Haute Ecole de Namur - Liège - Luxembourg
Département technique
Implantation IESN

Bachelier en Informatique

UE 159

# Organisation et exploitation des données

Corinne Derwa

Année académique 2023-2024

# ORGANISATION DU COURS

## UE : ORGANISATION ET EXPLOITATION DES DONNÉES

- Une seule AA de 4 ECTS
- Théorie : 16 heures
- Exercices : 32 heures
- Prérequis :
  - Principes de programmation

## EVALUATION ET RÉPARTITION DES POINTS

➔ Session de juin :
- Une interrogation formative en cours de quadrimestre
- Examen :
  - Evaluation d'applications de notions théoriques
  - Evaluation d'exercices

➔ Session de septembre :
- Examen :
  - Evaluation d'applications de notions théoriques
  - Evaluation d'exercices

## PLAN DU COURS

- *Module 1 : Introduction*
- *Module 2 : Les tableaux - compléments*
- *Module 3 : Les listes chainées*
- *Module 4 : Les piles et files*
- *Module 5 : Les arbres, arbres binaires, arbres binaires de recherche.*

# MODULE 1 : INTRODUCTION

Le principe de base d'une *structure de données* est de stocker en mémoire des données auxquelles le programmeur veut pouvoir accéder plus tard et sur lesquelles il veut pouvoir effectuer des opérations (lecture, mise à jour, suppression, insertion…).

Toutes les structures de données ne permettent pas les mêmes opérations, et surtout elles n'ont pas toujours le même *coût*. Par exemple, certaines structures facilitent l'ajout d'éléments contrairement à d'autres qui nécessitent une réorganisation complète. Le coût des structures de données peut se mesurer en termes de *complexité* d'un algorithme : pour chaque structure de données utilisée, on peut mesurer la complexité des opérations que l'on effectue.

On peut distinguer deux types de complexité, selon que l'on s'intéresse au temps d'exécution ou à l'espace mémoire occupé.

Complexité en temps : réaliser un calcul de complexité en temps revient à compter le nombre d'opérations élémentaires (affectation, calcul arithmétique ou logique, comparaison…) effectuées par l'algorithme. Pour rendre ce calcul réalisable, on émettra l'hypothèse que toutes les opérations élémentaires sont à égalité de coût. En pratique ce n'est pas tout à fait exact mais cette approximation est cependant raisonnable. On pourra donc estimer que le temps d'exécution de l'algorithme est proportionnel au nombre d'opérations élémentaires.

Complexité en espace : la complexité en espace est la taille de la mémoire nécessaire pour stocker les différentes structures de données utilisées lors de l'exécution de l'algorithme.

Cette connaissance des coûts a deux intérêts. Lorsqu'on nous donne un algorithme utilisant une structure de données particulière, on peut déterminer la complexité des opérations effectuées pour évaluer la complexité globale de l'algorithme. Mais surtout, quand on a une idée des opérations dont on a besoin pour un algorithme, on peut *choisir* la structure de données la plus adaptée (celle pour laquelle ces opérations sont les moins coûteuses).

Dans la pratique, le seul choix de la structure de données adéquate peut faire la différence au niveau des performances. La bonne connaissance et la sélection des structures de données adéquates sont des compétences importantes pour le programmeur.

Le cours d'organisation et exploitation des données sera essentiellement consacré à la définition des différentes structures (***organisation des données***), au choix de la structure optimale dans un problème donné ainsi qu'à l'utilisation des différentes structures dans les diagrammes d'actions (***exploitation des données***). La notion de complexité n'y sera que légèrement abordée mais elle est étudiée de manière plus approfondie dans le cours de Modélisation et traitement des données de l'UE IG131.

**Exemple** :

Considérons un ensemble de données concernant des clients, réparties en

- nom
- rue
- ville
- profession
- téléphone

…

A chaque usage, correspond une structure appropriée. En effet, les données peuvent être triées par ordre alphabétique sur le nom, ou rangées par même ville ou par même profession…

Le choix d'une structure appropriée permettra un traitement plus rapide et plus efficace.

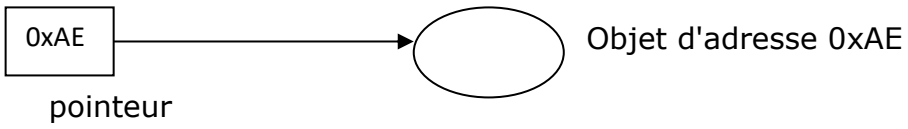
⇨ **Utiliser une structure de données appropriée permet de diminuer :**
- **la complexité d'une application informatique,**
- **le taux d'erreurs.**

Ne confondons pas *type de donnée* et *structure de données*.

- Le **type** d'une donnée caractérise l'ensemble des valeurs auquel une constante ou un littéral appartient ou qu'une variable ou expression peut prendre.

  Types de données primitifs :

  - *Entier* : il peut être de taille variable. Actuellement, un entier "normal" est codé sur 4 octets. => Résultat exact dans les opérations.
  - *Réel* : (codé sur 6 à 8 octets) précision suivant le nombre de décimales stockées => erreurs d'arrondi
  - *Booléen* : vrai ou faux, correspondance binaire.
  - *Caractère* : (codé sur 1 octet) stockage suivant le code ASCII ou EBCDIC.
  - *Pointeur* : adresse en mémoire permettant de repérer un objet.
    Sa valeur (contenu) désigne la place mémoire occupée par un objet.



- La **structure de données** (ou collection) est une structure logique destinée à contenir des données organisées de manière à simplifier le traitement.

  Types de structures de données :

  - Séquentielles (ou linéaires) : on peut y ranger les objets dans un ordre arbitraire.
    - Exemples :
      - Tableaux ;
      - Listes chainées ;
      - Piles ;
      - Files ;
  - Mappées : on y range les objets en fonction d'une clé.
    - Exemples :
      - Tables de hachage ;
      - Arbres binaires de recherche ;
      - B-trees ;
  - Autres **:** les graphes par exemple.

**C**ertains diagrammes d'action de ce syllabus sont génériques. Les noms des variables et structures sont de ce fait également génériques. Dans la pratique, ces noms doivent respecter le clean code. Ainsi, les noms **valeurs** , **nbValeurs**, **clé** et **cléRecherchée** par exemple seront remplacés par des noms évoquant le contexte et leur contenu.

# MODULE 2 : LES TABLEAUX - COMPLÉMENTS

## 2.1. Définition

Les tableaux sont les structures de données les plus anciennes que l'on retrouve dans les premiers langages de programmation évolués.

Un tableau est une structure de données :

- Homogène : tous les éléments sont du même type ;
- À accès direct : tous les éléments peuvent être choisis aléatoirement et on peut y accéder directement (grâce à l'indice). Le temps d'accès à un élément par son indice est constant, quel que soit l'élément désiré. Cela s'explique par le fait que les éléments d'un tableau sont contigus dans l'espace mémoire. Ainsi, il est possible de calculer l'adresse mémoire de l'élément auquel on veut accéder, à partir de l'adresse de début du tableau et de l'indice de l'élément. L'accès est immédiat, comme il le serait pour une variable simple.

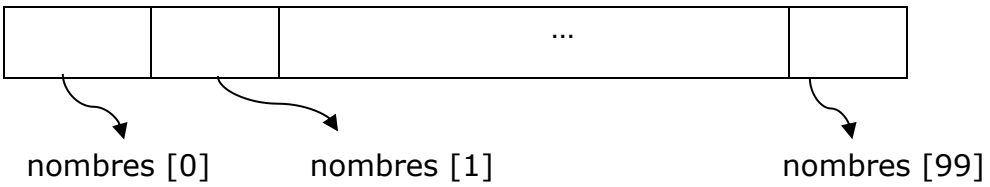
### 2.1.1. Tableau à une dimension

La forme de tableau la plus simple est le vecteur.

Un vecteur est une liste ordonnée de composants de même type. On accède à chaque élément du tableau (cellule) via son *indice*.

Représentation

**nombres** : Tableau de **100** cellules, chacune contenant un nombre




- Les éléments sont rangés de manière consécutive.
- L'espace mémoire est alloué de façon statique car la longueur du tableau est fixe.

Déclaration

- En PP, le tableau peut être identifié par l'instruction ARRAY(taille) où taille est un entier strictement positif ou une variable contenant un entier strictement positif, il représente la taille physique du tableau. En mémoire, un espace de ***taille*** cellules contigües est réservé.
- Lorsqu'un tableau n'est pas entièrement garni, les cellules non utilisées se trouvent généralement à la fin. Dans certains cas, ces cellules sont initialisées,

par exemple à « » ou 0 selon leur type. Mais le plus souvent, pour gérer ce tableau, on utilise une variable qui représente à tout instant le nombre réel de cellules garnies. Cette variable est la taille logique du tableau.

Tableau de structures

Le contenu d'une cellule peut comporter plusieurs champs dont certains peuvent être eux-mêmes des structures de données. Ainsi une cellule de tableau peut elle-même contenir un tableau ou, comme nous le verrons par la suite, un pointeur vers une liste chainée.

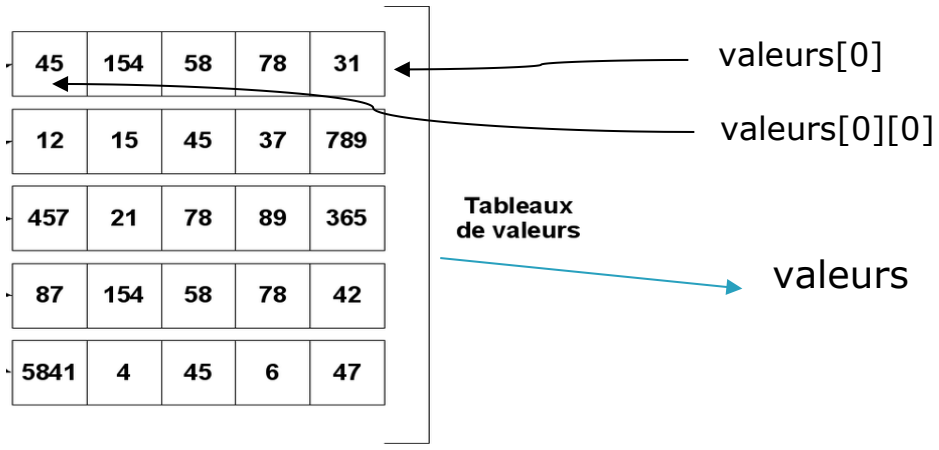
## 2.1.2. Tableau à une dimension trié

Les éléments (cellules) sont ordonnés suivant une relation d'ordre (alphabétique, croissant, chronologique…) sur un champ.

Les algorithmes de recherche vont varier selon que le tableau est trié ou non (voir point 2.2.). Les opérations d'insertion nécessiteront une recherche de la position puis un décalage des cellules dans le cas d'un tableau trié alors que dans le cas d'un tableau non trié, l'ajout peut se faire en fin de tableau.

## 2.1.3 Tableau à deux dimensions

Un tableau à deux dimensions (aussi appelé matrice) est un tableau normal (à une dimension) dont les éléments sont eux-mêmes des tableaux contenant les éléments du tableau à deux dimensions.




L'accès aux éléments se fait via deux indices. En prenant la représentation ci-dessus, l'accès à la valeur située en 2e ligne et 5e colonne (la valeur 789), se fait via les indices 1 puis 4 (rappel : on commence la numérotation des indices à zéro).

On a donc valeurs (1, 4) = 789 ou valeurs [1][4] = 789

# 2.2. Algorithmes de recherche

## 2.2.1. Recherche dans un tableau non trié

Dans ce type de tableau, la seule recherche possible est une recherche séquentielle.

Soit un tableau **valeurs** de **nbValeurs** cellules dont un champ s'appelle **clé**. La variable **nbValeurs** représente le nombre de cellules garnies du tableau, c'est donc la taille logique du tableau. Cette remarque est valable pour tous les exemples qui suivent.

