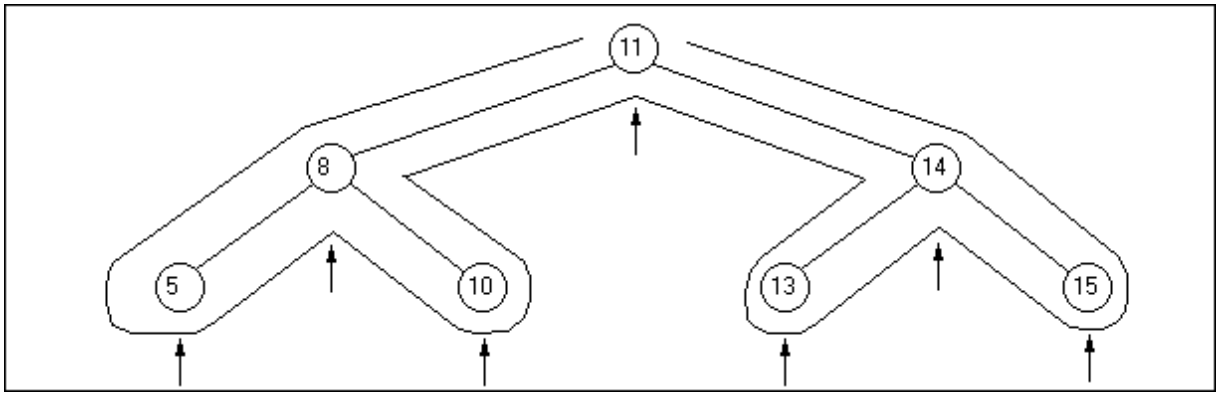
```
    traiter(A)
    préfixé(A^.FilsG)
    préfixé(A^.FilsD)
  finsi
Fin
```

**Remarque :**

- **Traiter(A)** : est le traitement à exercer sur l'arbre de racine A, ajout, suppression, recherche, affichage, ...

### 7.3.1.2 Parcours infixé

Dans ce mode de parcours, on traite le sous-arbre gauche, puis la racine, puis le sous-arbre droit.




Ainsi un parcours infixé pour cet arbre va nous donner comme résultat :

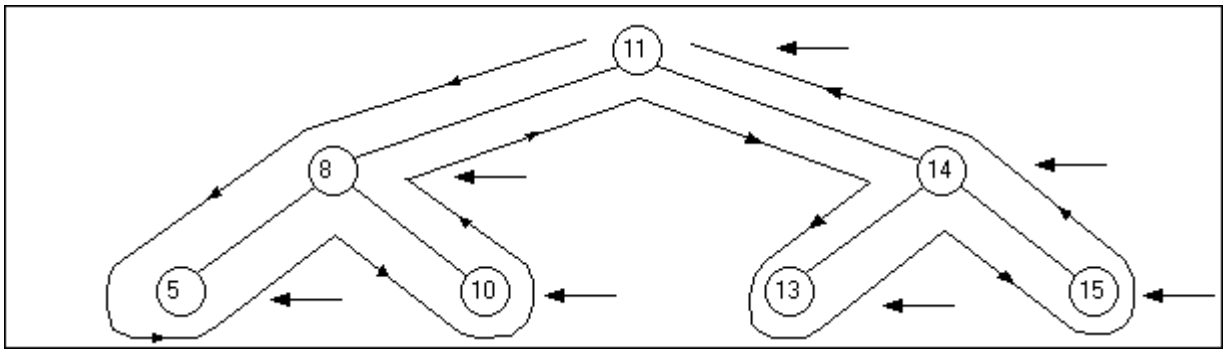
➔ 5 – 8 – 10 – 11 – 13 – 14 – 15

La procédure récursive de la recherche infixée est :

```
Procédure infixé ( A : Arbre )
 Début
  si (A < > nil) alors
    infixé(A^.FilsG)
    traiter(A)
    infixé(A^.FilsD)
  finsi
Fin
```

### 7.3.1.3 Parcours postfixé

Dans ce mode de parcours, on traite le sous-arbre gauche, le sous-arbre droit, puis la racine.




