Université Mohamed Seddik BENYAHIA – JIJEL –

Faculté des Sciences Exactes et Informatique

Département d'Informatique

PROGRAMMATION ET STRUCTURES DE DONNEES 2

Cours et Exercices Corrigés

Auteur : Dr. KARA Messaoud

Septembre 2020

Table des matières

AVANT PROPOS

Ce cours est le résultat de 5 années d'enseignement du module Programmation et Structures de Données (Algorithmique 1 et 2) au département MI (Socle Commun Mathématiques et Informatique) de l'université de Jijel.

Les objectifs de ce cours :

1- Acquérir les notions de base de la programmation modulaire
2- La récursivité
3- Comprendre les structures de données et leur utilité.
4- Approfondir ces connaissances par des applications données à travers des exercices corrigés dont la majeure partie était une partie d'un sujet d'interrogation, d'examen ou d'examen de rattrapage. Ils ont été étendus pour traiter d'autres aspects qui n'étaient pas traités en examen à cause du temps alloué qui est limité.

Le document est organisé en trois parties :

1- La première partie est consacrée aux cours sur la programmation modulaire (procédures, fonctions, récursivité), les structures de données dynamiques (Listes linéaires chaînées, Piles, Files et une brève introduction sur les arbres).
2- La deuxième partie est consacrée aux exercices corrigés avec des brefs éléments d'analyse.
3- La dernière partie donne les séries de TD et de TP utilisées en séances de TD et en salle machines pour les TPs. Les séries de TP contiennent un cours sur le langage C permettant aux étudiants d'avoir les notions nécessaires pour résoudre les exercices de la série de TP et, si nécessaire, pouvoir traduire les solutions données en TD (sous forme d'algorithmes) en un programme en langage C.

PARTIE 1– PROGRAMMATION MODULAIRE

1. Introduction

Un programme (algorithme) écrit en un seul bloc devient difficile à comprendre dès qu'il dépasse un nombre de lignes. Pour éviter ce problème, la programmation structurée offre deux outils : les procédures et les fonctions. Elles permettent de décomposer le problème en sous problèmes plus facile à écrire, comprendre et corriger si nécessaire.

La recherche de la solution du problème sera d'identifier au niveau du problème posé un ou plusieurs sous problèmes à résoudre séparément. Chaque sous problème est à son tour traité comme un nouveau problème.

Ainsi la solution d'un problème est décrite par un algorithme principal qui définit la méthode générale de résolution et des sous algorithmes. Un sous algorithme est un algorithme qui décrit la solution d'un sous problème.

Chaque procédure ou fonction est utilisée pour résoudre un sous problème du problème global posé. Elles sont définies dans la partie déclarative d'un algorithme et ont la même structure d'un algorithme. Elles sont constituées d'un entête, de déclarations de variables et d'un corps.

Les objectifs de l'utilisation des procédures et des fonctions sont :

1. **La simplification** : rendre l'écriture des algorithmes plus simple car chaque sous algorithme est traité séparément.
2. **La réutilisation** : Ecrire une procédure (ou une fonction) une seule fois puis la réutiliser plusieurs fois au lieu de la réécrire plusieurs fois.
3. **Faciliter la maintenance et l'évolution des algorithmes (programmes)** : En cas d'erreurs ou pour améliorer un algorithme (programme), on ne modifie qu'une partie de l'algorithme sans toucher les autres parties.

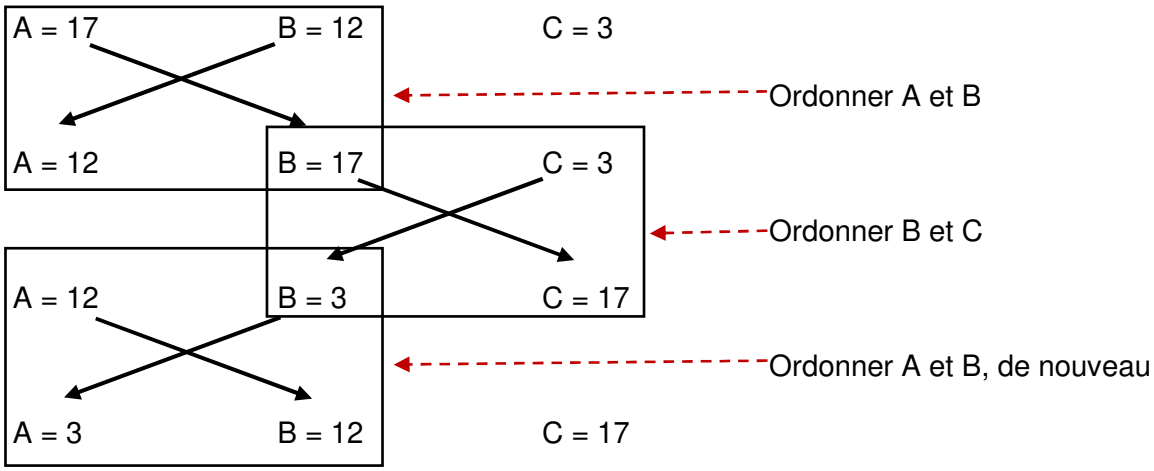
Rappel de la structure générale d'un algorithme

```
Algorithme nomAlgorithme
Const Liste_des_constantes
Type  Liste_des_types
Var   Liste_des_variables
      Liste_des_procédures_et_fonctions
Début
|     Liste des instructions        // Corps de l'algorithme (principal)
Fin
```

Exemple d'introduction

En s'inspirant de l'algorithme de tri à bulles, écrire un algorithme qui permet d'ordonner dans l'ordre croissant trois entiers A, B et C.

Exemple numérique : Si A = 17 et B = 12 et C = 3 alors A, B et C ne sont pas ordonnés dans l'ordre croissant. On commence par ordonner A et B, puis ordonner B et C et enfin ordonner A et B de nouveau. On a, en fait, trois fois la même opération qui consiste à ordonner deux entiers.



A la fin, on obtient, **A = 3** , **B = 12** et **C = 17**.

L'algorithme suivant permet de retranscrire les opérations expliquées ci-dessus :

```
Algorithme TriCroissantVersion1
Var A, B, C, Tmp : Entier
Début
    Ecrire("Donner trois entiers A, B et C")
    Lire( A, B, C )
    // Ordonner A et B.
    Si A > B Alors          // Si A et B ne sont pas dans l'ordre croissant.
        Tmp ← A
        A ← B               // Permuter les valeurs de A et B.
        B ← Tmp
    FSi
    Si B > C Alors          // Ordonner B et C.
        Tmp ← B
        B ← C
        C ← Tmp
    FSi
    Si A > B Alors          // Ordonner A et B de nouveau.
        Tmp ← A
```

```
        A ← B
        B ← Tmp
    FSi
    Ecrire("Les trois entiers dans l'ordre : ", A, B, C)
Fin
```

2. Les procédures

2.1- Définition d'une procédure

Une procédure décrit une suite d'actions bien identifiée, qui s'exerce sur un ou plusieurs paramètres (arguments) à travers lesquels la procédure reçoit les données et transmet les résultats des traitements qu'elle réalise.

2.2- Déclaration d'une procédure

Une procédure est déclarée selon la syntaxe suivante :

```
Procédure nomProcedure([Liste des paramètres formels])          // L'entête
[Var Liste de variables]          // Déclarations des variables locales, si nécessaire.
Début
    Liste des instructions        // Corps de la procédure
Fin
```

- [Liste des paramètres formels] : c'est une suite de déclarations de variables de la forme : nomParamètre1 : Type1 ; nomParamètre2 : Type2 ; … ; nomParamètreN :TypeN

Remarque : Si un paramètre est modifié par la procédure, il est précédé par le mot clé **VAR**.

Exemple : Procédure qui permet d'ordonner dans l'ordre croissant deux entiers A et B.

```
Procédure Ordonner( Var A : Entier ; Var B : Entier )
Var Tmp : Entier
Début
    Si A > B Alors // Si les valeurs de A et B ne sont pas dans l'ordre croissant
        Tmp ← A
        A ← B          // Permuter les valeurs de A et B
        B ← Tmp
    FSi
Fin
```

2.3- Appel d'une procédure

Dans un algorithme, une procédure joue le rôle d'une instruction. Elle est appelée selon la syntaxe suivante : **nomProcedure ([liste des paramètres effectifs])**

L'appel de la procédure provoque son exécution avec les paramètres effectifs.

La liste des paramètres effectifs doit avoir le même nombre, même ordre et les mêmes types que les paramètres formels (Exception : On peut mettre un Entier à la place d'un Réel).

Exemple : Pour appeler la procédure Ordonner sur deux variables X et Y on écrit l'instruction suivante : **Ordonner**(X, Y)

L'algorithme TriCroissantVersion1, donné en introduction, peut être réécrit comme suit :

```
Algorithme TriCroissantVersion2
Var A, B, C : Entier
Procédure Ordonner( Var A : Entier ; Var B : Entier )
Var Tmp : Entier
Début
    Si A > B Alors          // Si les valeurs de A et B ne sont pas dans l'ordre croissant
        Tmp ← A
        A ← B               // Permuter les valeurs de A et B
        B ← Tmp
    FSi
Fin
Début
    Ecrire("Donner trois entiers A, B et C")
    Lire( A, B, C )
    Ordonner( A, B)         // 1er appel de procédure
    Ordonner( B, C)         // 2e appel de procédure
    Ordonner( A, B)         // 3e appel de procédure
    Ecrire("Les trois entiers dans l'ordre : ", A, B, C)
Fin
```

Dans cette nouvelle version de l'algorithme, nous avons écrit la procédure Ordonner une seule fois et nous l'avons utilisé (appelé) trois fois.

3. Les fonctions

3.1- Définition d'une fonction

Une fonction peut être considérée comme un opérateur non primitif (complexe) créé à l'initiative du programmeur. Elle décrit une suite d'actions bien identifiées qui s'exerce sur un ou plusieurs paramètres et réalise le calcul d'une expression et par conséquent elle retourne un résultat unique.

Exemple : La fonction puissance $X^n = X * X * \ldots * X$ est un opérateur qui fait N opérations de multiplication. Pareil pour la factorielle : $N ! = N * (N - 1) * \ldots * 2 * 1$

3.2- Déclaration d'une fonction

Une fonction est déclarée selon la syntaxe suivante :

```
Fonction nomFonction([Liste des paramètres formels]) : TypeRésultat        // L'entête
[Var Liste de variables]                  // Déclarations des variables locales, si nécessaire.
Début
        Liste des instructions            // Corps de la fonction
        nomFonction ← résultat
Fin
```

Remarque : La dernière instruction d'une fonction est une affectation. Le nom de la fonction (nomFonction) reçoit le résultat calculé.