Ecrire le module qui reçoit une valeur de clé (**cléRecherchée**) et qui affiche le message adéquat selon que la clé a été trouvée ou non.

```
o──────────────────────────────o ↓ valeurs, nbValeurs, cléRecherchée
│ rechercherDansTableauNonTrié  │
o──────────────────────────────o
┌── *
│ o──────────────────o ↓ valeurs, nbValeurs, cléRecherchée
│ │ indiceRecherché  │
│ o──────────────────o ↓ iValeur
│
│ ┌── if(iValeur == nbValeurs)
│ │ sortir "clé inexistante"
│ ├── else
│ │ sortir "clé trouvée dans la cellule d'indice ", iValeur
│ └──
└──────────
```

```
o──────────────────o ↓ valeurs, nbValeurs, cléRecherchée
│ indiceRecherché  │
o──────────────────o ↓ iValeur

┌── *
│ iValeur = 0
│ ╔══ while (iValeur < nbValeurs and valeurs[iValeur].clé ≠ cléRecherchée)
│ ╚┌─ iValeur ++
└──
└──────────
```

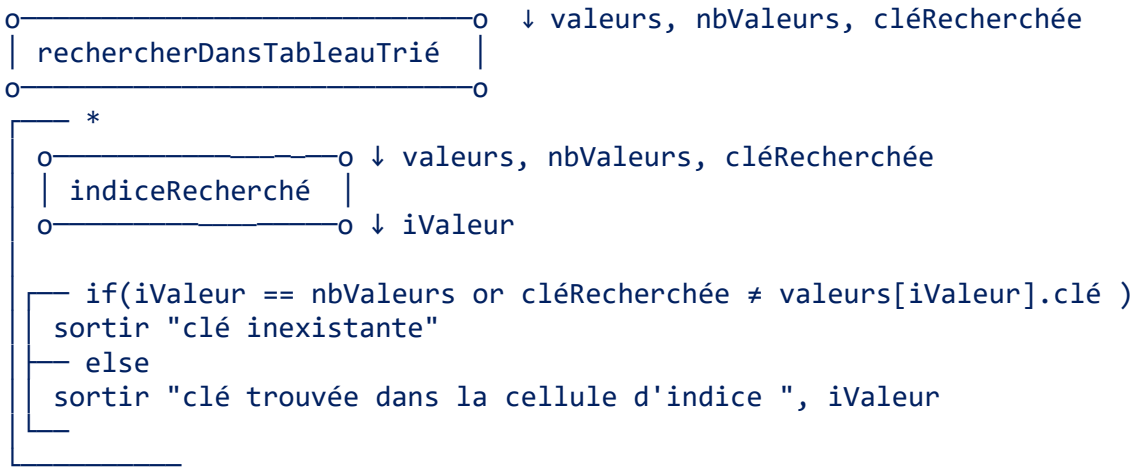
**Complexité en temps** :

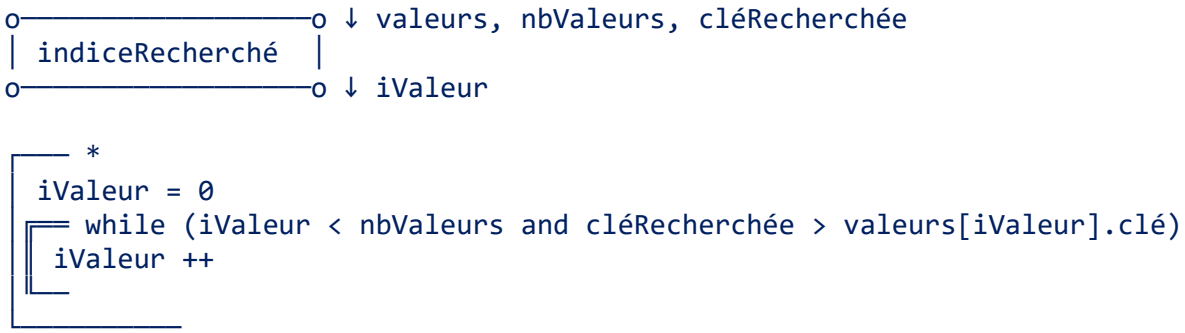
- Si recherche fructueuse : (nbValeurs + 1) / 2
- Si recherche infructueuse : nbValeurs

## 2.2.2. Recherche séquentielle dans un tableau trié

Soit un tableau **valeurs** de **nbValeurs** cellules dont un champ s'appelle **clé**, ce tableau est **trié par ordre croissant sur la clé**.

Ecrire le module qui reçoit une valeur de clé (**cléRecherchée**) et qui affiche le message adéquat selon que la clé a été trouvée ou non.




**Complexité en temps** :

- Si recherche fructueuse : (`nbValeurs` + 1) / 2
- Si recherche infructueuse : (`nbValeurs` + 1) / 2

## 2.2.3. Recherche dichotomique dans un tableau trié

Soit un tableau **valeurs** de **nbValeurs** dont un champ s'appelle **clé**, ce tableau est **trié en ordre croissant sur la clé**.

Ecrire le module qui reçoit une valeur de clé (**cléRecherchée**) et qui affiche le message adéquat selon que la clé a été trouvée ou non.

```
o──────────────────────────────o ↓ valeurs, nbValeurs, cléRecherchée
│ rechercherMéthodeDichotomique │
o──────────────────────────────o
┌─── *
│ borneInf = 0
│ borneSup = nbValeurs - 1
│
│ iMilieu= [(borneInf + borneSup)/2]ENT
│
│╔══ while (borneInf ≤ borneSup and valeurs[iMilieu].clé ≠ cléRecherchée)
│║
│║┌── if (cléRecherchée < valeurs[iMilieu].clé ))
│║│ borneSup = iMilieu - 1
│║├── else
│║│ borneInf = iMilieu + 1
│║└──
│║ iMilieu= [(borneInf + borneSup)/2]ENT
│║
│╚══
│┌── if(borneInf > borneSup)
│ sortir "clé inexistante"
│├── else
│ sortir "clé trouvée dans la cellule d'indice ", iMilieu
│└──
└──────────
```

**Complexité en temps** :

- Si recherche fructueuse : $\log_2$ (nbValeurs - 1)
- Si recherche infructueuse : $\log_2$ nbValeurs + 1

# 2.3. Bloc logique

Cette notion que nous allons étudier sur les tableaux à une dimension, n'est pas une notion propre à ces derniers. Nous verrons plus loin que l'on peut aussi trouver des blocs logiques dans les listes chainées ou les fichiers séquentiels.

## 2.3.1. Définition

Lorsque, dans un tableau trié selon un champ déterminé, les cellules successives contiennent la même valeur de champ, on a un *bloc logique*.

**Exemple** : Soit un tableau **écoles** de **nbEcoles** cellules dont chaque cellule contient un nom de ville et un nom d'école. Ce tableau est trié par ordre alphabétique sur le nom des villes et sa structure ***physique*** est représentée ci-dessous.

**écoles** [ cellules (**nbEcoles** *) ] [ **nomVille ( ↗ )** **nomEcole** ]

Il est évident que plusieurs cellules successives vont contenir le même nom de ville. Il y a donc un niveau de bloc logique sur base de la ville. Ce tableau peut ainsi être vu comme une succession de blocs de cellules. Chaque bloc est caractérisé par un nom de ville et des noms d'école.

Il est donc possible de représenter la structure ***logique*** du tableau comme suit :

**écoles** [ bloc ville (…*) ] [ **nomVille** par école de la ville (…*) [ **nomEcole** ] ]

*Illustration*

| Liège | Liège | Namur | Namur | Namur | Namur | Verviers | Verviers | … |
|---|---|---|---|---|---|---|---|---|
| ARL1 | ARL4 | Ecole3 | Ecole2 | Ecole1 | Ecole4 | E.Prov | A.Royal | … |

bloc ville 1 bloc ville 2 bloc ville 3 …

Comme nous le verrons au point suivant, lorsque le traitement à effectuer sur le tableau utilise cette structure logique (dans cet exemple, par ville), il est nécessaire d'adapter le diagramme d'actions afin d'en optimiser le code.

**Remarque** : Il n'est pas nécessaire que le tableau soit trié par ordre alphabétique sur les noms des villes pour utiliser le raisonnement qui suit. Il suffit en effet qu'il soit « rangé » par nom de ville, quel que soit l'ordre des noms.

## 2.3.2. Algorithme utilisant un bloc logique

On nous demande, à partir du tableau **écoles** décrit ci-dessus, d'afficher, pour chaque ville, le nom de la ville ainsi que la liste des écoles de cette ville suivie du nombre d'écoles de la ville.

Nous pouvons donc schématiser les sorties comme suit :

Sorties

Par ville (… *)
- nom de la ville
- par école (… *)
  - nom de l'école
- nombre d'écoles

*Illustration*

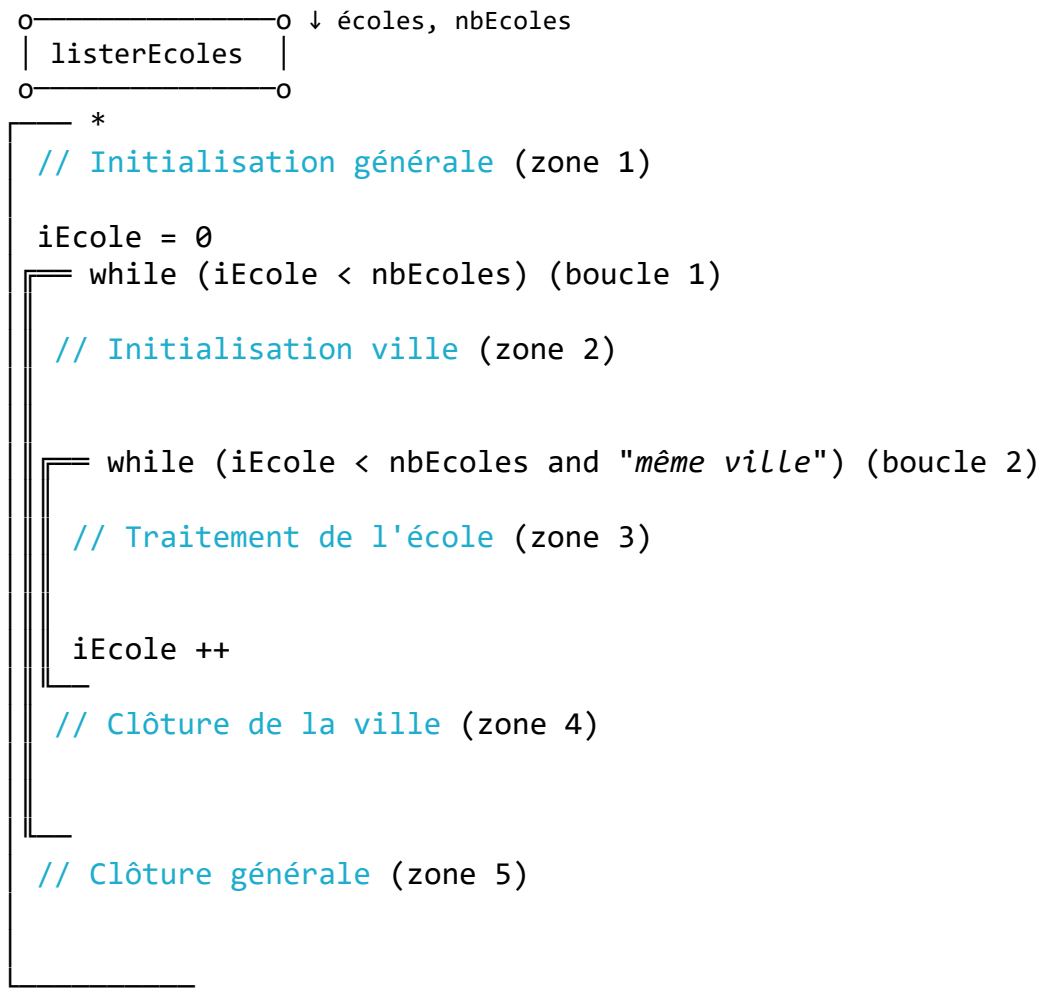
**Ville :** Liège
**Liste des écoles :**
ARL1
ARL4
**Nombre d'écoles :** 2

**Ville :** Namur
**Liste des écoles :**
Ecole3
Ecole2
Ecole1
Ecole4
**Nombre d'écoles :** 4

**…**

Vu la structure des sorties, nous écrirons un diagramme d'actions qui met en évidence la structure de bloc logique.

Ecrivons d'abord l'"ossature générale" du diagramme en faisant apparaître, via 2 boucles, 5 zones différentes numérotées de 1 à 5.




Nous remarquons que

- La boucle 1 sera exécutée autant de fois qu'il y a de villes différentes dans le tableau c'est-à-dire autant de fois qu'il y a de blocs "ville " ;
- Pour une ville donnée, la boucle 2 sera exécutée autant de fois qu'il y a d'écoles dans cette ville ;
- La zone 1 contient les instructions qui seront exécutées une seule fois en début de diagramme ;
- La zone 2 contient les instructions qui seront exécutées une fois par ville, au début du traitement de la ville ;
- La zone 3 contient les instructions qui seront exécutées pour chaque école ;
- La zone 4 contient les instructions qui seront exécutées une fois par ville, à la fin du traitement de la ville ;
- La zone 5 contient les instructions qui seront exécutées une seule fois en fin de diagramme.