La procédure récursive de la recherche postfixée est :

```
Procédure postfixé (A : Arbre )
 Début
  si (A < > nil) alors
    postfixé(A^.FilsG)
    postfixé(A^.FilsD)
    traiter(A)
  finsi
Fin
```

## 7.3.2 Application

Recherche récursive du père d'une valeur dans un sous arbre de racine A :

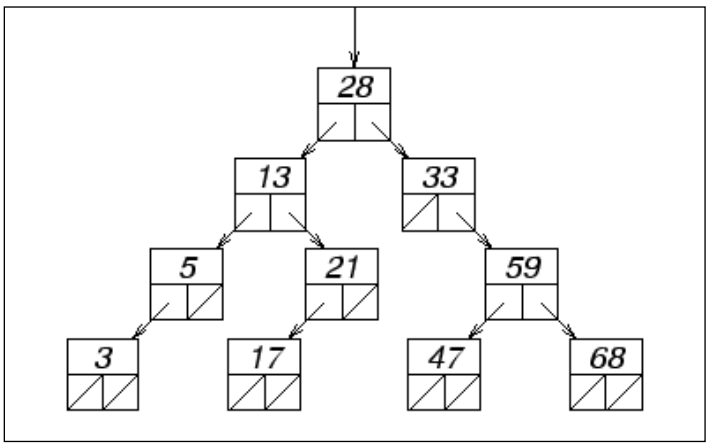
```
Algorithme pere(x : entier, A : Arbre): Arbre


  Var
    tmp, P : Arbre
  Début
    si (A=Nil) alors
       Retourner (Nil)
    sinon
       P ← A
```

```
        si (P^.valeur=x) alors
            Retourner (Nil)        // cas particulier pere de la racine
        sinon
          si (P.FilsG^.valeur=x) ou ( P.FilsD^.valeur=x) alors
            Retourner (P)
          sinon
            tmp ← pere(x, P^.FilsG)    // recherche récursive dans le SAG
            si (tmp=Nil) alors
                  // s'il n'est pas à gauche, c'est qu'il est à droite
                  tmp ← pere(x, P^.FilsD)
            finsi
            Retourner (tmp)
          finsi
        finsi
      finsi
  Fin
```

### 7.3.3 Arbre binaire de recherche

Un *arbre binaire de recherche* (ABR) est un arbre binaire tel que la valeur stockée dans chaque sommet est supérieure à celles stockées dans son sous-arbre gauche et inférieur à celles de son sous-arbre droit.

*Exemple :*




Algorithme de recherche dans un arbre binaire de recherche:

```
Algorithme Recherche(e : entier, B : arbre) : booléen
  // vérifier si e appartient au sous-arbre B (qui peut être vide)
    Début
      Si (B=Nil) Alors       // l'élément n'est pas dans l'arbre
          Retourner FAUX
      Sinon
          Si (e=B^.valeur) Alors       // on l'a trouvé
                Retourner VRAI
          Sinon
              Si (e<B^.valeur) Alors
                      // s'il existe, c'est dans le sous-arbre gauche
                      Retourner Recherche(e, B^.FilsG)
              Sinon
                      // s'il existe, c'est dans le sous-arbre droit
                      Retourner Recherche(e, B^.FilsD)
              Finsi
          Finsi
      Finsi
    fin
```

# Chapitre 8: Complexité des algorithmes

## Vue d'ensemble

Ce Chapitre présente la complexité des algorithmes.

### Objectifs

Ce chapitre a pour objectifs de permettre aux étudiants d'acquérir les connaissances de base se rapportant aux objectifs spécifiques suivants:

- Etre capable d'évaluer un algorithme en calculant sa complexité
- Savoir calculer la complexité d'un algorithme.