Exemple : Ci-dessous une fonction (Facto) qui calcule la factorielle d'un entier positif N.

```
Fonction Facto( N : Entier ) : Entier
Var I, F : Entier
Début
        F ← 1
        Pour I ← 2 à N Faire          // OU Pour I ← 1 à N Faire
                F ← F * I
        Fpour
        Facto ← F
Fin
```

3.3- Appel d'une fonction

Une fonction est utilisée (appelée) dans une expression. Son appel provoque son exécution avec les paramètres effectifs définis dans l'appel et renvoie un résultat.

Comme pour une procédure, la liste des paramètres effectifs d'une fonction doit avoir le même nombre, le même ordre et les mêmes types que les paramètres formels.

Exemple :

Pour appeler la fonction facto sur un entier X, on écrit par exemple : Y ← Facto(X)

Remarque : On peut trouver un appel de fonction :

dans une affectation (dans un calcul) : Exemple : T ← 1 + Facto(X) / Y + Z

à la place d'un paramètre non précédé du mot clé Var,

dans une opération d'écriture : Exemple : Ecrire(Facto(X)).

En fait, on peut mettre un appel de fonction, là où on peut utiliser une valeur constante.

Exemple : En utilisant la fonction Facto, on peut écrire un algorithme qui permet de lire un entier positif N et d'afficher sa factorielle comme suit :

```
Algorithme Factorielle
Var N : Entier
Fonction Facto( N : Entier ) : Entier
Var I, F : Entier
Début
    F ← 1
    Pour I ← 2 à N Faire
        F ← F * I
    Fpour
    Facto ← F
Fin
Début
    Répéter
        Ecrire("Donner un entier positif ( N ≥ 0 ) ")
        Lire( N )
    Jusqu'à N ≥ 0
    Ecrire( N, " ! = ", Facto( N ) ) // Appel de fonction
Fin
```

4. Les variables globales et les variables locales

- ✓ Une variable globale est déclarée dans l'algorithme principal et peut être utilisée dans une ou plusieurs procédures et/ou fonctions ainsi que dans l'algorithme principal.
- ✓ Une variable locale est déclarée dans une procédure ou une fonction et ne peut être utilisée que dans cette procédure (ou fonction).

Exemple : Reprenons l'algorithme TriCroissantVersion2. Ajoutons une variable globale Cpt. Cette variable Cpt est initialisée par l'algorithme principal et est incrémentée par la procédure Ordonner à chaque fois qu'elle est appelée.

```
Algorithme TriCroissantVersion3
Var A, B, C, Cpt : Entier
Procédure Ordonner( Var A : Entier ; Var B : Entier )
Var Tmp : Entier
Début
    Cpt ← Cpt + 1        // Cpt est une variable globale incrémentée à chaque appel.
    Si A > B Alors
        Tmp ← A
        A ← B
        B ← Tmp
    FSi
Fin
```

```
Début // algorithme principal
    Cpt ← 0 // Nombre d'appels est initialisé à zéro.
    Ecrire("Donner trois entiers A, B et C")
    Lire( A, B, C )
    Ordonner( A, B)        // 1er appel de procédure
    Ecrire(" La procédure Ordonner a été appelée ", Cpt, " fois")
    Ordonner( B, C)        // 2e appel de procédure
    Ecrire(" La procédure Ordonner a été appelée ", Cpt, " fois")
    Ordonner( A, B)        // 3e appel de procédure
    Ecrire(" La procédure Ordonner a été appelée ", Cpt, " fois")
    Ecrire("Les trois entiers dans l'ordre : ", A, B, C)
Fin
```

Remarques : ①-Toutes les variables A, B, C et Cpt sont globales.

②- La variable ***Tmp*** est locale à la procédure ***Ordonner***. Les variables ***F*** et ***I*** sont locales à la fonction ***Facto***.

③- Les paramètres d'une procédure (ou d'une fonction) sont considérées comme des variables locales. Il n'est pas possible de récupérer leurs valeurs à l'extérieur de la procédure (ou de la fonction).

④- Si une variable globale et une variable locale ont le même nom (même si elles sont de deux types différents), la variable globale devient inaccessible dans la procédure ou la fonction où la variable locale est déclarée.

Pour l'algorithme TriCroissantVersion3, la procédure ***Ordonner*** utilise deux paramètres ***A*** et ***B*** et l'algorithme principal utilise deux variables globales ***A*** et ***B*** ➔ les deux variables globales ***A***, ***B*** ne sont plus accessibles dans la procédure ***Ordonner***. Par contre les deux variables globales ***C*** et ***Cpt*** restent accessibles dans la procédure ***Ordonner***.

5. Le passage de paramètres

Il existe deux types (ou modes) de passage de paramètres : Passage par valeur et passage par adresse.

5.1- Le passage par valeur (par copie)

Les paramètres non précédés par le mot clé VAR, sont passés à la procédure (ou la fonction) par valeur. C'est-à-dire, la valeur du paramètre est copiée dans une variable locale sans toucher la variable donnée dans l'appel. C'est cette variable locale qui est utilisée dans la procédure (ou la fonction). Aucune modification de la variable locale dans la procédure (ou la fonction) ne modifiera la variable passée en paramètre.

5.2- Le passage par adresse (par variable ou par référence)

Les paramètres précédés par le mot clé VAR, sont passé à la procédure par adresse. La procédure travaille directement sur la variable passée en paramètre. Toutes les modifications sur le paramètre seront conservées par la variable passée en paramètre.

Remarques : ①- Tous les paramètres d'une fonction sont passés par valeur.

②- Une fonction retourne un seul résultat à travers le nom de la fonction et non pas à travers les paramètres.

③- Une procédure peut retourner aucun résultat (si la procédure n'utilise que des variables globales ou elle ne fait que de l'affichage), un seul résultat ou plusieurs résultats et cela à travers les paramètres passés par adresse (précédés par le mot clé VAR).

6. Les fonctions et procédures récursives

6.1- Définition

Une procédure (ou fonction) est dite récursive si elle s'appelle elle-même. C'est-à-dire dans son corps (liste des instructions) on trouve un appel à la procédure (ou fonction) elle-même.

- L'idée : en supposant qu'on a la solution pour le cas d'ordre (N-1), est ce qu'on pourrait avoir la solution pour le cas d'ordre N ?

- La récursivité est un moyen naturel de résolution de certains problèmes où la version récursive est plus simple à trouver que la version itérative (Exemple : Tours de Hanoï).
- Ce type de programmation permet de réaliser des fonctions définies à partir de relations de récurrence comme :

- La puissance : $X^n = X * X^{n-1}$ et $X^0 = 1$
- La factorielle : N ! = N * (N-1) ! et 1 ! = 1 et 0 ! = 1
- Plus Grand Commun Diviseur de deux entiers strictement positifs A et B :

$$\begin{cases} PGCD(A, B) = PGCD(B, A \text{ Mod } B) & \text{si } B \neq 0 \\ PGCD(A, B) = A & \text{si } B = 0 \end{cases}$$

- Suite de Fibonacci :

$$\begin{cases} U_n = U_{n-1} + U_{n-2} & \text{Si } n \geq 2 \\ U_0 = 0 \text{ et } U_1 = 1 & \text{Si } n < 2 \end{cases}$$

6.2- La structure d'une fonction récursive

Une fonction récursive à la structure générale suivante :

Fonction nomFonctRecursive(Parametre1 : Type1 ; … ; ParametreN :typeN) : TypeRésultat

/* Variables locales s'il y a besoin */

Début

```
    Si condition Alors /* Condition d'arrêt */
        nomFonctRecursive ← Résultat   /* Cas élémentaire (trivial ou particulier) */
    Sinon                                                    Appel récursif
        nomFonctRecursive ← … nomFonctRecursive( p1, …, pN )  … /* Cas général */
    FSi
```

Fin

Remarques : ①- L'appel d'une fonction (ou procédure) à l'intérieur d'elle-même (dans son corps) est dit appel récursif.

②- Un appel récursif doit obligatoirement être dans une instruction conditionnelle (Si … Sinon ...). Dans le cas contraire, la récursivité est sans fin. C'est-à-dire, on est en présence d'une boucle infinie.

③- Une procédure récursive a la même structure qu'une fonction récursive. C'est-à-dire, elle doit contenir au moins une structure conditionnelle (Si … Sinon …) pour traiter les deux cas : cas particulier et cas général. L'appel récursif de la procédure se fait par une instruction et n'est pas dans une expression.

6.3- Exemple d'une fonction récursive – La factorielle

La fonction suivante calcule la factorielle d'un entier positif N. En se basant sur la relation de récurrence N ! = N * (N–1) ! et en sachant que : 1 ! = 1 et 0 ! = 1.

Fonction FactoRec(N : Entier) : Entier

Début

```
    Si (N = 1) OU (N =0) Alors
        FactoRec ← 1
    Sinon
        FactoRec ← N * FactoRec( N – 1 )          /* appel récursif */
    FSi
```

Fin

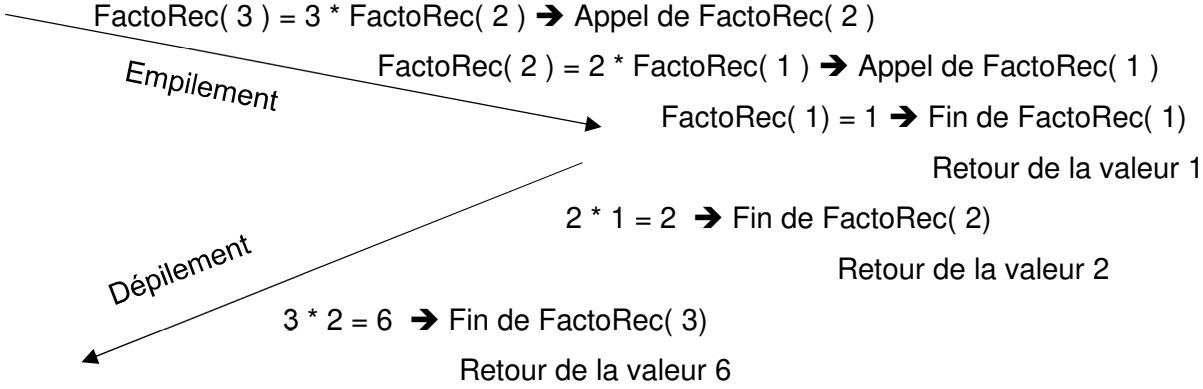
On peut réécrire l'algorithme qui affiche la factorielle d'un entier positif comme suit :

```
Algorithme Factorielle
Var N : Entier
Fonction FactoRec( N : Entier) : Entier
Début
    Si (N = 1) OU (N =0) Alors
        FactoRec ← 1
    Sinon
        FactoRec ← N * FactoRec( N – 1 )        /* appel récursif */
    FSi
Fin
Début
    Répéter
        Ecrire("Donner un entier positif ( N ≥ 0 ) ")
        Lire( N )
    Jusqu'à N ≥ 0
    Ecrire( N, " ! = ", FactoRec( N ) )
Fin
```

6.4- Comment la récursivité marche ?