Nous pouvons à présent compléter le DA de la page précédente :

```
o──────────────o ↓ écoles, nbEcoles
│ listerEcoles │
o──────────────o
 ┌──── *
 │
 │  // Initialisation générale
 │
 │  iEcole = 0
 │ ╔═══ while (iEcole < nbEcoles)
 │ ║
 │ ║  // Initialisation ville
 │ ║
 │ ║  villeEnCours = écoles[iEcole].nomVille
 │ ║  sortir "Ville: ", villeEnCours
 │ ║  nbEcolesParVille = 0
 │ ║  sortir "Liste des école(s):"
 │ ║
 │ ║ ╔═══ while (iEcole < nbEcoles and
 │ ║ ║                  villeEnCours == écoles[iEcole].nomVille)
 │ ║ ║
 │ ║ ║  // Traitement de l'école                = " même ville "
 │ ║ ║
 │ ║ ║  nbEcolesParVille ++
 │ ║ ║  sortir écoles[iEcole].nomEcole
 │ ║ ║  iEcole ++
 │ ║ ╚═══
 │ ║
 │ ║  // Clôture de la ville
 │ ║
 │ ║  sortir "Nombre d'écoles : ", nbEcolesParVille
 │ ╚═══
 │
 │  // Clôture générale
 │
 └──────────
```

## 2.3.3. Plusieurs blocs logiques

Certains tableaux peuvent contenir plusieurs blocs logiques.

Ainsi un tableau **hôtels** dont chacune des **nbHôtels** cellules concerne un hôtel et contient les informations suivantes : la catégorie de l'hôtel (c'est-à-dire le nombre d'étoiles), le nom de l'hôtel ainsi que le nom de la ville dans laquelle il se situe et le nombre de chambres.

Nous précisons en outre que le tableau est trié par catégorie et, pour chaque catégorie, par ordre alphabétique sur le nom de la ville.

Nous constatons alors que ce tableau est constitué d'un premier niveau de blocs « catégorie » et chaque bloc « catégorie » est lui-même constitué d'un deuxième niveau de blocs « ville ».

Si les sorties le nécessitent, nous devrons utiliser un D.A. dont l'ossature mettra en évidence ce double niveau de bloc logique.

**Exercice résolu**

On demande d'écrire le D.A. qui, à partir de **hôtels** décrit ci-dessus, affiche

- Par catégorie d'hôtel
  - La catégorie
  - Par ville :
    - Le nom de la ville
    - Le nombre d'hôtels
  - Le nom de la ville qui possède le plus d'hôtels
- La catégorie la moins bien représentée.

Nous voyons clairement que les sorties nécessitent d'utiliser la structure de blocs logiques du tableau dans le D.A.

En mémoire

Structure physique

**hôtels** — cellule (**nbHôtels** *) — **catégorie** (↗) ; **nom** ; **ville** (↗↗) ; **nbChambres**

Structure logique

**hôtels** — bloc catégorie (…*) — **catégorie** ; bloc ville (…*) — **ville** ; par hôtel (…*) — **nom** ; **nbChambres**

```
o──────────────────────────────o ↓ hôtels, nbHôtels
│ AfficherStatistiquesHôtels   │
o──────────────────────────────o
──── *
// Initialisation générale
nbMinHôtelsCatégorie = HV
iHôtel = 0

═══ while (iHôtel < nbHôtels )

 // Initialisation catégorie
 catégorieEnCours = hôtels[iHôtel].catégorie
 sortir catégorieEnCours
 nbHôtelsCatégorie = 0
 nbMaxHotelsVille = 0

 ═══ while (iHôtel < nbHôtels and
                  catégorieEnCours == hôtels [iHôtel].catégorie )

  // Initialisation ville
  villeEnCours = hôtels[iHôtel].ville
  sortir villeEnCours
  nbHôtelsVille = 0

  ═══ while (iHôtel < nbHôtels and
                 catégorieEnCours == hôtels [iHôtel].catégorie and
                 villeEnCours == hôtels [iHôtel].ville )

   // Traitement hôtel
   nbHôtelsVille ++
   iHôtel ++
  ────

  // Clôture ville
  sortir nbHôtelsVille
  nbHôtelsCatégorie += nbHôtelsVille
  ──── if (nbHôtelsVille > nbMaxHotelsVille )
   nbMaxHotelsVille = nbHôtelsVille
   nomMaxVille = villeEnCours
  ────

 ═══

 // Clôture catégorie
 ──── if (nbHôtelsCatégorie < nbMinHôtelsCatégorie)
  nbMinHôtelsCatégorie = nbHôtelsCatégorie
  catégorieMoinsReprésentée = catégorieEnCours
 ────

 sortir nomMaxVille
──────

// clôture générale
sortir catégorieMoinsReprésentée
──────────
```

# MODULE 3 : LES LISTES CHAINÉES

## 3.1. Définition

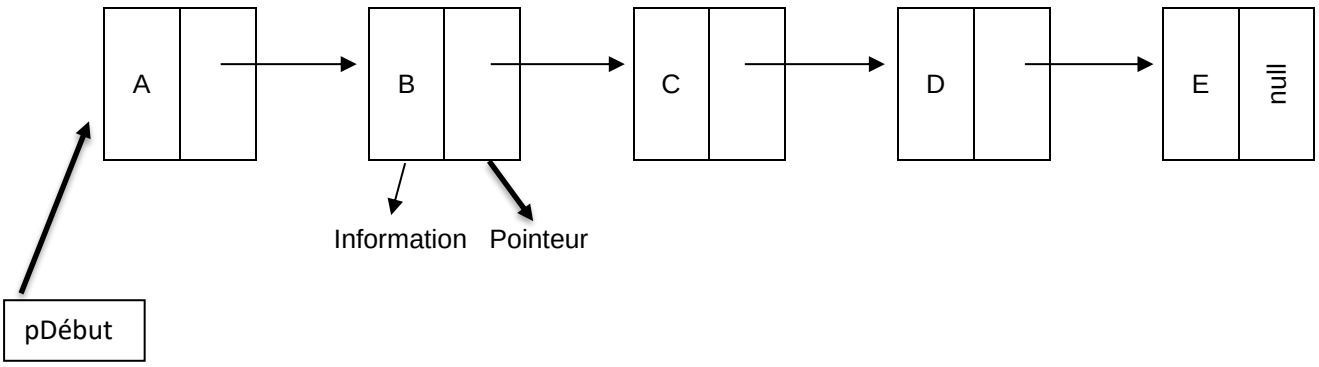
Une liste chainée est une structure de données homogène (tous les éléments sont de même type) constituée d'éléments ordonnés linéairement et chaînés entre eux.

Chaque élément (**chainon**) contient

- Une partie *information* pouvant comporter un certain nombre de champs,
- un (ou plusieurs) pointeur(s) vers un autre élément de la liste ou une marque de fin équivalente à null s'il n'y a pas d'élément successeur.

Une liste chainée peut se présenter sous différentes formes.

La forme la plus simple est la **liste chainée simple**.




La liste chainée simple est repérée par une variable de type pointeur (**pDébut** par exemple) qui contient l'adresse du premier chainon (ou **null** si la liste est vide). Le dernier chainon de la liste possède un pointeur dont la valeur est **null**.

On accède aux divers éléments de la liste en suivant la chaine des pointeurs. Les chainons voisins d'une liste n'occupent pas spécialement des places mémoires consécutives ; ils peuvent être éparpillés dans la mémoire.

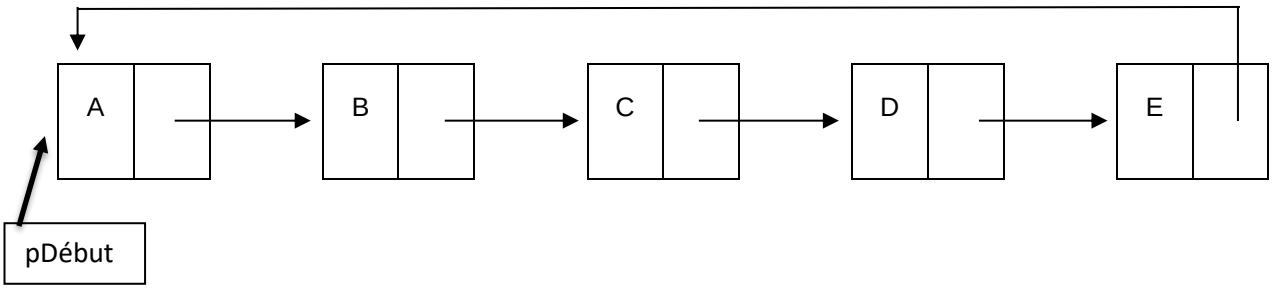
Les listes chainées permettent de gérer la mémoire de manière **dynamique** et sans déplacement de chainons.

En effet, l'espace mémoire est réservé au fur et à mesure de la création de chainons et libéré lors des suppressions.

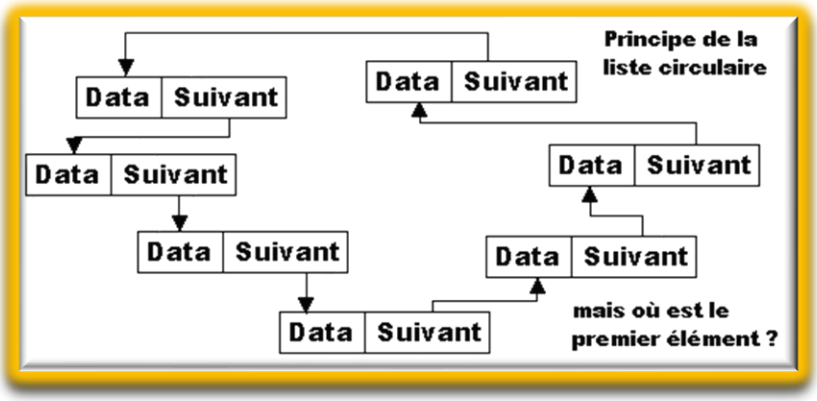
L'utilisation des listes chainées implique l'existence d'un mécanisme permettant **l'allocation dynamique de mémoire**.

Notons encore qu'une liste peut être **non triée ou triée** selon un ou plusieurs champs.

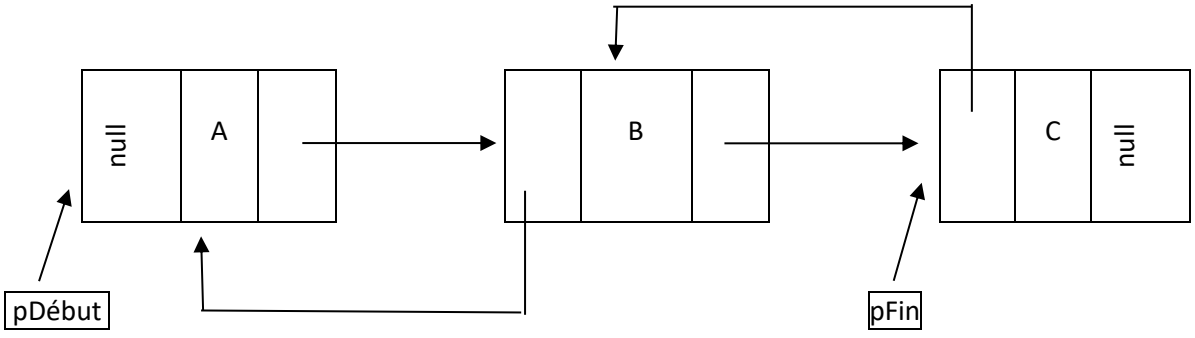
Une deuxième forme de liste est la **liste chainée circulaire**.




Il s'agit d'une liste chainée simple dont le pointeur du dernier chainon n'est pas vide (contenu différent de **null**) mais contient l'adresse du premier chainon. On dit que le dernier chainon pointe vers le premier.



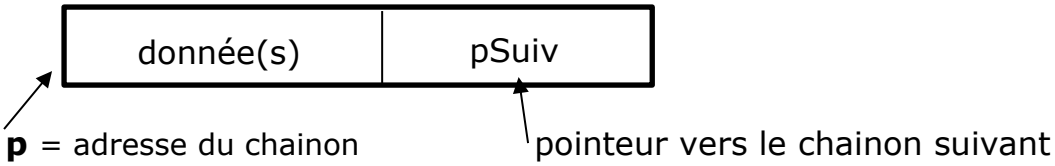

Une autre forme importante de liste est la liste chainée double ou liste symétrique ou encore liste bilatère.



Dans une liste double, chaque chainon est composé de champ(s) contenant les données, d'un pointeur vers le chainon suivant (**pSuiv**) et d'un pointeur vers le chainon précédent (**pPrec**). Un pointeur contient l'adresse du premier chainon (ici **pDébut**) et un éventuel pointeur contient l'adresse du dernier chainon (ici **pFin**).

## 3.2. Notations et opérations

Pour la suite du cours, nous schématiserons un chainon d'une liste simple de la manière suivante :




Nous noterons :

- **p** : pointeur vers un chainon quelconque ;
- **pNouv :** pointeur vers un nouveau chainon **;**
- **p -> donnée(s) :** donnée(s) du chainon pointé par p ;
- **p -> pSuiv :** adresse du chainon qui suit celui pointé par p ;
- **pDébut :** pointeur vers le premier chainon de la liste.