### Pré-requis

- Algorithmique 1
- Algorithmique 2

### Durée

- 3 H

## Eléments de contenu

1. Introduction
2. Définition
3. Calcule de la complexité
4. Type de la complexité
5. Complexité de quelque algorithme

## 8.1 Introduction

Un algorithme est une procédure finie et un enchaînement des étapes de résolution d'un problème. Un algorithme doit être appliqué sur toutes les données possibles du problème et doit fournir une solution correcte dans chaque cas. En informatique, il est possible de résoudre un problème donné par implantation d'un algorithme conçu spécification pour la résolution. Cependant, pour un problème donné, il peur être résolu souvent avec plusieurs algorithmes.

Le problème qui se pose alors : Comment choisir entre les algorithmes? Comment déterminer l'efficacité d'un algorithme par rapport à un autre ? De ces questions est née la complexité des algorithmes.

## 8.2 Définition

La mesure de la complexité d'un algorithme est une estimation de son temps d'exécution ainsi que de l'espace mémoire utilisé. Ces 2 mesures sont appelées complexité spatiale et temporelle de l'algorithme.

- La complexité spatiale exprime la taille occupée par les données de l'algorithme dans la mémoire de l'ordinateur.

- La complexité temporelle exprime le temps que prendra l'exécution de l'algorithme.

La mesure de la complexité permet de décider si un algorithme est efficace ou non indépendamment de la machine qui va l'exécuter et du langage d'implémentation, elle ne dépend que de l'algorithme lui-même.

## 8.3 Calcul de la complexité

Le temps d'exécution d'un algorithme est fonction du nombre d'opérations élémentaires dans cet algorithme. Par exemple, pour le parcourt d'un tableau et la recherche d'un élément donnée, la comparaison entre l'élément courant du tableau et l'élément recherché sera prise en compte.

La complexité d'un algorithme dépend de plusieurs facteurs, tel que la taille des données à traiter et le type d'opérations à effectuer, par exemple une addition est plus rapide qu'une multiplication.

Il n'existe pas de méthode formelle pour calculer la complexité d'un algorithme mais il y a des règles qui peuvent être appliquées.

On note T (Op) : le temps d'exécution d'une opérations Op, voici les règles qui peuvent être appliqué lors du calcul du temps d'exécution d'un algorithme :

- Si Op est une instruction élémentaire alors T (Op) est considéré comme constant et il est noté $O(1)$.
- Si Op est une instruction conditionnelle (si condition alors op1 sinon op2) alors T (Op) est estimé à **T (condition) + Max (T (op1) + T (op2))**.
- Si Op est une boucle (Pour i de 1 à N faire Operation) alors T (Op) est estimé à Somme (**T (Operation) )** qui représente la somme des temps des opérations de chaque itération.

## 8.4 Type de complexité

Les algorithmes usuels peuvent être classés en un certain nombre de grandes classes de complexité.

- Les algorithmes sous-linéaires, dont la complexité est en général en $O(\log n)$. C'est le cas de la recherche d'un élément dans un ensemble ordonné fini de n éléments (recherche d'un élément dans un tableau de taille n).

- Les algorithmes linéaires en complexité $O(n)$ sont considérés comme rapides, comme l'évaluation de la valeur d'une expression composée de $n$ symboles ou les algorithmes de tri de tableaux de n éléments.

- Plus lents sont les algorithmes de complexité située entre $O(n^2)$ et $O(n^3)$, c'est le cas de la multiplication des matrices de taille n.

Puisque on calcule la complexité temporelle en faisant une approximation, on utilise donc les notations suivantes:

$O(1)$ :complexité constante

$O(log(n))$ :complexité logarithmique

$O(n)$ : complexité linéaire

$O(n^2)$ : complexité quadratique

$O(n^3)$ : complexité cubique

$O(n^p)$ : complexité polynomiale

$O(2^n)$ : complexité exponentielle

# 8.5 Complexité de quelques algorithmes

## 8.5.1 Recherche du plus petit élément

Soit l'exemple suivant:

```
fonction plusPetit (x: Tableau[1..n] de entier)
var      k,i: entier
Debut
    k←0
         pour i de 1 à  n  faire

       si(x[i] < x[k]) alors
          k ← i
           finsi
         finpour
         Retourner (k)
Fin
```

Dans cette fonction, on exécute *n*-1 tests de comparaison. La complexité temporelle est donc *n*-1 elle est noté *O*(*n*).