Si l'utilisateur a saisi 3 pour la valeur de N alors pour calculer et afficher le message **3 ! = 6**, l'algorithme fait comme suit :

Appel de FactoRec(3)



Affichage à l'écran du message **3 ! = 6**

Remarques :

Pour pouvoir gérer ces appels, le système dispose d'une **pile d'appels (pile d'exécution)**. Pour chaque appel de fonction (ou de procédure), sont sauvegardées dans cette pile les valeurs des paramètres donnés dans l'appel, les valeurs des variables locales et l'adresse de retour.

A chaque fois qu'une fonction (ou une procédure) se termine, les informations sauvegardées lors de son appel sont supprimées de la pile.

La pile d'appels a une taille limitée, ce qui implique que la récursion ne doit pas être de très grande taille. Sinon, la pile devient pleine et la récursion ne donne pas de résultat !

6.5- Résolution récursive d'un problème

Pour créer une fonction (procédure) récursive, il faut :

1. Décomposer le problème initial en un ou plusieurs sous-problèmes du même type. On résout les sous-problèmes par des appels récursifs.
2. Les sous-problèmes doivent être de taille plus petite que le problème initial. C'est-à-dire que la taille doit diminuer.
3. La décomposition doit en fin de compte arriver à un cas élémentaire qui n'est pas décomposé en sous-problèmes (Condition d'arrêt). On obtient directement un résultat.

Remarques : L'intérêt d'écrire des fonctions (ou procédures) récursives réside dans une meilleure lisibilité. L'inconvénient est l'accroissement de la taille de la pile d'appels lors de son utilisation et des coûts prohibitifs lors d'une utilisation non optimisée (Exemple de la suite de Fibonacci si programmation directe).

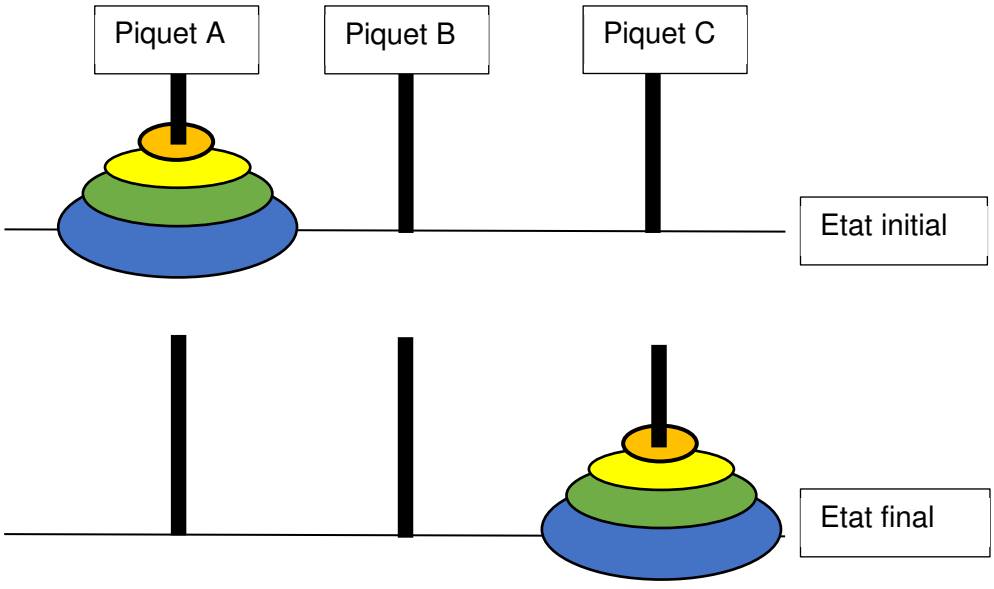
6.6- Exemple de procédure récursive – Les Tours de Hanoï

C'est un jeu. On dispose de trois piquets (***A***, ***B*** et ***C***) et d'un certain nombre de disques empilés sur le piquet ***A***. On désire déplacer la totalité des disques sur le piquet ***C*** en utilisant le piquet ***B*** comme intermédiaire et en respectant les deux règles suivantes :

①- On déplace un disque à la fois.

②- Les disques doivent être disposés du plus petit au plus grand sur chaque piquet.

Ecrire une procédure récursive qui permet de modéliser ce problème ainsi que l'algorithme principal permettant de l'utiliser.



Analyse :

Quel est le cas particulier ?

Réponse : Cas où N =1.

Si on a un seul disque, alors il suffit de le déplacer directement du piquet A vers le piquet C.

Quel est le cas général ?

Réponse : Cas où N > 1.

Si on a plusieurs disques alors :

1. On déplace les (N – 1) disques du piquet A vers le piquet B en utilisant le piquet C comme intermédiaire.
2. On déplace le dernier disque du piquet A directement vers le piquet C.
3. Enfin, on déplace les (N – 1) disques du piquet B vers le piquet C en utilisant le piquet A comme intermédiaire.

Solution :

```
Algorithme ToursDeHanoi
Var     N : Entier
Procédure deplacer( N : Entier ; T1 : Entier ; T2 : Entier ; T3 : Entier )
/*      N : Le nombre de disques à déplacer
        T1 : Numéro de la tour Source
        T2 : Numéro de la tour Intermédiaire
        T3 : Numéro de la tour Destination   */
Début
    Si N = 1 Alors
        Ecrire ("De la tour ", T1, " à la tour " , T3 )
    Sinon
        deplacer( N – 1 , T1 , T3 , T2 )
        deplacer( 1 , T1 , T2 , T3 )     // OU Ecrire ("De la tour ", T1, " à la tour " , T3 )
        deplacer( N – 1 , T2 , T1 , T3 )
    Fsi
Fin
Début
    Répéter
        Ecrire("Donner le nombre de disques ( N > 0 ) ")
        Lire( N )
    Jusqu'à N > 0
    deplacer( N, 1, 2, 3 )  // Commencer le déplacement des disques
Fin
```

Remarque :

Les paramètres T1, T2, T3 peuvent être de type Caractère.

Dans ce cas l'entête de la procédure devient :

Procédure deplacer(N : Entier ; T1 : Caractère ; T2 : Caractère ; T3 : Caractère)

L'appel dans le programme principal devient :

deplacer(N, 'A', 'B', 'C')

7. Synthèse : Procédures, Fonctions, Récursivité

- ✓ Les procédures et fonctions (sous-algorithmes) sont créées notamment pour :
  - Faciliter la résolution des problèmes en les découpant en petits problèmes faciles à résoudre,
  - Réutiliser les mêmes opérations au lieu de les réécrire à chaque fois,
  - Faciliter la modification et l'amélioration des algorithmes (des programmes) en modifiant qu'une seule partie de l'algorithme global.
- ✓ Les procédures et fonctions sont définies dans la partie déclarations d'un algorithme juste en dessous des variables.
- ✓ Les procédures sont utilisées pour renommer une suite d'actions.
- ✓ Une procédure est utilisée (appelée) comme une instruction simple.
- ✓ Une fonction est utilisée pour faire un calcul.
- ✓ Une fonction est utilisée comme un opérateur.
- ✓ Les paramètres passés à une procédure sont passés par Valeur (Le paramètre n'est pas modifié) ou par Adresse (Le paramètre peut être modifié).
- ✓ Les paramètres passés à une fonction sont tous passés par Valeur.
- ✓ Les variables globales sont définies dans l'algorithme principal et sont reconnues partout. Dans toutes les procédures et fonctions et dans l'algorithme principal.
- ✓ Une procédure peut retourner zéro, un ou plusieurs résultats à travers le passage des paramètres par Adresse.
- ✓ Une fonction retourne un et un seul résultat à travers le nom de la fonction. En effet la dernière instruction de la fonction doit être une affectation du résultat au nom de la fonction.
- ✓ La récursivité est une technique qui permet de définir une fonction (ou procédure) par l'appel à la fonction (ou la procédure) elle-même. Pour ce faire, il faut définir une relation de récurrence entre les paramètres de la fonction (ou de la procédure) et un cas trivial (particulier) pour que les appels récursifs s'arrêtent.
- ✓ La récursivité permet d'écrire des algorithmes très facilement à des problèmes difficiles à résoudre par des algorithmes itératifs (Exemple : Tours de Hanoï).

- ✓ La version itérative est toujours préférable par rapport à la solution récursive, car cette dernière consomme plus d'espace mémoire. En effet, il faut sauvegarder toutes les données (paramètres et variables locales) de tous les appels récursifs dans une pile nommée « pile des appels » ou « pile d'exécution ».

PARTIE 2– STRUCTURES DYNAMIQUES

1. Introduction

Dans cette partie nous nous intéressons aux structures dynamiques ; essentiellement les listes linéaires chaines. Puis nous aborderons les types abstraits de piles et de files. Nous aborderons d'une manière introductive le type arbre.

2. Les pointeurs

2.1- Définition d'un pointeur

Un pointeur est une variable qui contient l'adresse mémoire (notée @) d'une autre variable (ou information) stockée en mémoire centrale (RAM).