Dans la pratique, pour des raisons de clean code, nous utiliserons des noms de pointeurs plus explicites c'est-à-dire illustrant le contenu du chainon. Par exemple : pClient, pClientNouv, pLivre , pDébutLivres …

Les différentes opérations que l'on peut pratiquer sur une liste et dont nous étudierons quelques cas dans la suite sont :

- recherche d'un élément ;
- insertion d'un nouveau chainon ;
- suppression d'un chainon ;
- union de 2 listes en une seule ;
- séparation d'une liste en 2 listes ;
- copie d'une liste ;
- comptage des éléments d'une liste ;

  …

Les algorithmes du point suivant concernent des listes simples.

# 3.3. Algorithmes

## 3.3.1. Recherche dans une liste simple non triée

```
o──────────────────────────────o ↓ pDébut, donnée
│ rechercherDansListeNonTriée  │
o──────────────────────────────o
┌─── *
│ o─────────────o ↓ pDébut, donnée
│ │ pRecherché  │
│ o─────────────o ↓ p
│
│┌── if (p == null)
│ sortir " Donnée absente de la liste "
│├── else
│ sortir " Donnée trouvée : ", p → donnée
│└──
└────────────
```

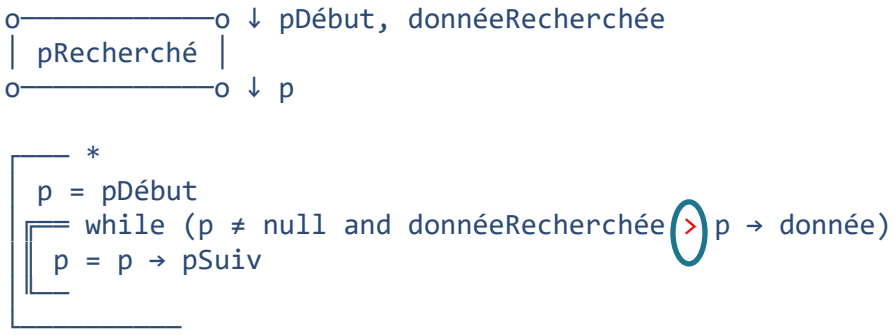
```
o─────────────o ↓ pDébut, donnéeRecherchée
│ pRecherché  │
o─────────────o ↓ p
```

```
┌─── *
│ p = pDébut
│╔══ while (p ≠ null and donnéeRecherchée ≠ p → donnée)
│ p = p → pSuiv
│╚══
└────────────
```

## 3.3.2. Recherche dans une liste simple triée

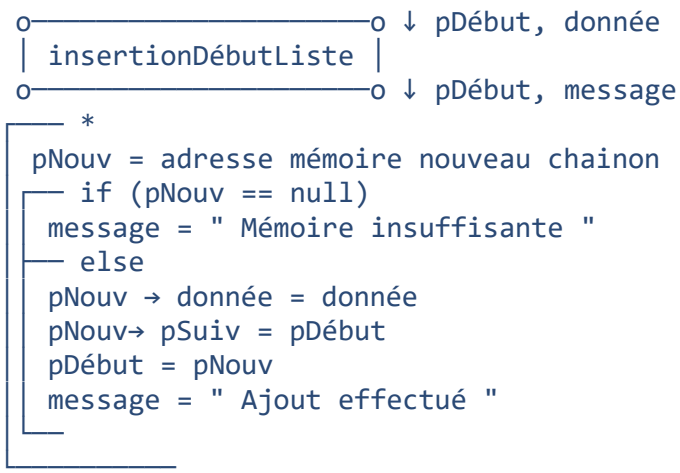
Nous supposons ici que la liste est triée par ordre alphabétique sur le champ ***donnée***.

```
o──────────────────────────────o ↓ pDébut, donnée
│ rechercherDansListeTriée     │
o──────────────────────────────o
┌─── *
│ o─────────────o ↓ pDébut, donnée
│ │ pRecherché  │
│ o─────────────o ↓ p
│
│┌── if (p == null or donnée < p → donnée)
│ sortir " Donnée absente de la liste "
│├── else
│ sortir " Donnée trouvée : ", p → donnée
│└──
└────────────
```


pRecherché
pDébut, donnéeRecherchée
p
*
p = pDébut
while (p ≠ null and donnéeRecherchée > p → donnée)
p = p → pSuiv

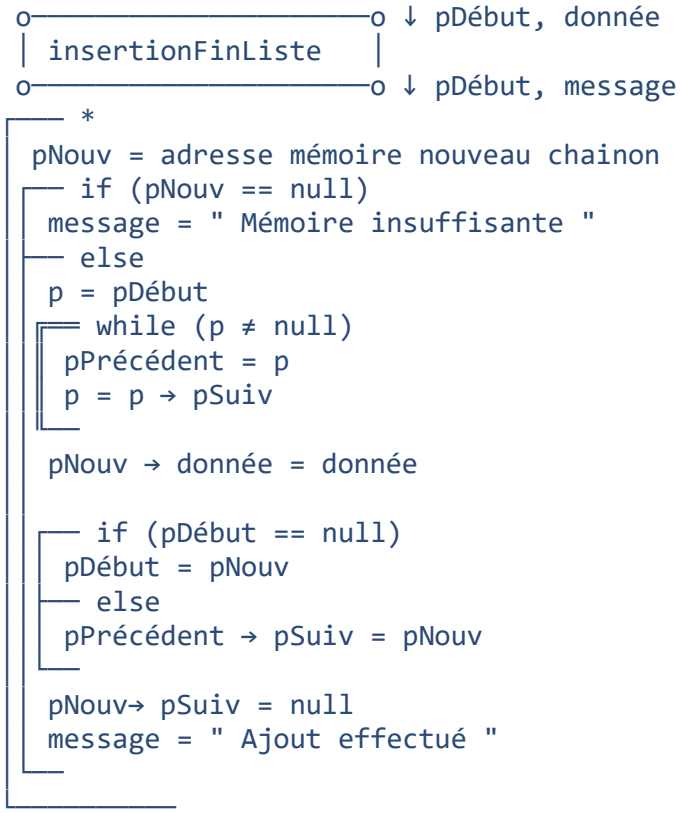
## 3.3.3. Insertion d'un nouveau chainon dans une liste simple

### 3.3.3.1. Insertion en début de liste




Remarquons que ce diagramme est correct également en cas de liste vide au départ.

### 3.3.3.2. Insertion en fin de liste




Remarquons que ce diagramme est correct également en cas de liste vide au départ.

### 3.3.3.3. Insertion dans une liste triée

Puisque la liste est triée, l'endroit où sera inséré le nouveau chainon est imposé par le tri. L'ajout peut se faire dans une liste vide, en début de liste, en milieu de liste ou en fin de liste.

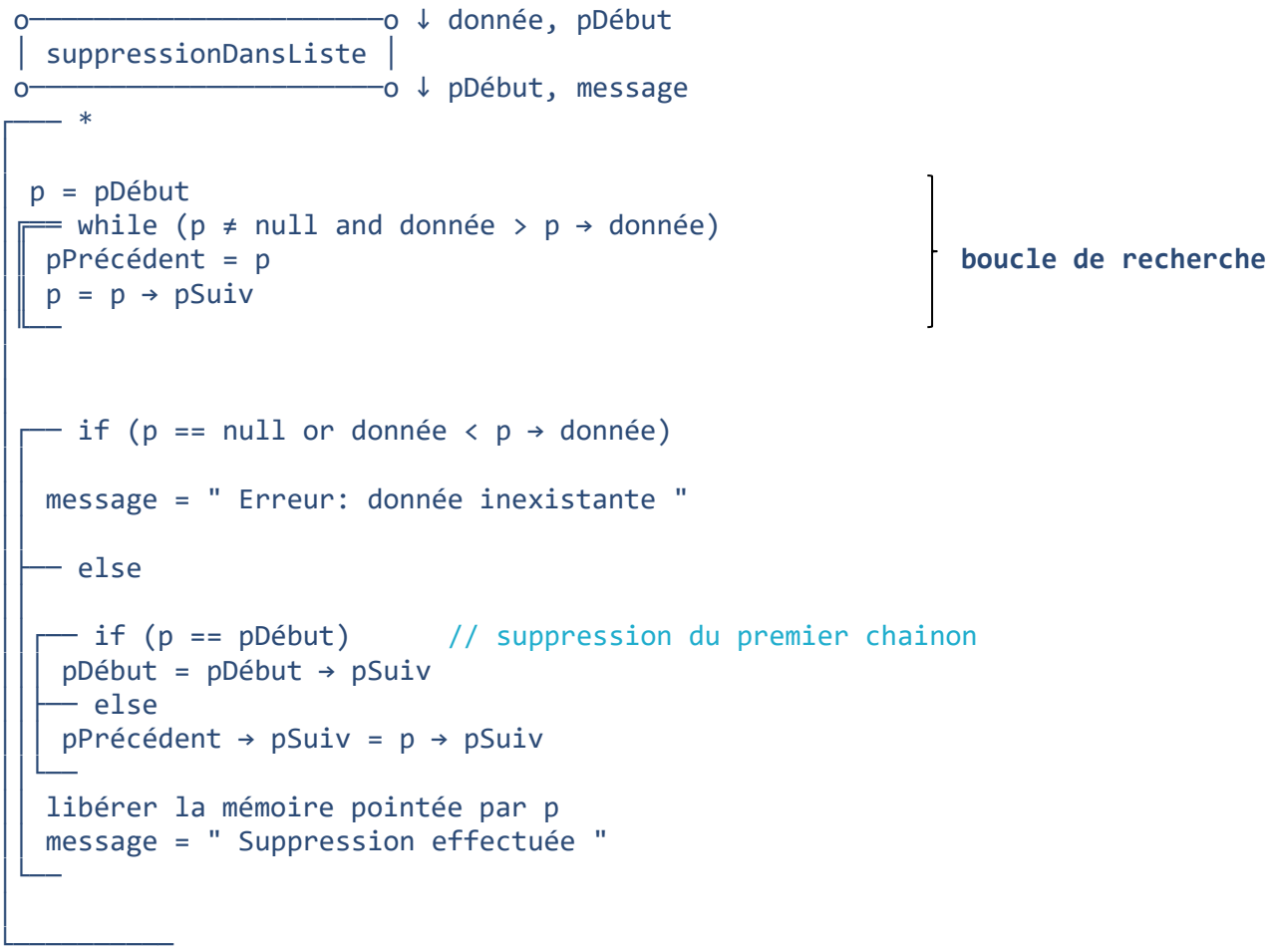
```
o──────────────────────────o ↓ pDébut, donnée
│ insertionDansListeTriée  │
o──────────────────────────o ↓ pDébut, message
┌── *
│ pNouv = adresse mémoire nouveau chainon
│┌─ if (pNouv == null)
│ message = "mémoire insuffisante"
│├─ else
│ p = pDébut
│╔═ while (p ≠ null and donnée > p → donnée)      ┐
│║ pPrécédent = p                                 ├ boucle de recherche
│║ p = p → pSuiv                                  ┘
│└─
│ pNouv → donnée = donnée
│
│┌─ if (p == pDébut)    // ajout début de liste ou liste vide
│ pNouv→ pSuiv = pDébut
│ pDébut = pNouv
│
│├─ else
│┌─ if (p == null)     // ajout en fin de liste
│ pPrécédent → pSuiv = pNouv
│ pNouv → pSuiv = null
│
│├─ else              // ajout milieu de liste
│ pPrécédent → pSuiv = pNouv
│ pNouv → pSuiv = p
│└─
│└─
│ message = « Ajout effectué »
└──
```

Nous pouvons encore synthétiser la partie déterminée par l'accolade en :

```
pNouv → pSuiv = p
┌─ if (p == pDébut)  // ajout début de liste ou liste vide
pDébut = pNouv
├─ else              // ajout milieu ou fin de liste
pPrécédent → pSuiv = pNouv
└─
```

## 3.3.4. Suppression d'un chainon dans une liste simple

Traitons également le cas d'erreur d'une donnée inexistante dans la liste. Supposons la liste triée. Si ce n'est pas le cas, seules la boucle de recherche et la condition qui la suit doivent être modifiées.



## 3.4. Comparaison des tableaux et des listes

| Avantages des listes | Inconvénients des tableaux |
|---|---|
| *Plus efficaces pour l'insertion et la suppression quand ces opérations doivent être effectuées un nombre élevé de fois* | *L'ajout et la suppression de données nécessitent des décalages* |
| *Allocation dynamique de la mémoire : très utile quand on ne connaît pas le volume des données ou quand ce volume varie très fort* | *Allocation statique de la mémoire* |

*Tableau 1*

| Inconvénients des listes | Avantages des tableaux |
|---|---|
| *Nécessité de parcourir la liste depuis le début chainon par chainon pour accéder à une donnée* | *Si on connaît l'indice, l'accès aux données est direct* |
| *Chaque chainon contient un pointeur vers le chainon suivant donc nécessite plus de place mémoire qu'une cellule de tableau* | *Si le volume des données est déterminé à l'avance, moins de place mémoire occupée avec les tableaux (car plus besoin de pointeur)* |

*Tableau 2*

# MODULE 4 : LES PILES ET LES FILES

## 4.1. Définition

Les piles et les files sont des structures de données dont **les éléments sont ordonnés linéairement** mais qui ne permettent l'accès qu'à un seul élément à la fois.