## 8.5.2 Produit de matrice

Soit l'exemple suivant:

```
Algorithme Produit_Matrice
var
    u,v, w : Tableau[1..n] de entier
  i : entier
Debut
pour i de 1  à n faire
         pour j de 1  à n faire
              w[i,j] ← u[i,j] * v[i,j]
          finpour
fin pour
fin
```

La complexité temporelle de cet algorithme est la somme des temps nécessaire pour faire les multiplications sur une ligne de la matrice w à savoir n * le temps d'une multiplication qui est considéré comme constante donc O(n). Ce calcul est répété pour chaque ligne donc la complexité précédente doit être multipliée par n.

Finalement on peut en déduire que l'algorithme est de complexité égale à $O(n^2)$.

## 8.6 Exercices d'application

**Exercice 1 :**

Donnez la complexité des programmes suivants

```
(a) pour i = 1 à  n faire
      pour j = 1 à n faire
          x +← 3
```

```
(b) pour i = 1 à n faire
      pour j = 1 à i faire
          x +← 3
```

```
(c) pour i = 5 à  n-5 faire
      pour j = i-5 à  i+5 faire
          x +← 3
```

```
(d) pour i = 1 à n faire
      pour j = 1 à n faire
          pour k = 1 à n faire
              4 x ← x+a
```

```
(f) pour i = 1 à n faire
      pour j = 1 à i faire
          pour k = 1 à j faire
              x ← x+a
```

**Exercice 2 :**

Soit l'algorithme suivant:

```
Fonction  Partition (var T : tableau [1..N] de entier, deb, fin : entier) : entier
Var
  i, j, v, p: entier
Debut
  i← deb-1
  j← fin
  p ← T[fin]
  Repeter
    Repeter
      i ← i+1
    Jusqu'à ( T[i] ≥ p )
```

```
    Repeter
      j ← j-1
    Jusqu'à ( T[j] ≤ p )
    Si ( i< j) Alors
      v ← T[i]
      T[i] ← T[j]
      T[j] ← v
    FinSi
  Jusqu'à ( i ≥ j )
  T[fin] ← T[i]
  T[i] ← p
  Retourner (i)
Fin
Procedure  Tri_Rapide (var T : tableau[1..N] de entier, deb, fin :entier)
Var
  Pos : entier
Debut
  Si (deb < fin) Alors
    Pos ← Partition (T,deb,fin)
    Tri_Rapide (T,deb,pos-1)
    Tri_Rapide (T,pos+1,fin)
  FinSi
Fin
```

Déterminer la complexité de la procédure **Tri_Rapide**

## 8.7 Solutions des exercices

**Exercice 1 :**

**a- O(N)**

**b- O ($N^2$)**

**c- O ($N^2$)**

**d- O ($N^3$)**

**f- O ($N^3$)**

**Exercice 2 :**

O (log(n))

# DS et examen

# Bibliographie

## Livres

**[1]:** Amina Bouraoui baccar, *Algorithmique avancée et structures de données cours et applications*.

**[2]:** Warin Bruno, *L'algorithmique votre passeport informatique pour la programmation*, Ellipses, 2002

**[3]:** ROHAUT, *Algorithmique et Techniques fondamentale de programmation*, Edition Eni 2007

**[3]:** LIGNELET P., *Algorithmique. Méthodes et modèles*, Paris : Masson, 1985

## Sites Web

**[1]:** www.intelligentedu.com/blogs/post/free_computer_books/3760/the-algorithm-design-manual/fr/