2.2- Déclaration d'un type pointeur

TYPE nomTypePointeur **=** **^**Type_de_base

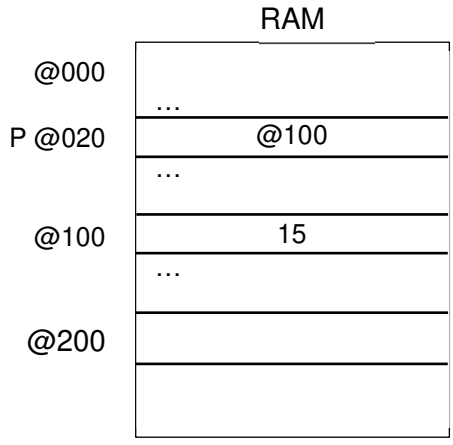
Exemple : déclaration d'un type pointeur sur un entier

Type *PTREntier* **=** **^**Entier

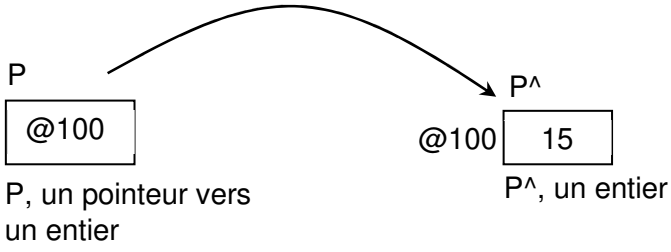
Var P : *PTREntier*

- P est une variable de type pointeur sur un entier. Elle contient l'adresse mémoire d'un entier.
- P^ est une variable entière pointée par le pointeur P.

Exemple : Si on suppose que la variable P (Le pointeur P) est rangée en mémoire centrale (RAM) à l'adresse @020 et que le pointeur contient l'adresse @100 alors P = @100 et P^ = 15.



Schématiquement, on représente ces informations comme suit :



P, un pointeur vers un entier

P^, un entier

2.3- Actions sur les pointeurs

Initialisation

Pour initialiser un pointeur P, on doit lui affecter la valeur **NIL (Not In List)**. NIL est une constante qui représente une adresse particulière qui ne pointe vers aucune donnée. On écrit alors P ← NIL.

Allocation de l'espace mémoire

Pour réserver de l'espace mémoire pour la variable pointée, il faut utiliser la procédure **ALLOUER**. ***Allouer(P)*** : réserve un espace mémoire pour une donnée pointée par le pointeur P.

Libération de l'espace mémoire

Pour libérer l'espace mémoire (qui a été précédemment réservé avec la procédure Allouer), il faut utiliser la procédure **LIBERER**. ***Liberer(P)*** : permet de supprimer la donnée pointée par le pointeur P. Cette procédure, ne supprime pas l'adresse contenue dans le pointeur P. On doit lui affecter la valeur NIL pour casser (rompre) la liaison entre le pointeur P et l'ancienne donnée stockée en mémoire.

Remarques

①- Un pointeur a un type de données spécifique. Deux pointeurs de types différents ne peuvent pas être affectés l'un à l'autre.

②- La déclaration d'une variable pointeur réserve (d'une manière statique) l'espace mémoire nécessaire pour le stockage d'une adresse mémoire, mais ne réserve aucune mémoire pour la variable pointée (la donnée pointée).

3. Les Listes Linéaires Chainées (LLC)

3.1- Exemples d'introduction

Comment faire pour stocker les nombres trouvés si on avait les deux questions suivantes

Q1) Ecrire un algorithme qui permet de trouver et de sauvegarder tous les nombres premiers inférieur à un entier N.

Q2) Ecrire un algorithme qui permet de trouver tous les nombres amis inférieurs à un entier N. Deux entiers A et B sont dits amis si la somme des diviseurs propres de A (A est exclu) est égale à B et la somme des diviseurs propres de B (B est exclu) est égale à A.

Proposition

Utiliser un tableau (structure statique).

- Si un tableau est utilisé, il n'est pas possible de définir sa taille avec précision même si la valeur de N est connue. Il faut donc choisir une taille suffisante dès le début, ce qui peut amener à un gaspillage (perte) considérable de place mémoire.

- Les tableaux ont un deuxième inconvénient : si un élément est supprimé du tableau, il faut déplacer (décaler) tous les éléments qui le suivent et même cela ne modifiera pas la taille du tableau car elle est fixée à la déclaration.

Solution

Donc on a besoin d'utiliser une structure dynamique qui évolue en taille. C'est-à-dire, au début, elle est vide, puis on lui rajoute un premier élément, puis un deuxième, puis un troisième et ainsi de suite. Ces éléments sont reliés entre eux par des pointeurs. Ces éléments constituent des maillons d'une nouvelle structure appelée **UNE LISTE**.

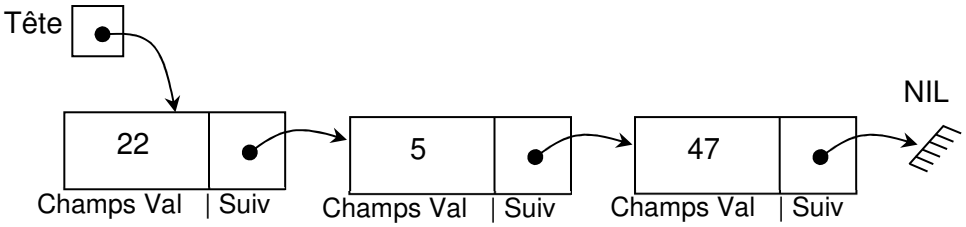
3.2- Définition d'une liste

Une Liste Linéaire Chaînée (LLC) est un ensemble d'éléments (Cellules, Nœuds, Maillons) alloués dynamiquement chaînés (reliés) entre eux. Chaque élément est un enregistrement qui contient au moins deux champs :

①- Un ou plusieurs champs qui contiennent l'information (Champ Valeur, Val, Info ou Data).

②- Le dernier champ contient un pointeur sur l'élément suivant (Champ Suivant, Suiv).

Schématiquement on peut représenter une liste de trois éléments (nombres entiers) comme suit :



Une liste est définie par un pointeur qui pointe vers le premier élément ; cet élément est appelé tête de la liste.

Le dernier élément de la liste est appelé queue.

3.3- Déclaration d'une Liste Linéaire Chainée

```
Type   Liste = ^Element
       Element = Enregistrement
                 Val : TypeDonnee /* Un type quelconque : entier, réel, booléen, tableau,
                 enregistrement …*/
                 Suiv : Liste  /* Pointeur vers l'élément suivant */
       Fin
Var    L, P, Q : Liste
```

Exemple : Pour déclarer une liste d'entiers, on doit écrire :

```
Type    Liste = ^Element
Element = Enregistrement
                Val : Entier
                Suiv : Liste
          Fin
Var     tete : Liste      // tete est une liste d'entiers.
```

Remarque : Dans la suite du cours, nous travaillons avec des listes d'entiers. Les mêmes principes étudiés son applicables aux autres types de données.

3.4- Accès aux données d'une Liste Linéaire Chainée

Une liste est un pointeur. Pour accéder à la valeur pointée (élément, nœud, cellule, maillon), on doit utiliser l'opérateur chapeau (^). Chaque élément (nœud, cellule, maillon) est un enregistrement. Pour accéder aux champs de l'enregistrement, on utilise l'opérateur point (.).
Pour accéder au champ de données (Val) on écrit :

L^.Val ← 5 (par exemple)

Pour accéder au champ Pointeur (Suiv) on écrit :

L^.Suiv ← NIL (par exemple)

3.5- Opérations sur les listes

On peut classer les opérations sur les listes en trois catégories : construction (ou modification), consultation (ou exploitation) et destruction (ou suppression).

5.1- Construction :

1- Initialisation (création d'une liste vide).

2- Insertion (ajout) en tête (au début), au milieu ou en queue (à la fin).

3- Insertion par position.

4- Insertion dans une liste triée, …

5.2- Consultation (exploitation) :

1- Affichage des éléments de la liste,

2- Calcul de la longueur de la liste (nombre d'éléments),

3- Recherche d'une valeur si elle existe,

4- Nombres d'occurrences, …

5.3- Destruction (suppression) :

1- Suppression d'un élément : En tête, au milieu ou à la fin,

2- Destruction de la liste entière.

3.5.1- Initialisation d'une Liste Linéaire Chainée

Si L est une liste, son initialisation consiste à lui donner la valeur NIL (Not In List).

On écrit alors : L ← NIL.

Rappel : NIL est une constante qui représente une adresse particulière qui ne pointe vers aucune donnée.

3.5.2- Insertion dans une Liste Linéaire Chainée

Soit **L** le nom de la liste et **P** le pointeur sur le nouvel élément créé.

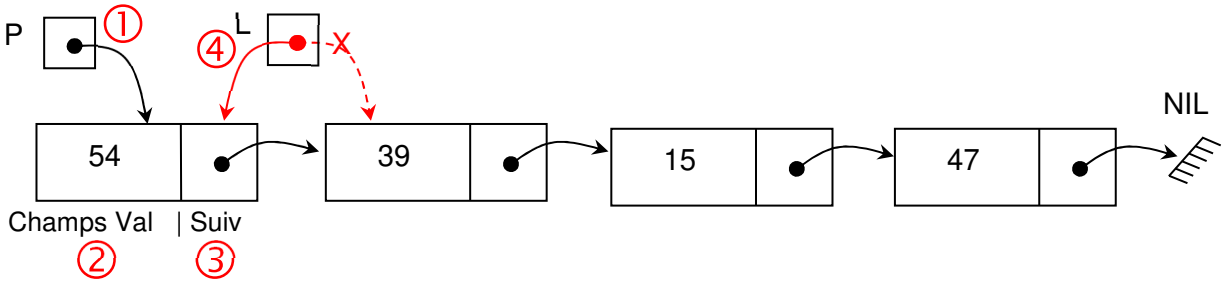
Dans tous les cas d'insertion, cette opération se fait en deux étapes :

Un nouvel élément (cellule) est créé avec l'instruction **ALLOUER(P).** Puis le champ **Valeur (Val)** est initialisé avec la valeur voulue **P^.Val ← valeur**.

L'élément créé est inséré dans la liste à l'endroit souhaité. C'est-à-dire, le nouvel élément est relié (chaîné) avec les autres éléments de la liste en adaptant leurs champs **Suivant (Suiv)**.

3.5.2.1- Insertion en tête de liste

Dans ce cas, il faut modifier le pointeur de tête de liste (L) et d'établir un lien entre l'élément nouvellement créé et l'élément qui se trouvait en tête (s'il existait). Ces opérations sont expliquées sur le schéma suivant :



On peut écrire une procédure ***InsererTete*** qui permet d'insérer en tête (au début) d'une liste L, une valeur val (val = 54 dans l'exemple).

```
Procédure InsererTete(Var L :Liste, val : Entier)
Var     P : Liste
Début
        ① Allouer( P )          // Créer un nouvel élément.
        ② P^.Val ← val          // Initialiser le champ de données (Val), insérer la valeur val.
        ③ P^.Suiv ← L           /* Initialiser le champ pointeur (Suiv). Créer le lien entre le
        nouvel élément et l'ancienne tête de la liste */
        ④ L ← P                 // Changer la tête (le début) de la liste
Fin
```

Exercice :

Ecrire un algorithme qui permet de créer une liste à partir des éléments d'un tableau d'entiers. Il faut que l'ordre des éléments dans la liste reste le même que celui des éléments du tableau.