Elles permettent de stocker des informations en attente de traitement et de les récupérer dans un ordre précis (clarification des algorithmes).

Elles sont souvent associées à des algorithmes récursifs.

Dans une pile, on peut ajouter et supprimer des éléments suivant la règle du *dernier entré premier sorti (DEPS)*.

Le nom de pile vient d'une analogie avec une pile d'assiettes (par exemple) où on poserait toujours les assiettes sur le dessus de la pile et l'on prendrait toujours les assiettes sur le dessus de la pile.

Dans une file, on peut ajouter et supprimer des éléments suivant la règle du *premier entré premier sorti (PEPS).*

Le nom de file vient d'une analogie avec une file d'attente à un guichet, dans laquelle le premier arrivé sera le premier servi. Les usagers arrivent en queue de file et sortent de la file à sa tête.

Les piles et les files peuvent être implémentées par un tableau ou par une liste chainée[1]. Dans les deux cas, il est commode de réaliser sur ces structures des opérations de base qui seront décrites ci-dessous.

## 4.2. Pile

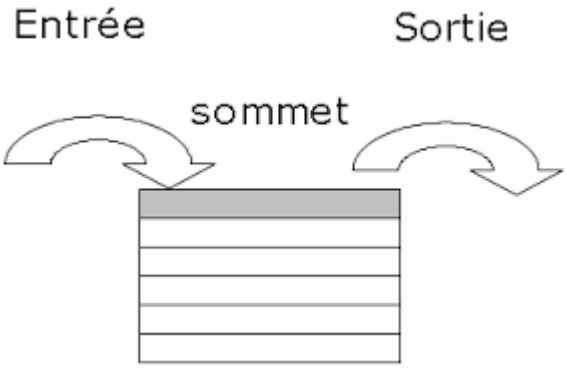
### 4.2.1. Définition

Une pile est un

- **ensemble d'éléments ordonnés linéairement**, initialement vide ;
- dont les éléments sont empilés les uns sur les autres ;
- et dont le seul élément accessible est celui du **sommet** de la pile.

Pour stocker et récupérer un élément dans une pile, on utilise la technique **LIFO** – Last In, First Out => le dernier élément inséré dans la pile est le premier à pouvoir être récupéré.

[1] Dans certains langages, comme en java par exemple, il existe des classes dédiées qui ont les méthodes adéquates.




https://www.editions-eni.fr

## 4.2.2. Opérations sur les piles

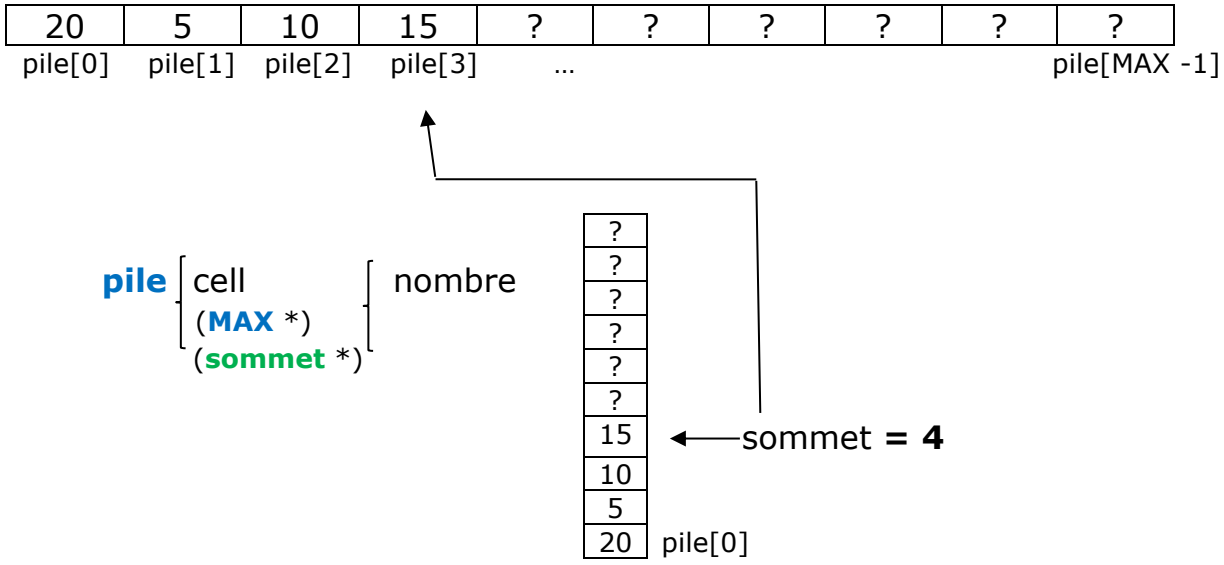
Les opérations de base à réaliser sur une pile sont :

- initialiser la pile ;
- vérifier si la pile est vide ;
- obtenir la valeur au sommet de la pile ;
- dépiler : retirer la valeur au sommet de la pile ;
- empiler : insérer une valeur au sommet de la pile (s'il reste de la place).

## 4.2.3. Représentation d'une pile

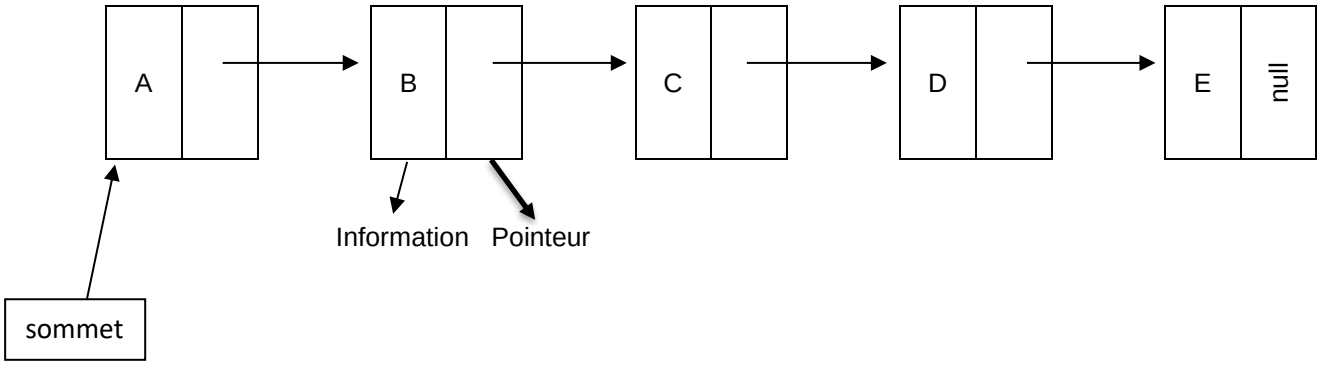
### 4.2.3.1. Tableau à simple indice

Une pile contenant au maximum *MAX* éléments de type numérique sera représentée comme suit :




Dans les algorithmes, la variable **sommet** contient le nombre d'éléments de la pile. **Max** est la taille physique de la pile et **sommet** sa taille logique.

#### 4.2.3.2. Liste chainée simple




Le premier chainon est pointé par **sommet**.

Dans les algorithmes, la variable **sommet** est un pointeur qui contient l'adresse du premier chainon.

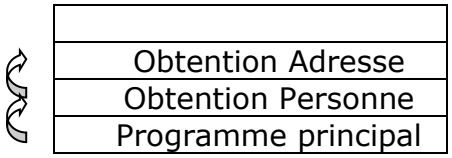
### 4.2.4. Exemple d'application : la pile d'appel

Dans la plupart des langages de programmation compilés, il y a une pile particulière dans laquelle sont poussées tout ou partie des informations concernant l'appel à des procédures ou fonctions :

- les paramètres d'appel ;
- l'adresse de retour ;
- un espace pour les variables locales.

Cette pile est formée de cadres de piles (stack frames) comprenant ces informations pour chaque procédure en cours d'appel imbriqué.

L'utilisation de piles permet donc aux processus de conserver la trace de leur situation courante pendant l'exécution d'une tâche secondaire, de telle façon qu'à la fin de cette tâche, ils puissent revenir à la tâche principale et continuer.



## 4.2.5. Algorithmes utilisant un tableau

**pile** [ cell (**MAX***) (sommet *) ] [ chaque cellule contient une donnée, par exemple alphanumérique ]

### 4.2.5.1 Initialisation

```
o──────────────────o
│ initialisation   │
o──────────────────o ↓ sommet
┌── *
│ //  pile = ARRAY(MAX)
│ sommet = 0
└──────────
```

### 4.2.5.2 Empiler

```
o──────────o ↓ pile, sommet, donnéeNouvelle
│ empiler  │
o──────────o ↓ pile, sommet, message
┌── *
│  message = " "
│┌── if (sommet < MAX)
│ pile[sommet] = donnéeNouvelle
│ sommet ++
│├── else
│ message = "la pile est pleine"
│└──
└──────────
```

### 4.2.5.3 Dépiler

```
o──────────o ↓ pile, sommet
│ dépiler  │
o──────────o ↓ pile, sommet, donnée, message
┌── *
│  message = " "
│┌── if (sommet > 0)
│ sommet --
│ donnée= pile[sommet]
│├── else
│ message = "erreur: la pile est vide"
│ donnée= " "
│└──
└──────────
```

## 4.2.6. Algorithmes utilisant une liste chainée simple


### 4.2.6.1 Initialisation

```
o──────────────────o
│ initialisation   │
o──────────────────o ↓ sommet
┌── *
│ sommet = null
└──────────
```

### 4.2.6.2 Empiler

```
o──────────o ↓ sommet, donnéeNouvelle
│ empiler  │
o──────────o ↓ sommet, message
┌── *
│  message = " "
│  pNouveau = adresse mémoire nouveau chainon
│ ┌── if (pNouveau ≠ null)
│ │ pNouveau → donnée = donnéeNouvelle
│ │ pNouveau → pSuiv = sommet
│ │ sommet = pNouveau
│ ├── else
│ │ message = "plus de place mémoire"
│ └──
└──────────
```

### 4.2.6.3 Dépiler

```
o──────────o ↓ sommet
│ dépiler  │
o──────────o ↓ sommet, donnée, message
┌── *
│  message = " "
│ ┌── if(sommet == null)
│ │ message = "erreur, la pile est vide"
│ │ donnée = " "
│ ├── else
│ │ donnée = sommet → donnée
│ │ pSauvé = sommet
│ │ sommet = sommet → pSuiv
│ │ libérer la place pointée par pSauvé
│ └──
└──────────
```

# 4.3. File

## 4.3.1. Définition

Une file est un

- **ensemble ordonné d'éléments**, initialement vide;
- les éléments sont insérés à une extrémité : la **queue**;
- ils sont récupérés à l'autre extrémité : la **tête**.

Les files sont également appelées listes FIFO (First In First Out) et fonctionnent comme les files d'attente : le premier arrivé est le premier servi.

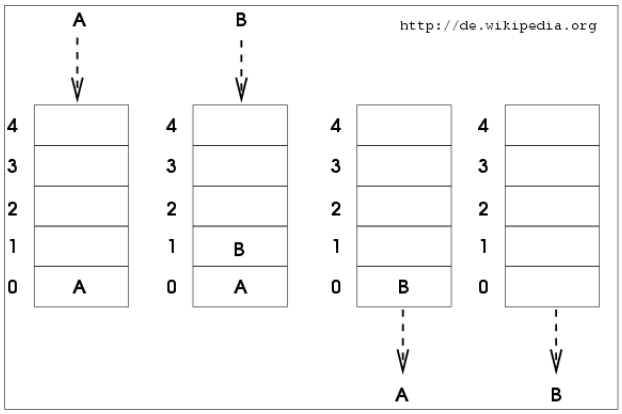



*Figure 1*

## 4.3.2. Opérations sur les files

Les opérations de base à réaliser sur une file sont :

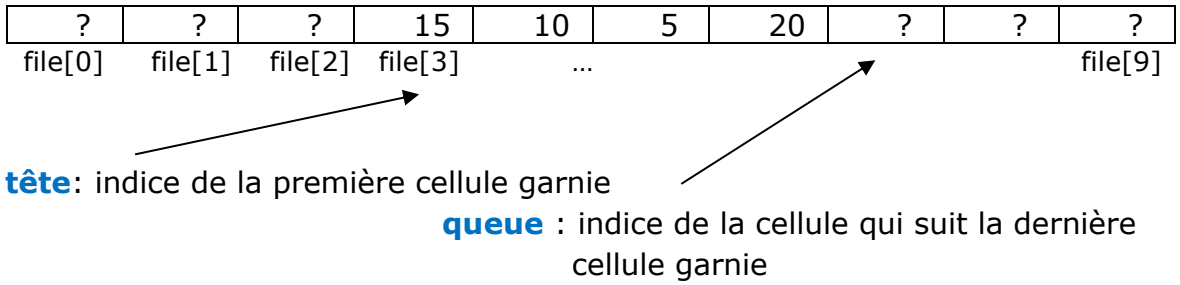
- initialiser la file;
- vérifier si la file est vide;
- accéderTête : obtenir la valeur de l'élément de tête;
- défiler : supprimer l'élément de tête;
- enfiler : ajouter un élément en queue de file.