Après la création de la liste, afficher ses éléments ainsi que leur nombre.

Solution :

```
Algorithme CreationListe
Const  N = 10
Type   Liste = ^Element
       Element = Enregistrement
           Val : Entier
           Suiv : Liste
       Fin
Var    Tab : Tableau[ 1 .. N ] d'entier
       I : Entier
       L, P : Liste      // L est la tête de la liste
Procédure InsererTete(Var L :Liste, val : Entier)
Var P : Liste
Début
       ① Allouer(P)
       ② P^.Val ← val
       ③ P^.Suiv ← L   // On crée le lien entre le nouvel élément et l'ancienne tête de la liste
       ④ L ← P   // On modifie la tête de liste
Fin
Début
       // Lecture des éléments du tableau
       Pour I ← 1 à N Faire
           Lire( Tab[ I ] )
       Fpour
       L ← Nil   // Initialisation de la liste pour indiquer qu'elle ne contient aucun élément.
      /* Création de la liste : Pour que les éléments gardent le même ordre, il faut
      commencer l'insertion par le dernier élément et continuer jusqu'au premier. Au final, la
      liste garde le même ordre. Sinon, les éléments de la liste seront dans l'ordre inverse */
       I ← N                                  // Solution 2
       TQ I > 0 faire                         Pour I ← N à 1 PAS = -1 Faire
           InsererTete( L, Tab[ I ] )             InsererTete( L, Tab[ I ] )
           I ← I – 1                          Fpour
       FTQ
       // Affichage et comptage des élément de la liste
       Cpt ← 0
       P ← L
       TQ P ≠ Nil Faire        // Tant que on n'est pas encore arrivé à la fin de la liste
```

```
        Ecrire( P^.Val)          // Affichage de la valeur
        Cpt ← Cpt + 1            // Compter cet élément
        P ← P^.Suiv              // Passer à l'élément suivant
    FTQ
    Ecrire("La liste contient ", Cpt, " élément(s)")        // Cpt doit être égal à N
Fin
```

Remarque : L'affichage et le comptage seront, plus loin dans le cours, l'objet d'une procédure ***AfficherListe*** et une fonction ***Longueur*** respectivement.

3.5.2.2- Insertion au milieu de la liste

Dans une liste simplement chaînée, le sens de parcours est toujours d'un élément (cellule) vers l'élément suivant : on ne peut pas revenir en arrière.

Pour insérer entre deux éléments, on insérera toujours après un élément donné.

Soit **Courant** l'élément après lequel on va insérer le nouvel élément.

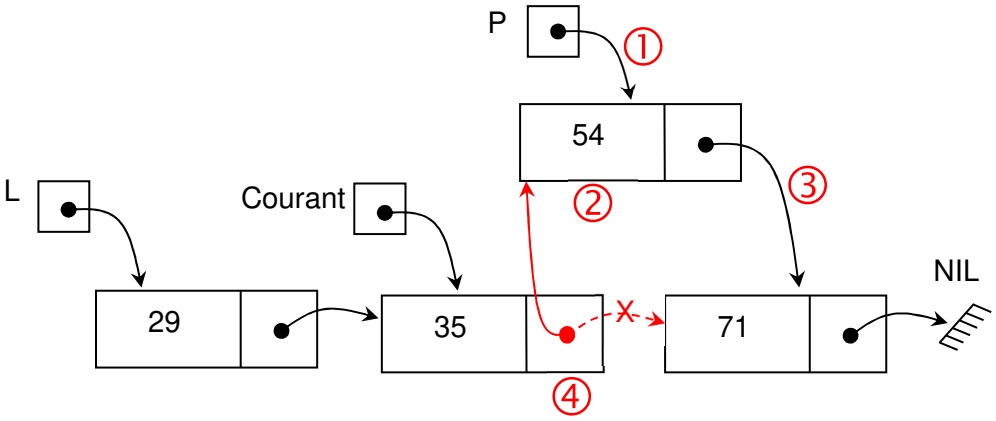
- On mémorise d'abord l'élément suivant de Courant dans le champ Suivant de P.

P^.Suiv ← Courant^.Suiv (Opération ③ sur le schéma ci-dessous)

- On peut maintenant faire pointer le champ Suivant de Courant sur P.

Courant^.Suiv ← P (Opération ④ sur le schéma ci-dessous)

Le schéma ci-dessous illustre l'insertion d'une valeur (54) dans une liste triée.




```
Procédure InsererALaSuiteDe(Courant : Liste, val : Entier)
Var     P : Liste
Début
    ① Allouer(P)
    ② P^.Val ← val
    ③ P^.Suiv ← Courant^.Suiv
    ④ Courant^.Suiv ← P
Fin
```

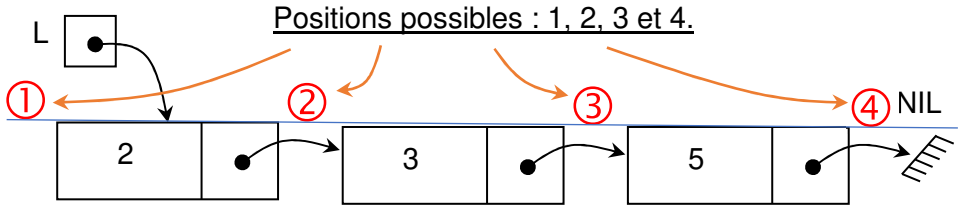
Remarque : Si on a une liste qui contient 4 éléments, on peut insérer un nouvel élément entre le 1er et le 2e élément ou entre le 2e et le 3e élément ou encore entre le 3e et le

4eélément. Pour faire le choix de l'endroit où l'insertion doit se faire, il faut avoir un critère. En général, on en a deux :

- ✓ Soit la liste est triée et donc chercher où on doit insérer le nouvel élément pour que la liste reste triée.
- ✓ Soit, on insère le nouvel élément par position. Par exemple, on souhaite l'insérer à la 3e position et donc on va l'insérer entre le 2e et le 3e élément. C'est-à-dire, on va l'insérer après le 2e élément.

3.5.2.2.1- Insertion par position

Le but est d'insérer le nouvel élément à une position donnée, si c'est possible.
Exemple : Si la liste contient trois éléments, on peut (toujours) insérer à la position 1, à la position 2, à la position 3 et à la 4e position (la dernière). Mais, il est impossible d'insérer à d'autres positions comme la position 5, 6, …
Sur le schéma suivant, sont montrées les positions possibles si on a une liste de trois éléments.




```
Procédure InsererParPosition(Var L : Liste ; Pos : Entier ; val : Entier)
Var     Cpt : Entier
        A, Courant, P : Liste
Début
    Si Pos = 1 Alors
        InsererTete( L, val) // Insertion en position 1 (en tête), opération toujours possible.
    Sinon
        Cpt ← 0
        Courant ← L
        TQ ( Courant ≠ Nil ) ET ( Cpt < (Pos – 1) ) Faire
            Cpt ← Cpt + 1
            A ← Courant  // Sauvegarder la valeur de Courant
            Courant ← Courant^.Suiv     // passer à l'élément Suivant
        FTQ
        Si Pos = Cpt + 1 Alors          // Vérifier si la position existe réellement !
            ① Allouer( P )              // Insérer le nouvel élément après l'élément A
            ② P^.Val ← val
```

```
                    ③ P^.Suiv ← A^.Suiv
                    ④ A^.Suiv ← P
                /*Sinon
                    Ecrire("Erreur : impossible d'insérer à la position ", Pos)*/
                FSi
        FSi
Fin
```

3.5.2.2.2- Insertion dans une liste triée

On suppose que la liste peut contenir plusieurs occurrences de la même valeur.

Solution 1

```
Procédure InsererListeTriee(Var L : Liste ; val : Entier)
Var     P, Q, Courant : Liste
        Stop : Booléen
Début
    ① Allouer(P)
    ② P^.Val ← val
    Si L = Nil Alors          // La liste est vide ➔ Insertion en tête
        ③ P^.Suiv ← Nil       // ≡ P^.Suiv ← L
        ④ L ← P
    Sinon
        Si L^.Val > val Alors  // Val est supérieure à la 1ière Valeur de la liste
            ③ P^.Suiv ← L          // Insertion en tête
            ④ L ← P
        Sinon
            // Insertion au milieu ou à la fin de la liste
            Courant ← L
            Stop ← Faux
            TQ (Courant^.Suiv ≠ Nil) ET (NON Stop) Faire
                Q ← Courant^.Suiv    // Q utilisée pour simplifier l'écriture
                Si Q^.Val > val Alors
                    Stop ← Vrai
                Sinon
                    Courant ← Q
                FSi
            FTQ
            ③ P^.Suiv ← Courant^.Suiv   // Que Courant^.Suiv soit Nil ou pas
            ④ Courant^.Suiv ← P
        FSi
    FSi
Fin
```

Solution 2

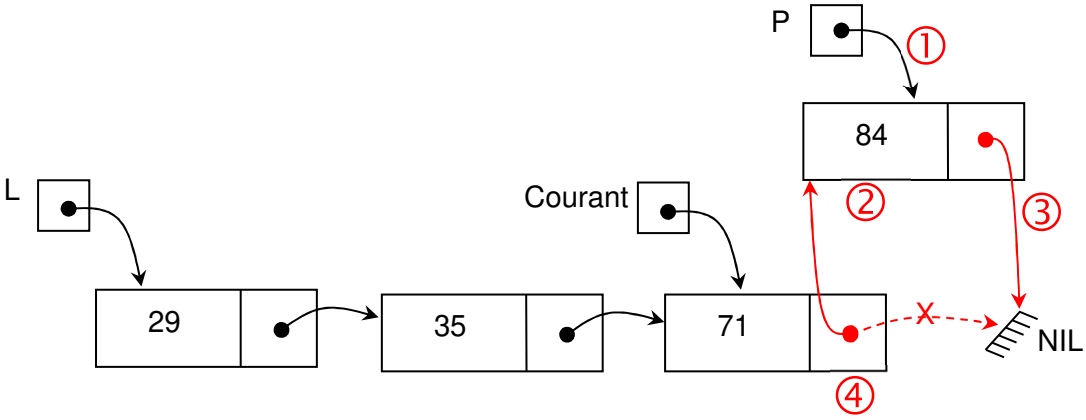
```
Procédure InsererListeTriee(Var L : Liste ; val : Entier)
Var     Q, Courant : Liste
        Stop : Booléen
Début
    Si L = Nil Alors          // La liste est vide ➔ Insertion en tête
        InsererTete(L, val)
    Sinon
        Si L^.Val > val Alors   // val est supérieure à la 1ière valeur de la liste
            InsererTete(L, val)
        Sinon
            // Insertion au milieu ou à la fin de la liste
            Courant ← L
            Stop ← Faux
            TQ (Courant^.Suiv ≠ Nil) ET (NON Stop) Faire
                Q ← Courant^.Suiv    // Q utilisée pour simplifier l'écriture
                Si Q^.Val > val Alors
                    Stop ← Vrai
                Sinon
                    Courant ← Q
                FSi
            FTQ
            InsererTete(Courant^.Suiv, val)   // Que Courant^.Suiv soit Nil ou pas
        FSi
    FSi
Fin
```