## 4.3.3. Représentation d'une file

### 4.3.3.1. Tableau à simple indice

Une file contenant au maximum *MAX* éléments de type numérique sera représentée comme suit :

**file** [ **cell (MAX*)** [ **nombre**




Max est la taille physique de la file.
Deux options existent pour gérer le décalage des éléments vers la droite au cours du temps. La première option est de recommencer à enfiler à partir de la cellule d'indice 0 lorsqu'on est arrivé au bout du tableau (gestion circulaire). La deuxième est de décaler les cellules vers la gauche chaque fois qu'on défile un élément. De cette façon, la tête sera toujours l'indice 0. C'est cette deuxième option que nous utiliserons dans ce cours. Dans cette option, la taille physique de la file sera égale à **queue** - 1.

### 4.3.3.2. Liste chainée simple

Le pointeur **tête** est l'adresse du 1er chainon et le pointeur **queue**, l'adresse du dernier chainon, ou l'inverse.

## 4.3.4. Exemples d'application

- **Les serveurs d'impression**
  Ils doivent traiter les requêtes dans l'ordre où elles arrivent ; Ils les insèrent donc dans une file d'attente.
- **Les sémaphores**
  Une sémaphore est un dispositif permettant de synchroniser les accès aux ressources partagées.
  Lorsqu'un fichier est accessible pour plusieurs utilisateurs et que ce fichier est protégé par une sémaphore, durant chaque accès d'un utilisateur, une file d'attente est créée afin de déterminer qui sera le prochain à accéder au fichier.

- **Les systèmes d'événements en temps réel** (Windows et ses fenêtres) :
  mise en file d'évènements pouvant survenir sans que le système de gestion soit prêt.

# 4.3.5. Algorithmes utilisant un tableau

**file** [ cell (**MAX***) [ chaque cellule contient une donnée, par exemple alphanumérique

## 4.3.5.1 Initialisation

```
o──────────────o
│ initialisation │
o──────────────o ↓ (tête), queue
┌── *
│ //  file = ARRAY(MAX)
│ (tête = 0)
│ queue = 0
└──────────
```

## 4.3.5.2 Enfiler

```
o──────────o ↓ file, queue, donnéeNouvelle
│ enfiler │
o──────────o ↓ file, queue, message
┌── *
│ message = " "
│┌─ if (queue < MAX)
│ file[queue] = donnéeNouvelle
│ queue ++
│├─ else
│ message = "la file est pleine"
│└─
└──────────
```

## 4.3.5.3 Défiler

```
o──────────o ↓ file, (tête), queue
│ défiler │
o──────────o ↓ file, queue, donnée, message
┌── *
│ message = " "
│┌─ if (tête == queue) // ou  if( queue == 0)
│ message = "erreur, la file est vide"
│ donnée = " "
│├─ else
│ donnée = file[tête]  // ou donnée = file[0]
│ o──────────o ↓ file, queue
│ │ décalage │
│ o──────────o ↓ file, queue
│└─
└──────────
```

```
o──────────o ↓ file, queue
│ décalage │
o──────────o ↓ file, queue
┌─── *
│ ind = 0
│╔══ while (ind < queue - 1)
│╠═ file [ind] = file [ind + 1]
│╠═ ind ++
│└──
│ queue --
└──────────
```

## 4.3.6. Algorithmes utilisant une liste chainée simple

- **tête** (pointeur vers le premier chainon)


- **queue : pointeur vers le dernier chainon**

### 4.3.6.1 Initialisation

```
o──────────────────o
│ initialisation   │
o──────────────────o ↓ tête, queue
┌─── *
│ tête = queue = null
└──────────
```

### 4.3.6.2 Enfiler

```
o──────────o ↓ tête, queue, donnéeNouvelle
│ enfiler  │
o──────────o ↓ tête, queue, message
┌─── *
│ message = " "
│ pNouveau = adresse mémoire nouveau chainon
│┌── if (pNouveau ≠ null)
│├ pNouveau → donnée = donnéeNouvelle
│├ pNouveau → pSuiv = null
│├
│├── if (tête == null) // ajout dans une file vide
│├ tête = queue = pNouveau
│├── else
│├ queue → pSuiv = pNouveau
│├ queue = pNouveau
│└──
│├── else
│├ message = "plus de place mémoire"
│└──
└──────────
```

### 4.3.6.3 Défiler

```
o──────────o ↓ tête, queue
│ défiler  │
o──────────o ↓ tête, queue, donnée, message
┌── *
│  message = " "
│┌── if (tête == null)
│ message = "erreur, la file est vide"
│ donnée = " "
│├── else
│ donnée = tête → donnée
│ pSauvé = tête
│ tête = tête→ pSuiv
│ libérer la place pointée par pSauvé
│┌── if (tête == null) // la file ne contenait qu'un seul chainon
│ queue = null
│└──
└──
└──────────
```

# MODULE 5 : LES ARBRES

## 5.1. Présentation

Un arbre est une structure de données non linéaire dans laquelle les informations sont retenues dans ce qu'on appelle des **nœuds**. Les nœuds sont organisés de manière hiérarchique et liés entre eux par des arcs ou arêtes)

Exemples : arbre généalogique, organigramme d'une société, tournoi de tennis…

Illustration

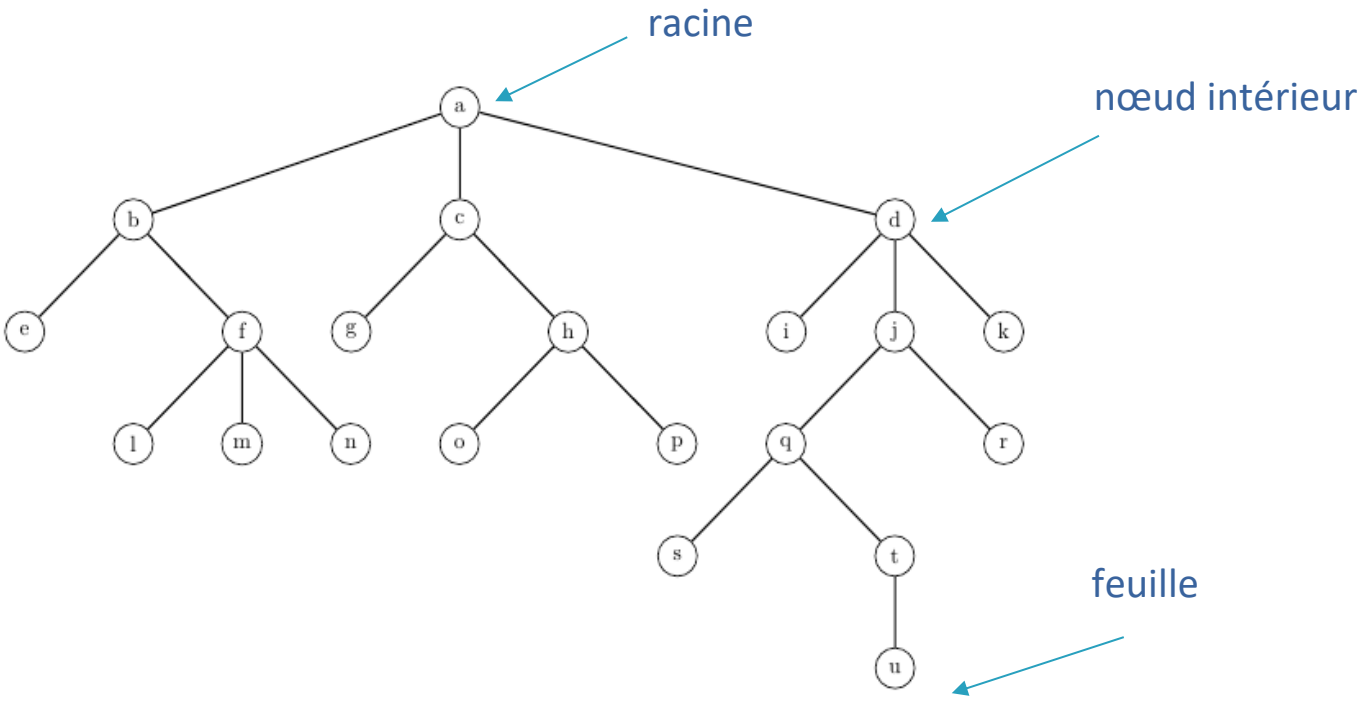



*Figure 2*

Terminologie

- L'arborescence débute par un nœud particulier appelé **racine** ; c'est le seul nœud à ne pas avoir de père ;
- Tout nœud autre que la racine possède un seul **père** ;
- Tout nœud peut posséder des descendants : ses **fils** ; les fils peuvent eux-mêmes être pères ;
- Les nœuds qui terminent l'arborescence sont appelés **feuilles** ou **nœuds terminaux** ; une feuille ne possède pas de fils ;
- Un **nœud intérieur** est un nœud qui a un père et au moins un fils ;
- Tout nœud de l'arbre est la racine d'un **sous-arbre** constitué par sa descendance et lui-même. Un arbre est donc une **structure récursive**.

## 5.2. Définitions

Un arbre non vide est un ensemble fini d'un ou plusieurs nœuds
- dont un nœud particulier appelé la racine ;
- dont les nœuds restants sont partitionnés en ensembles disjoints qui sont eux-mêmes des arbres que l'on appelle les sous-arbres de la racine.

Un arbre est qualifié de connexe car la séquence des pères partant d'un nœud vers la racine est toujours unique. En effet, il faut pour cela partir de n'importe quel nœud *n* autre que la racine et se déplacer vers le père de *n* puis vers le père du père du *n* et ainsi de suite pour atteindre la racine de l'arbre.

Définitions associées

- chemin : suite (unique) d'arcs à parcourir entre 2 nœuds
- longueur d'un chemin : nombre d'arcs d'un chemin
- niveau d'un nœud : longueur du chemin depuis la racine jusqu'à ce nœud
- hauteur d'un nœud : longueur du plus long chemin depuis ce nœud jusqu'à une feuille
- hauteur de l'arbre : hauteur de la racine

Ainsi pour l'arbre de la figure 2 ci-dessus, le nœud :
- a a une hauteur de 5
- c a une hauteur de 2
- o a une hauteur de 0
- a a une profondeur de 0
- p a une profondeur de 3

- degré extérieur d'un nœud : nombre de sous-arbres du nœud
- degré de l'arbre : degré extérieur maximum des nœuds de l'arbre

Ainsi pour l'arbre de la figure 2 ci-dessus, le nœud :
- a a un degré extérieur de 3
- c a un degré extérieur de 2
- t a un degré extérieur de 1

et l'arbre a un degré de 3.

Attention, le degré de l'arbre ne correspond pas nécessairement au degré de la racine.

Un arbre est qualifié d'**ordonné** si l'ordre des sous-arbres est significatif.

## 5.3. Applications

- Interfaces utilisateurs : chaque interface (fenêtre), sauf la première, a un parent à partir duquel elle est lancée, et peut avoir plusieurs filles.
- Répertoires : structure d'organisation des dossiers et sous-dossiers.

- Traitement arithmétique par les compilateurs :
  ((7-4)/2)*(5+6)

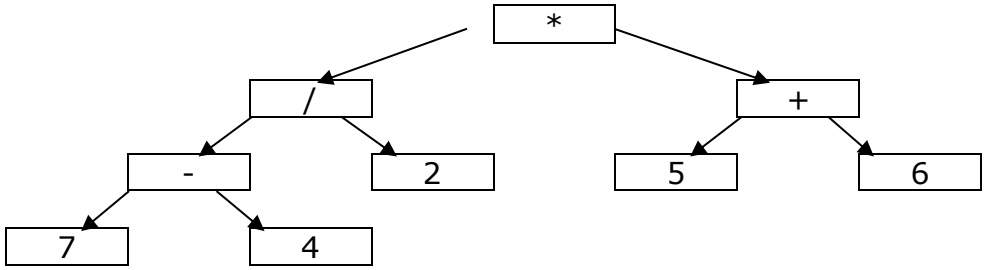



*Figure 3*

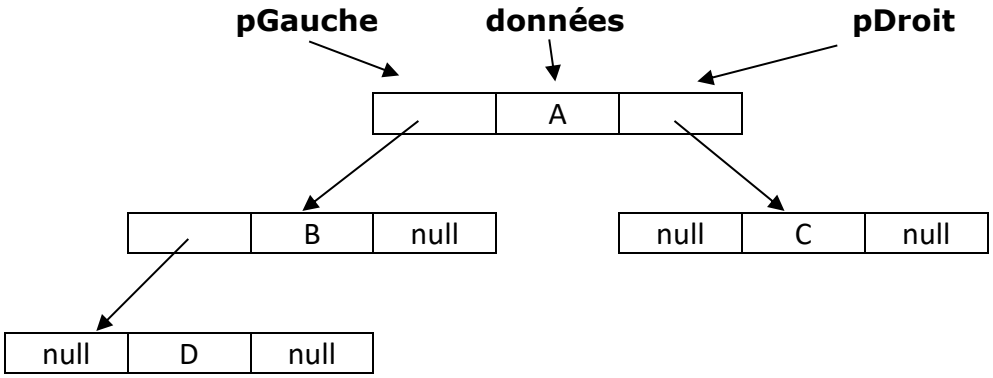
- Intelligence Artificielle : problèmes complexes comme les jeux d'échecs par exemple.
  Un nœud = un état = point à partir duquel on prend une décision.
  Sous-arbre = conclusion(s) dérivées(s) d'une suite de décisions.
- Files avec priorité gérées avec des tas (arbres binaires).
  Exemple : plusieurs serveurs sur un réseau. La racine sera le serveur qui peut accepter la plus lourde charge.