3.5.2.3- Insertion à la fin de la liste

C'est un cas particulier de l'insertion au milieu de la liste : Insérer à la fin de liste, c'est insérer après le dernier élément dont le champ **Suivant** vaut **Nil**. Donc, on doit parcourir la liste jusqu'au dernier élément (Courant), puis insérer le nouvel élément après lui.
Après insertion, le champ Suivant de Courant vaut P et le champ suivant de P vaut Nil.

```
P^.Suiv ← Nil
Courant^.Suiv ← P
```

Sur le schéma ci-dessous, sont expliquées les opérations nécessaires pour l'insertion d'un nouvel élément (valeur 84) à la fin de la liste L.



En suivant ces étapes, on peut écrire la procédure ***InsererQueue*** comme suit :

```
Procédure InsererQueue(Var L :Liste, val : Entier)
Var P, Courant : Liste
Début
    ① Allouer(P)
    ② P^.Val ← val
    ③ P^.Suiv ← NIL
    ④ Si L = Nil Alors
        L ← P   // Si la liste est vide, alors InsererQueue ≡ InsererTete.
    Sinon
        Courant ← L
        TQ Courant ^.Suiv ≠ Nil Faire   // Parcourir la liste jusqu'au dernier élément.
            Courant ← Courant ^.Suiv
        FTQ
        Courant ^.Suiv ← P
    FSI
Fin
```

3.5.3- Consultation

Les opérations de consultation (ou d'exploitation) permettent d'utiliser la liste sans lui apporter aucune modification (ni ajout, ni suppression, ni modification d'ordre des élément, …). Parmi ces opérations, on peut citer : l'affichage de la liste, le comptage des éléments, la recherche d'une valeur si elle existe, le calcul du nombre d'occurrences d'une valeur, La vérification si la liste est triée ou pas, …

3.5.3.1- Affichage des éléments de la liste

Pour afficher la liste, on commence par sa tête et on passe par ses éléments un par un et on affiche leurs champs valeur (Val), jusqu'en arrive à la fin de la liste.

```
Procédure AfficherListe(L :Liste)
Var     P : Liste
Début
    Si L = Nil Alors
        Ecrire("Liste vide")
    Sinon
        P ← L
        TQ P ≠ Nil Faire
            Ecrire( P^.Val )
            P ← P^.Suiv
        FTQ
    FSi
Fin
```

```
/* Comme L est passée par valeur, on peut
s'en passer de la variable locale P */
TQ L ≠ Nil Faire
    Ecrire( L^.Val )
    L ← L^.Suiv
FTQ
```

3.5.3.2- Calcul de la longueur de la liste

La longueur d'une liste est définie comme étant le nombre d'éléments qu'elle contient.

```
Fonction Longueur(L : Liste) : Entier
Var     P : Liste
        Cpt : Entier
Début
    Cpt ← 0
    P ← L
    TQ P ≠ Nil Faire
        Cpt ← Cpt + 1
        P ← P^.Suiv
    FTQ
    Longueur ← Cpt
Fin
```

```
/* Comme L est passée par valeur, on peut ne
pas utiliser la variable locale P */
TQ L ≠ Nil Faire
    Cpt ← Cpt + 1
    L ← L^.Suiv
FTQ
```

3.5.3.3- Recherche d'une valeur dans la liste

Le résultat de recherche peut être soit un Booléen pour dire si la valeur recherchée existe ou n'existe pas, soit un pointeur (de type Liste) sur l'élément qui contient cette valeur. Si la valeur recherchée n'existe pas, le résultat de la fonction sera la constante Nil.
Nous optons ici pour cette dernière option.

```
Fonction Adresse(L : Liste ; val : Entier) : Liste
Var     P : Liste
        TRV : Booléen
Début
        TRV ← Faux
        P ← L
        TQ ( P ≠ Nil ) ET ( NON TRV ) Faire
                Si P^.Val = val Alors
                        TRV ← Vrai
                Sinon
                        P ← P^.Suiv
                FSi
        FTQ
        Adresse ← P
Fin
```

Remarque : Pour retourner un booléen, il suffit de retourner la valeur de la variable locale TRV au lieu du pointeur P.

Exercice : Modifier cette fonction pour qu'elle retourne, le nombre d'occurrences de la valeur val.

Solution

Pour compter le nombre d'occurrences, on doit parcourir la liste en entier. Donc, nous n'avons pas besoin de s'arrêter si la valeur val est trouvée. On a aussi besoin d'un compteur (entier) pour compter le nombre d'occurrences.

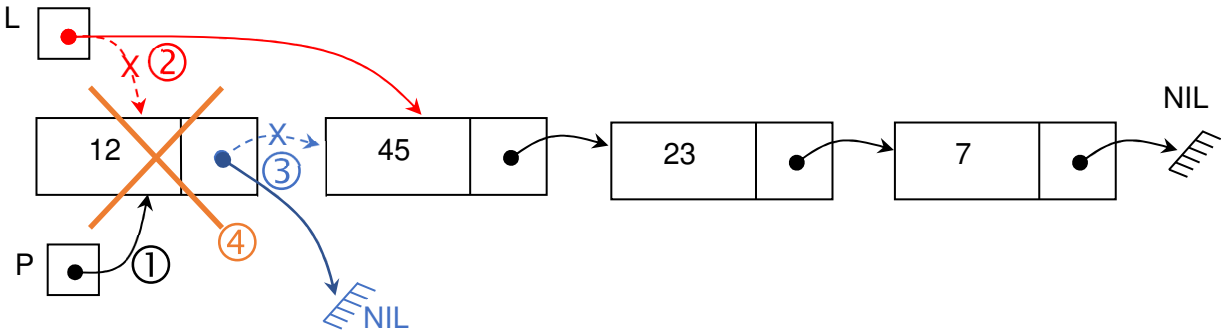
```
Fonction NBOccurrences(L : Liste ; val : Entier) : Entier
Var     P : Liste
        Cpt : Entier
Début
        Cpt ← 0
        P ← L
        TQ ( P ≠ Nil ) Faire
                Si P^.Val = val Alors
                        Cpt ← Cpt + 1           // Compter l'élément trouvé.
                FSi
                P ← P^.Suiv   // Poursuivre en allant à l'élément suivant.
        FTQ
        NBOccurrences ← Cpt
Fin
```

3.5.4- Modification

Les opérations de modification permettent de changer l'état de la liste en supprimant des éléments, en modifiant l'ordre des éléments, en modifiant les valeurs enregistrées, en fusionnant deux listes, … et enfin la destruction complète de la liste.
La suppression, comme l'insertion, peut se faire au début, au milieu ou à la fin.
Parmi ces opérations, nous nous intéressons dans ce cours à la suppression au début et à la destruction de la liste. D'autres opérations seront l'objet de travaux dirigés.

3.5.4.1- Suppression de la tête de la liste

Les opérations nécessaires pour supprimer la tête de la liste sont explicitées sur le schéma ci-dessous.




```
Procédure SupprimerTete(Var L :Liste)
Var     P : Liste
Début
    Si L ≠ Nil Alors     // On ne peut supprimer un élément que si la liste n'est pas vide !
        ① P ← L                  // Garder l'adresse du 1er élément.
        ② L ← L^.Suiv            // Modifier la tête de la liste
        ③ P^.Suiv ← Nil     // Optionnel. Casser le lien entre le 1er et le reste de la liste
        ④ Libérer(P)   // Libérer l'espace mémoire qu'occupait le 1er élément.
    FSi
Fin
```

3.5.4.2- Destruction de la liste

Pour détruire la liste, il est nécessaire de supprimer tous ces éléments. On peut utiliser la procédure SupprimerTete autant de fois que nécessaire, jusqu'à ce que la liste soit vide.

```
Procédure Supprimerliste(Var L :Liste)
Début
    TQ L ≠ Nil Faire        // Tant que la liste contient des éléments.
        SupprimerTete( L ) // Supprimer l'élément en 1e position. L est modifiée.
    FTQ
Fin
```

3.5.4.3- Fusion de deux listes triées

Le but est de fusionner deux listes triées pour avoir une troisième liste triée.