## 5.4. Arbre binaire

### 5.4.1. Définition

Un **arbre binaire** est un arbre ordonné de degré extérieur 2. Chaque nœud a donc au plus deux fils. On distingue le fils gauche du fils droit.

Représentation




Chaque nœud est constitué de :

- **pGauche** : pointeur vers le fils gauche ou null si pas de fils gauche
- **données :** structure contenant des données (lettre dans l'exemple ci-dessus)
- **pDroit** : pointeur vers le fils droit ou null si pas de fils droit.

## 5.4.2. Parcours d'un arbre binaire

Parcourir un arbre binaire signifie visiter tous les nœuds afin d'effectuer un traitement sur chaque nœud.
Il existe 3 parcours possibles qui diffèrent par l'ordre dans lequel on traite les nœuds.
Pour les 3 parcours, on commence à la racine et on parcourt l'arbre en visitant tous les nœuds dans le sens contraire des aiguilles d'une montre.

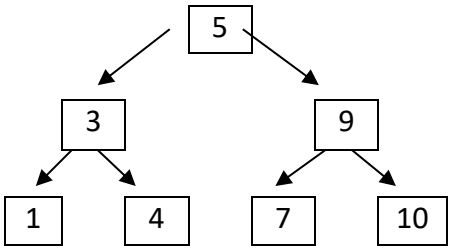



*Figure 4*

**Parcours préfixe ou pré-ordre**

On traite les nœuds dans l'ordre : **R**acine **G**auche **D**roit
On traite le nœud quand on le rencontre pour la 1ère fois.

> Exemple de la figure 3 :
> Ordre des nœuds visités : 5 3 1 3 4 3 5 9 7 9 10 9 5
> Ordre de traitement des nœuds : 5 3 1 4 9 7 10

**Parcours suffixe, postfixe ou post-ordre**

On traite les nœuds dans l'ordre : **G**auche **D**roit **R**acine
On traite le nœud quand on le rencontre pour la dernière fois.

> Exemple de la figure 3 :
> Ordre des nœuds visités : 5 3 1 3 4 3 5 9 7 9 10 9 5
> Ordre de traitement des nœuds : 1 4 3 7 10 9 5

**Parcours infixe ou in-ordre**

On traite les nœuds dans l'ordre : **G**auche **R**acine **D**roit
On traite le nœud entre le traitement de ses deux fils.

> Exemple de la figure 3 :
> Ordre des nœuds visités : 5 3 1 3 4 3 5 9 7 9 10 9 5
> Ordre de traitement des nœuds : 1 3 4 5 7 9 10

Exercice : Quel est le parcours qui va convenir pour le traitement de l'expression arithmétique ((7-4)/2)*(5+6) illustrée dans la figure 3 page 38 ?

# 5.5. Arbre binaire de recherche (ABR)

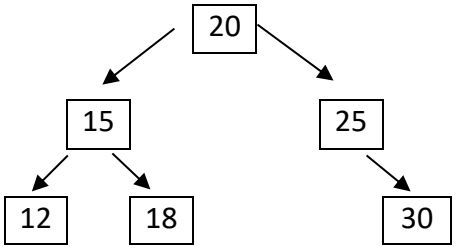
## 5.5.1. Définition

Un **arbre binaire de recherche** ( ou **ABR**) est un arbre binaire éventuellement vide.

S'il n'est pas vide, il vérifie les caractéristiques suivantes :

- parmi les données des nœuds, se trouve une clé unique qui identifie chaque nœud de manière univoque
- pour chaque nœud :
  - les clés d'un sous-arbre gauche non vide sont inférieures à la clé de ce nœud
  - les clés d'un sous-arbre droit non vide sont supérieures à la clé de ce nœud

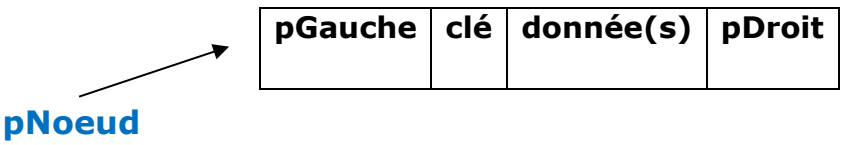
**Exemple** : On construit un arbre binaire de recherche en ajoutant successivement dans l'arbre les nœuds dont les clés sont les suivantes : 20, 15, 18, 25, 12 et 30.




Remarque : la structure de l'arbre dépend de l'ordre dans lequel les nœuds ont été ajoutés.

## 5.5.2. Notations et opérations

Pour la suite du cours, nous schématiserons un nœud d'un ABR de la manière suivante :




Nous noterons :

- **pNoeud** : pointeur vers un nœud quelconque;
- **pNoeudNouv :** pointeur vers un nouveau nœud **;**
- **pNoeud -> clé :** clé du nœud pointé par pNoeud ;
- **pNoeud -> donnée(s) :** donnée(s) du nœud pointé par pNoeud ;
- **pNoeud -> pGauche :** adresse du fils gauche de pNoeud ;
- **pNoeud -> pDroit :** adresse du fils droit de pNoeud ;
- **racine :** adresse du nœud qui est à la racine.

Dans la pratique, pour des raisons de clean code, nous utiliserons un nom de pointeur plus explicite c'est-à-dire illustrant le contenu du nœud. Par exemple : pNoeudClient, pNoeudClientNouv, pNoeudLivre , racineLlivres …

Les différentes opérations que l'on peut pratiquer sur un ABR et que nous étudierons dans la suite sont :

- recherche d'un nœud ;
- ajout d'un nouveau nœud ;
- suppression d'un nœud ;
- parcours itératif d'un arbre ;
- parcours récursif d'un arbre.

## 5.5.3. Algorithmes

### 5.5.3.1. Recherche dans un arbre binaire de recherche

```
o──────────────────────o ↓ racine, cléRecherchée
│ rechercheDansArbre   │
o──────────────────────o
┌── *
│ o──────────────────o ↓ racine, cléRecherchée
│ │ pNoeudRecherché  │
│ o──────────────────o ↓ pNoeud, père
│ ┌── if( pNoeud == null)
│ │  sortir "clé absente de l'arbre"
│ ├── else
│ │  sortir "clé trouvée:", pNoeud → donnée
│ └──
└────────────
```

```
o──────────────────o ↓ racine, cléRecherchée
│ pNoeudRecherché  │
o──────────────────o ↓ pNoeud, père
┌── *
│ père = null
│ pNoeud = racine
│ ╔══ while (pNoeud ≠ null and cléRecherchée ≠ pNoeud → clé)
│ ╠  père = pNoeud
│ ╠  ┌── if(cléRecherchée < pNoeud → clé)
│ ╠  │  pNoeud = pNoeud → pGauche
│ ╠  ├── else
│ ╠  │  pNoeud = pNoeud → pDroit
│ ╠  └──
│ ╚══
└────────────
```

Remarquons que le module ***pNoeudRecherché*** renvoie la variable ***père***. Cette variable n'est pas nécessaire dans le cas d'une recherche mais sera indispensable lorsque cette recherche sera suivie d'un ajout ou d'une suppression de nœud.

### 5.5.3.2. Ajout d'un nouveau nœud dans un arbre binaire de recherche

Dans un ABR, l'ajout d'un nouveau nœud se fera toujours en tant que feuille. Si l'arbre est vide, le nouveau nœud devient la racine de l'arbre.

```
o──────────────o ↓ racine, cléNouvelle, donnéeNouvelle
│ ajoutDansABR │
o──────────────o ↓ racine, message
┌── *
│ message = " "
│ o──────────────────o ↓ racine, cléNouvelle
│ │ pNoeudRecherché  │
│ o──────────────────o ↓ pNoeud, père
│┌── if (pNoeud ≠ null)
│ message = "erreur, la clé est déjà présente dans l'arbre"
│├── else
│ pNoeudNouv = adresse mémoire nouveau nœud
│ ┌── if (pNoeudNouv == null)
│ │ message = "Plus de place mémoire"
│ ├── else
│ │ // garnir le nouveau nœud ***
│ │ pNoeudNouv → clé = cléNouvelle
│ │ pNoeudNouv → donnée = donnéeNouvelle
│ │ pNoeudNouv → pGauche = null
│ │ pNoeudNouv → pDroit = null
│ │ // attacher le noeud à l'arbre ***
│ │ ┌── if(racine == null) // arbre vide
│ │ │ racine = pNoeudNouv
│ │ ├── else
│ │ │ ┌── if(cléNouvelle < père → clé)
│ │ │ │ père → pGauche = pNoeudNouv
│ │ │ ├── else
│ │ │ │ père → pDroit = pNoeudNouv
│ │ │ └──
│ │ └──
│ └──
└──────────
```

### 5.5.3.3. Suppression d'un nœud dans un arbre binaire de recherche

Il existe plusieurs techniques de suppression d'un nœud dans un arbre.

La technique étudiée dans ce syllabus (technique du **déplacement**) est la suivante :

- Si le nœud à supprimer est la racine de l'arbre et que la racine est une feuille (un seul nœud dans l'arbre), la racine devient **null**.
- Sinon, si le nœud à supprimer est une feuille, il faut mettre le pointeur correspondant de son père à **null** et libérer la mémoire pointée par le nœud.
- Sinon, si le nœud à supprimer ne possède pas de fils droit, il faut accrocher le sous-arbre gauche du nœud à supprimer au père du nœud à supprimer si ce père existe sinon le sous-arbre gauche devient la racine. Il faut ensuite libérer la mémoire pointée par le nœud.
- Sinon, le nœud a un fils droit et éventuellement un fils gauche. Il faut accrocher le sous-arbre droit du nœud à supprimer au père du nœud à supprimer si ce père existe sinon le sous-arbre droit devient la racine. Il faut ensuite, s'il existe, accrocher le sous-arbre gauche du nœud à supprimer à la gauche du plus petit nœud du sous-arbre droit. Il faut enfin libérer la mémoire pointée par le nœud.

Une autre technique consiste à **remplacer** les données du nœud à supprimer, si celui-ci n'est pas une feuille, par les données du nœud de son sous-arbre droit dont la clé est directement supérieure à celle du nœud à supprimer. Si le nœud à supprimer n'a pas de sous-arbre droit, on utilise le nœud du sous-arbre gauche dont la clé est directement inférieure à celle du nœud à supprimer.
La suppression d'une feuille se fait de la même façon que dans la première technique de suppression. Cette deuxième technique sera vue en laboratoire.