Solution 1 (la plus facile) : Les deux listes L1 et L2 ne sont pas détruites. Dans ce cas, il faut créer la liste L3 et insérer les valeurs de L1 et L2 dans le bon ordre dans L3 et à chaque fois à la fin de la liste L3.

```
Procédure Fusion(L1, L2 : Liste ; Var L3 : Liste)
Début
    L3 ← Nil
    TQ (L1 ≠ Nil) ET (L2 ≠ Nil) Faire      /* Les deux listes contiennent des éléments */
        Si L1^.Val < L2^.Val Alors
            InsererQueue(L3, L1^.Val)
            L1 ← L1^.Suiv          /* Passer à l'élément suivant dans L1 */
        Sinon
            InsererQueue(L3, L2^.Val)
            L2 ← L2^.Suiv          /* Passer à l'élément suivant dans L2 */
        FSi
    FTQ
    /* Au moins une liste est vide */
    TQ L1 ≠ Nil Faire      /* Si L1 n'est pas vide */
        InsererQueue(L3, L1^.Val)
        L1 ← L1^.Suiv
    FTQ
    TQ L2 ≠ Nil Faire      /* Si L2 n'est pas vide */
        InsererQueue(L3, L2^.Val)
        L2 ← L2^.Suiv
    FTQ
Fin
```

Solution 2 : Les deux listes L1 et L2 sont détruites et leurs éléments (cellules) sont réutilisés pour construire L3. Les pointeurs sur les éléments suivants sont modifier afin de construire la nouvelle liste L3.

```
Procédure Fusion(Var L1, L2, L3 : Liste)
Var P, D : Liste
Début
    L3 ← Nil
    P ← Nil
    TQ (L1 ≠ Nil) ET (L2 ≠ Nil) Faire      /* Les deux listes contiennent des éléments */
        Si L1^.Val < L2^.Val Alors
            P ← L1
            L1 ← L1^.Suiv          /* Passer à l'élément suivant dans L1 */
        Sinon
            P ← L2
            L2 ← L2^.Suiv          /* Passer à l'élément suivant dans L2 */
        FSi
        /* Insertion de l'élément pointé par P à la fin de la liste L3 */
        P^.Suiv ← Nil /* Retirer l'élément de sa liste d'origine */
        Si L3 = Nil Alors
            L3 ← P          /* Insertion en tête de la liste L3 */
        Sinon
            D^.Suiv ← P   /* Insertion à la fin de la liste L3 */
        FSi
        D ← P           /* D est le dernier élément de L3 */
    FTQ
    /* Au moins une liste est vide */
    SI L1 ≠ Nil Alors       /* Si L1 n'est pas vide */
        P ← L1
        L1 ← Nil /* Réinitialiser L1 */
    FSi
    Si L2 ≠ Nil Alors       /* Si L2 n'est pas vide */
        P ← L2
        L2 ← Nil /* Réinitialiser L2 */
    FSi
    /* Insertion de tous les éléments restants de L1 ou de L2 à la fin de la liste L3 */
    Si L3 = Nil Alors
        L3 ← P
    Sinon
        D^.Suiv ← P
    FSi
Fin
```

3.5.4.4- Eclatement d'une liste en deux listes selon le critère de parité

Le but est de construire deux listes ***L1*** et ***L2*** à partir des éléments d'une liste ***L*** en se basant sur le critère de parité.

Le principe : Le principe est de parcourir la liste ***L*** est d'extraire l'élément qui est en tête, puis de l'insérer à la fin de la liste ***L1*** si la valeur est impaire ou à la fin de la liste ***L2*** si la valeur est paire. A la fin, la liste ***L*** originale devient vide.

```
Procédure Eclater(Var L, L2, L3 : Liste)
Var P, D1, D2 : Liste
Début
    L1 ← Nil        /* L1 est vide */
    L2 ← Nil        /* L1 est vide */
    TQ L ≠ Nil Faire
        P ← L
        L ← L^.Suiv
        P^.Suiv ← Nil          /* Un élément est retiré de L à chaque itération */
        Si P^.Val Mod 2 = 1 Alors     /* Insertion à la fin de L1 (nombres impairs) */
            Si L1 = Nil Alors
                L1 ← P          /* Initialiser L1 */
            Sinon
                D1^.Suiv ← P /* Insertion après le dernier élément de L1*/
            FSi
            D1 ← P                  /* P devient le dernier élément de L1 */
        Sinon  /* Insertion à la fin de L2 (nombres pairs) */
            Si L2 = Nil Alors
                L2 ← P          /* Initialiser L2 */
            Sinon
                D2^.Suiv ← P /* Insertion après le dernier élément de L2*/
            FSi
            D2 ← P                  /* P devient le dernier élément de L2 */
        FSi
    FTQ
Fin
```

3.5.4.5- Inversion d'une liste

Le principe : On parcourt la liste du début jusqu'à la fin, et on sauvegarde les adresses (pointeurs) des trois (3) éléments qui se suivent : P (Précédent), Q (Courant) et L (Suivant)

et on modifie le champ Suiv de l'élément courant (Q). Il ne pointera plus vers l'élément suivant (L), mais sur l'élément précédent (P).

```
Procédure inverserListe(Var L : Liste)
Var P, Q : Liste
Début
    P ← Nil
    TQ L ≠ Nil Faire
        Q ← L          /* L'élément courant (actuel) */
        L ← L^.Suiv    /* L'élément suivant */
        Q^.Suiv ← P    /* Suiv pointe sur l'élément précédent (Nil au début). */
        P ← Q
    FTQ
    L ← P
Fin
```

4. Les algorithmes récursifs sur les listes

Dans cette section, nous nous intéressons à la version récursive de quelques opérations que nous avons déjà écrit leurs algorithmes en version itérative.

Parmi ces opérations, nous nous intéressons à l'affichage de la liste, la recherche d'une valeur et le nombre d'occurrences d'une valeur.

Dans toutes ces opérations, nous cherchons le (ou les) cas particulier(s) puis le cas général.

- Le cas particulier est souvent le cas où la liste est vide (ou a été entièrement parcourue).
- La cas général, lorsque la liste n'est vide. On traite l'élément courant (qui est en tête de liste), puis on fait un appel récursif pour traiter la liste contenant les éléments suivants.

4.1- Affichage récursif de la liste

```
Procédure AfficherListeREC(L : Liste)
Début
    Si L= Nil alors
        Ecrire("Liste Vide")
    Sinon
        Ecrire( L^.Val )     // Op1
        Si L^.Suiv ≠ Nil Alors
            /* Si ce n'est pas le dernier élément, on fait un appel récursif pour
            afficher les éléments suivants. Sinon on s'arrête */          // Op2
            AfficherListeREC( L^.Suiv ) )
        FSi
    FSi
Fin
```

4.2- Affichage inversé de la liste

Le but est d'afficher les éléments de la liste dans l'ordre inverse de leur présence dans la liste.

On sait que dans une liste (simplement chaînée), chaque élément ne donne accès qu'à l'élément qui le suit ; par conséquent écrire une version itérative pour cette opération n'est pas facile.

Cependant, pour écrire une version récursive, il suffit d'inverser d'ordre des opérations **Op1** et **Op2** de l'algorithme précédent pour obtenir un affichage inversé des éléments de la liste.

4.3- Recherche d'une valeur dans la liste

- Si la liste est vide, aucune valeur ne peut être trouvée dans la liste.
- Si la liste n'est pas vide, on vérifie si l'élément en tête de liste est égal à la valeur recherchée. Si c'est le cas, on a trouvé la valeur et on s'arrête. Sinon, avec un appel récursif, on vérifie si la valeur recherchée existe dans le reste de la liste.

```
Fonction AdresseREC(L : Liste ; val : Entier) : Liste
Début
    Si L = Nil Alors
        AdresseREC ← Nil    // val ne peut être trouvée.
    Sinon
        Si L^.Val = val Alors
            AdresseREC ← L      // val est trouvée.
        Sinon
            // Chercher val dans le reste de la liste.
            AdresseREC ← AdresseREC( L^.Suiv, val )
        FSi
    FSi
Fin
```

Remarque : Si nous n'avons pas besoin du pointeur sur l'élément qui contient la valeur **val** et nous souhaitons que la réponse soit : Vrai ou Faux, il suffit de changer le type de retour en **Booléen** et à la place de **Nil** mettre **Faux** et à la place de **L** mettre **Vrai**.

4.4- Calcul de la longueur d'une liste

Si la liste est vide, sa longueur est nulle (égal à zéro). Sinon, on compte l'élément courant, puis avec un appel récursif on ajoute la longueur du reste de la liste.

```
Fonction LongueurREC(L : Liste) : Entier
Début
    Si L = Nil Alors
        LongueurREC ← 0
    Sinon
        LongueurREC ← 1 + LongueurREC(L^.Suiv)
    FSi
Fin
```

4.5- Calcul du nombre d'occurrences d'une valeur donnée

Si la liste est vide, le nombre d'occurrence est nul (égal à zéro). Sinon, on vérifie si l'élément courant contient la valeur recherchée, si oui elle est comptée et ajoutée au nombre d'occurrences dans le reste de la liste, sinon, le nombre d'occurrences sera le nombre de fois où la valeur existe dans le reste de la liste seulement.

```
Fonction NombreOccurREC( L :Liste ; val : Entier ) : Entier
Début
    Si L = Nil Alors
        NombreOccurREC ← 0
    Sinon
        Si L^.Val = val Alors
            NombreOccurREC ← 1 + NombreOccurREC( L^.Suiv, val )
        Sinon
            NombreOccurREC ← NombreOccurREC( L^.Suiv, val )
        FSi
    FSi
Fin
```

4.6- Insertion par position

```
Procédure InsererPos(Var L : Liste ; val : Entier ; pos : Entier)
Début
    Si pos = 1 Alors
        InsererTete(L, val)
    Sinon
        Si L ≠ Nil Alors
            InsererPos(L^.Suiv, val, pos -1)
        /* Sinon
            Ecrire("Erreur : Position ", pos, " n'existe pas. Insertion impossible") */
        FSi
    FSi
Fin
```

4.7- Insertion dans une liste triée

```
Procédure InsererListeTriee(Var L : Liste ; val : Entier)
Début
    Si L = Nil Alors          // La liste est vide ➔ Insertion en tête.
        InsererTete(L, val)
    Sinon
        Si L^.Val > val Alors   // Val est supérieure à la valeur en tête de liste.
            InsererTete(L, val)
        Sinon
            InsererListeTriee(L^.Suiv, val)          // Insérer val dans la liste restante.
        FSi
    FSi
Fin
```

4.8- Suppression de toutes les occurrences d'une valeur donnée

```
Procédure SupprimerVal( Var L : Liste ; val : Entier )
Début
    Si ( L ≠ Nil ) Alors
        Si L^.Val = val Alors
            SupprimerTete( L )
            SupprimerVal( L, val )          // La valeur de L a changé.
        Sinon
            SupprimerVal( L^.Suiv, val )
        FSi
    FSi
Fin
```

4.9- Destruction de la liste

```
Procédure SupprimerListe( Var L : Liste )
Début
    Si ( L ≠ Nil ) Alors
        SupprimerTete( L )    // Supprimer l'élément en 1e position.
        SupprimerListe( L )    // La valeur de L a changé. Répéter !
    FSi
Fin
```

5. Listes linéaires chaînées particulières

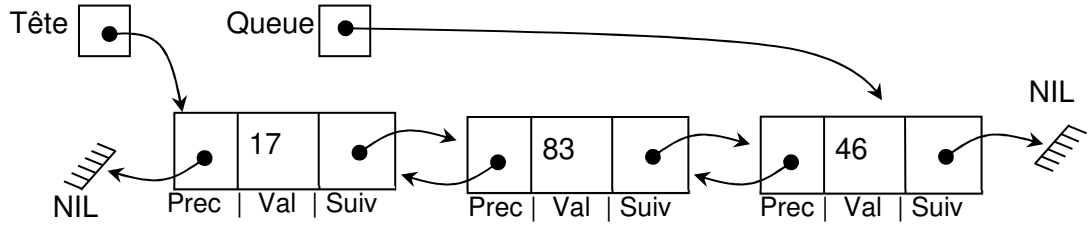
5.1- Les Listes bidirectionnelles (doublement chaînées)

Une liste linéaire chaînée est une liste où on ne peut effectuer qu'un parcours de gauche à droite. On peut qualifier cette liste de monodirectionnelle. Si on cherche à parcourir la liste de droite à gauche, il faut ajouter un deuxième pointeur permettant l'accès à la cellule précédente (champ Prec). On qualifie alors la liste de bidirectionnelle (ou doublement chaînée).