On peut aussi rencontrer différentes variantes de ces deux techniques.

```
o──────────────────────────────────o ↓ racine, cléASupprimer
│ suppressionNœudParDéplacement │
o──────────────────────────────────o ↓ racine, message
┌── *
│ o──────────────────o ↓ racine, cléASupprimer
│ │ pNoeudRecherché │
│ o──────────────────o ↓ pNoeud, père
│ // voir point 5.5.3.
│
│ ┌── if (pNoeud == null)
│ │
│ │ message = "clé absente de l'arbre"
│ │
│ ├── else
│ │
│ │ ┌── if (pNoeud → pDroit == null)     // le noeud n'a pas de fils droit
│ │ │
│ │ │ ┌── if(pNoeud == racine)
│ │ │ │ racine = racine → pGauche
│ │ │ ├── else
│ │ │ │ ┌── if(cléASupprimer < père → clé)
│ │ │ │ │ père → pGauche = pNoeud → pGauche
│ │ │ │ ├── else
│ │ │ │ │ père → pDroit = pNoeud → pGauche
│ │ │ │ └──
│ │ │ └──
│ │ ├── else                              // le noeud a un fils droit
│ │ │
│ │ │ ┌── if(pNoeud == racine)
│ │ │ │ racine = racine → pDroit
│ │ │ ├── else
│ │ │ │ ┌── if(cléASupprimer < père → clé)
│ │ │ │ │ père → pGauche = pNoeud → pDroit
│ │ │ │ ├── else
│ │ │ │ │ père → pDroit = pNoeud → pDroit
│ │ │ │ └──
│ │ │ └──
│ │ │
│ │ │ ┌── if( pNoeud → pGauche ≠ null)
│ │ │ │ pNoeudAccroche = pNoeud → pDroit
│ │ │ │ ╔══ while (pNoeudAccroche → pGauche ≠ null)
│ │ │ │ ╚═ pNoeudAccroche = pNoeudAccroche → pGauche
│ │ │ │ // pNoeudAccroche est le plus petit noeud du sous-arbre droit
│ │ │ │
│ │ │ │ pNoeudAccroche  → pGauche = pNoeud → pGauche
│ │ │ └──
│ │ └──
│ │ libérer la mémoire pointée par pNoeud
│ └──┘
└──────────
```

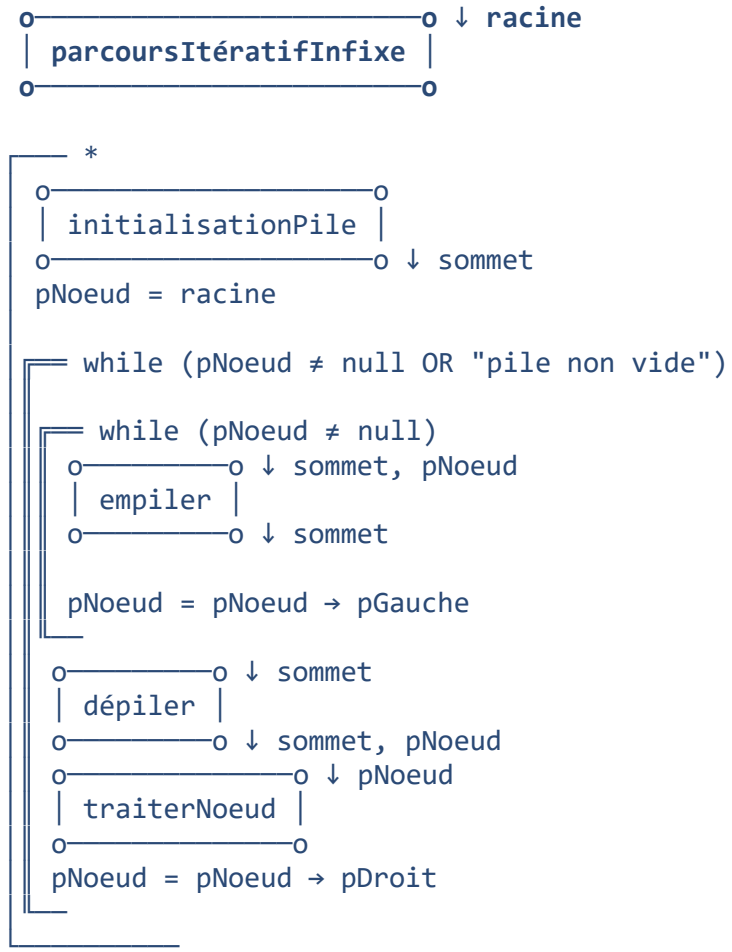
### 5.5.3.4. Parcours itératifs d'un arbre binaire de recherche

Dans le point 5.4.2., sont décrites les trois façons de parcourir l'arbre. Ces trois façons diffèrent par l'ordre de traitement des nœuds et sont illustrées par les trois diagrammes ci-dessous.

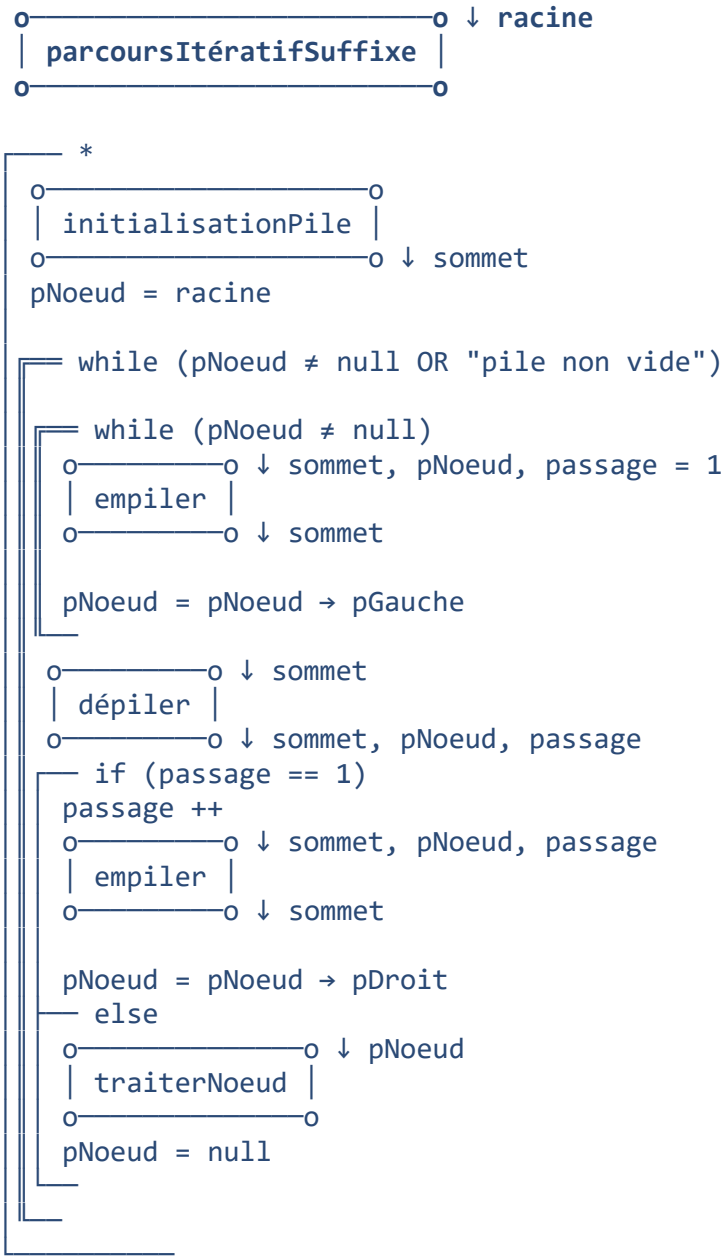
Les parcours itératifs nécessitent l'utilisation d'une pile. Les modules *initialisationPile, empiler* et *dépiler* ainsi que la condition *« pile non vide »* utilisés dans les parcours ci-dessous devront donc être adaptés selon que l'on gère la pile avec un tableau ou avec une liste chainée.

**Parcours préfixe**

**Parcours infixe**

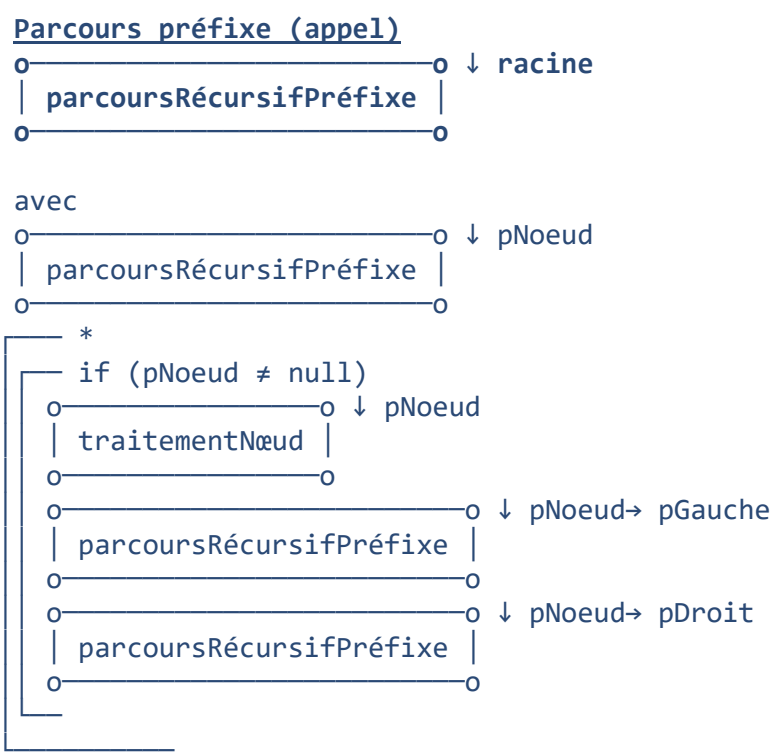


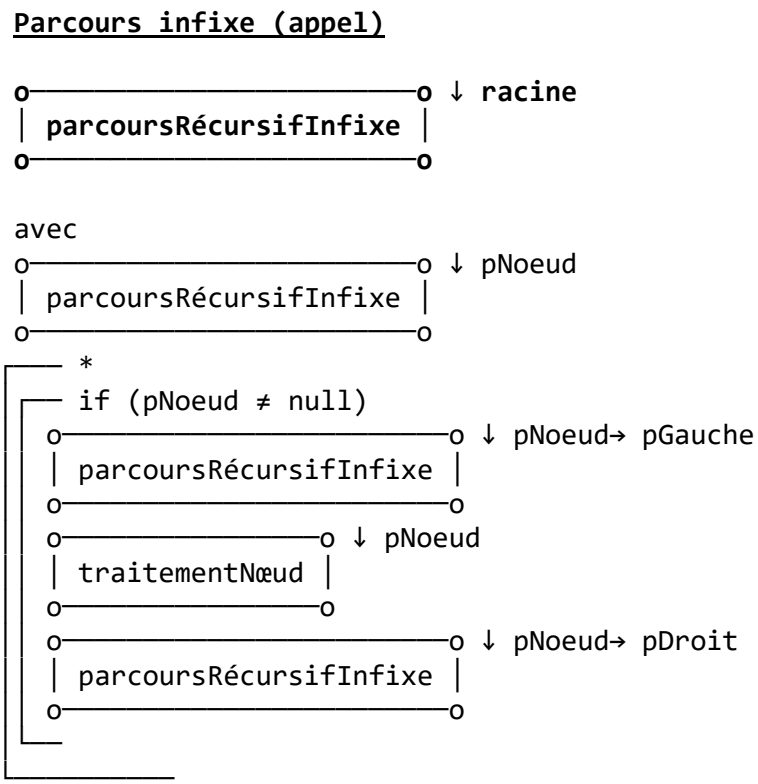
**Parcours suffixe**

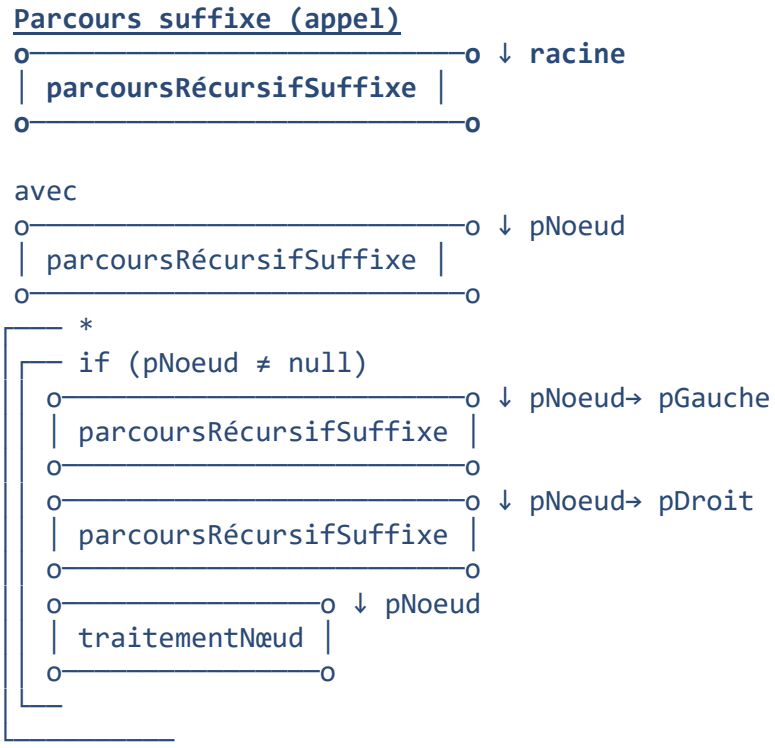


### 5.5.3.5. Parcours récursifs d'un arbre binaire de recherche

Les trois parcours sont illustrés par les trois diagrammes ci-dessous.




Parcours suffixe (appel)
parcoursRécursifSuffixe
racine
avec
parcoursRécursifSuffixe
pNoeud
*
if (pNoeud ≠ null)
parcoursRécursifSuffixe
pNoeud→ pGauche
parcoursRécursifSuffixe
pNoeud→ pDroit
traitementNœud
pNoeud