Définition

Une liste bidirectionnelle est une LLC que l'on peut parcourir dans les deux sens : de gauche à droite et de droite à gauche.

Schématiquement, une liste bidirectionnelle de 3 éléments se présente comme suit :



Chaque élément est un enregistrement qui contient trois (3) champs :

①- Un champ qui contient l'information (Val),

②- Un pointeur qui permet d'accéder à l'élément suivant (Suiv),

③- Un pointeur qui permet d'accéder à l'élément précédent (Prec).

Déclaration d'une Liste doublement Chaînée (bidirectionnelle)

```
Type    ListeDC = ^Element
        Element = Enregistrement
                  Suiv : ListeDC              // Un pointeur vers l'élément suivant
                  Val : TypeQuelconque        // Champ de données
                  Prec : ListeDC              // Un pointeur vers l'élément précédent
        Fin
Var     Tete, Queue : ListeDC       /* Tête : pointeur vers le 1er élément. Queue : pointeur vers le dernier élément */
```

Remarque : Pour exploiter efficacement ce type de liste, il est préférable d'utiliser deux pointeurs un pour la tête de la liste et un autre pour la queue. Ainsi, on peut par exemple afficher la liste dans le sens inverse en parcourant la liste à partir de la fin et en utilisant les champs Prec.

Opérations sur les listes doublement chaînées

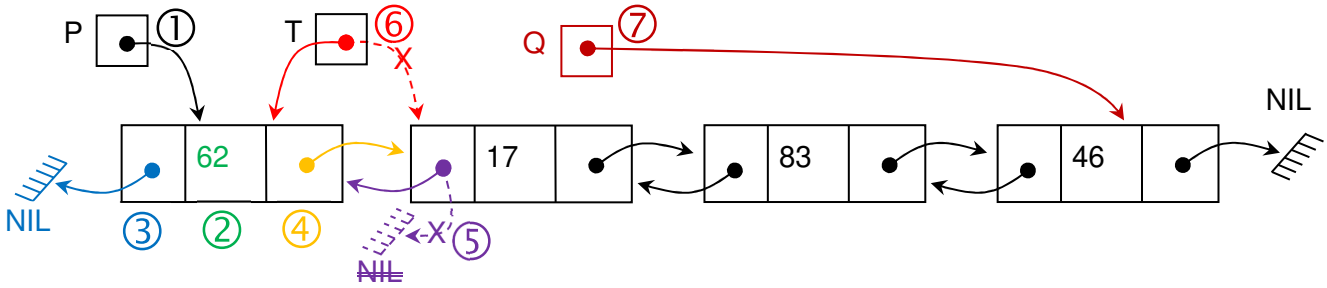
Toutes les opérations possibles sur les listes avec un seul chaînage sont possibles sur les listes doublement chaînées. Exemples : insertion dans une liste vide, insertion en tête,

insertion au milieu, insertion à la fin, suppression au début, suppression au milieu, suppression à la fin, recherche d'un élément, insertion à une position, calcul de la longueur… La seule différence est qu'il faut gérer deux chainages (les pointeurs Suiv et Prec) et gérer deux têtes (début et fin).

Pour illustrer ses opérations, nous choisissons l'exemple de l'insertion d'un nouvel élément en tête d'une liste doublement chaînée.

Insertion en tête d'une liste doublement chaînée

Les actions nécessaires pour insérer un nouvel élément dans une liste doublement chaînée (définie par les deux pointeurs T (Tête) et Q (Queue)) sont montrées sur le schéma ci-dessous :



Les opérations présentées sur le schéma précédent sont retranscrites dans la procédure ci-dessous.

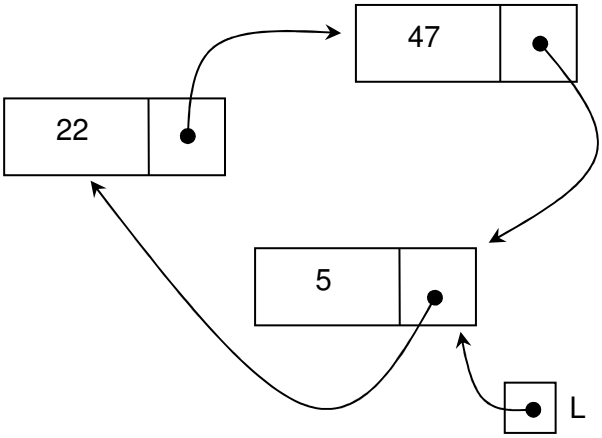
```
Procédure InsertionTete(Var T : ListeDC ; Var Q : ListeDC ; val : Entier)
Var     P : ListeDC
Debut
    ① Allouer( P )          // Allouer un nouvel élément
    ② P^.Val ← val          // Remplir le champ Valeur
    ③ P^.Prec ← Nil         // Remplir le champ Précédent
    ④ P^.Suiv ← T           // Remplir le champ Suivant
    Si T ≠ Nil Alors
        ⑤ T^.Prec ← P       // Si ce n'est pas la première insertion dans de la liste.
    FSi
    ⑥ T ← P         // Changer la tête de la liste
    Si Q = Nil Alors
        ⑦ Q ← P     // Si c'est la première insertion dans la liste.
    FSi
Fin
```

5.2- Les Listes circulaires (anneaux)

Une liste circulaire ou anneau est une liste linéaire dans laquelle le dernier élément pointe sur le premier. Il n'y a donc ni premier, ni dernier. Il suffit de connaître l'adresse d'un élément pour parcourir tous les éléments de la liste.

Une liste circulaire peut être simplement ou doublement chaînée.

Ci-après une liste circulaire (simplement chaînée) de trois éléments.



Remarque : Pour certaines applications, un tableau peut être utilisé pour représenter des listes (LLCs). Dans ce cas, chaque élément du tableau est un enregistrement qui renferme au moins deux champs : l'information (champ Val) et l'indice vers l'élément suivant (Champ Suiv). L'indice 0 ou -1 peut être utilisé pour remplacer la valeur de Nil. Dans ce cas, le nombre d'élément à créer est limité par la taille du tableau et non plus par la taille de la mémoire. Si le tableau est plein, on ne peut plus créer de nouveaux éléments.

6. Synthèse sur les listes

- ✓ Une liste (linéaire chaînées) est une structure dynamique dont la taille change au cours du déroulement de l'algorithme (l'exécution du programme) : Elle est vide au départ ; au besoin, les éléments sont insérés (ajoutés) ou supprimés (retirés) selon les besoins.
- ✓ Les insertions et les suppressions peuvent se faire au début, au milieu ou à la fin de la liste.
- ✓ Les insertions et les suppressions au milieu de la liste doivent se faire selon un critère : Par position ou le fait de garder la liste triée …
- ✓ A la fin du déroulement de l'algorithme (l'exécution du programme) la liste doit être détruite.
- ✓ La liste est une structure récursive par excellence où la version récursive d'un traitement est, en général, facile à déduire.

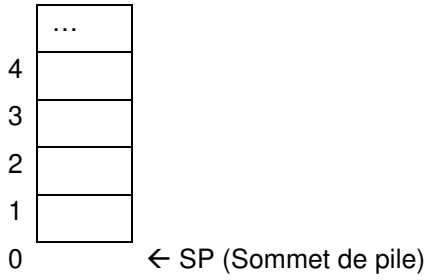
7. PILES

7.1- Définition, principe, domaines d'application

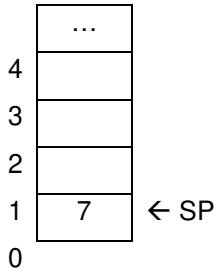
La pile constitue l'un des concepts les plus utilisés dans la science des ordinateurs.
Une pile peut être définie comme une collection d'éléments où les insertions et les suppressions d'éléments se font à la même extrémité de la liste appelée **sommet de pile** (noté SP).
Elle applique le principe LIFO pour Last In First Out (Dernier entré, Premier Servi).
La pile est très utilisée dans le domaine de la compilation : résolution de la portée des variables, récursivité, évaluation d'expression …

7.2- Exemple

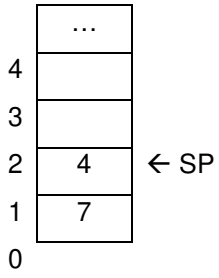
Pour expliciter le fonctionnement d'une pile, prenons l'exemple suivant : On suppose qu'on a une pile et on souhaite insérer les trois éléments (des valeurs entières) 7, 4 et 10 et puis en retirer deux éléments. Chaque insertion ou retrait se fait au sommet de pile.
On peut schématiser ces opérations comme suit :
1- Etat initial de la pile : Au départ, la pile est vide, elle ne contient aucun élément, par conséquent, le sommet de pile (SP) ne pointe vers aucun élément.



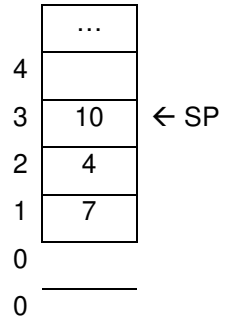
2- Insertion du 1er élément (valeur 7) :



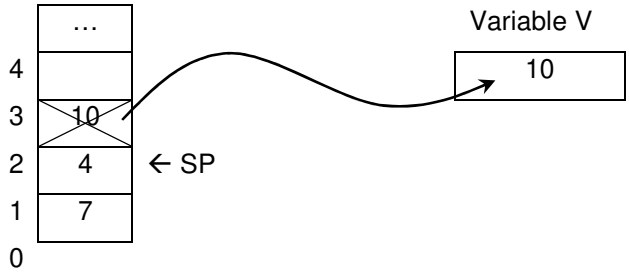
3- Insertion du 2e élément (valeur 4) :



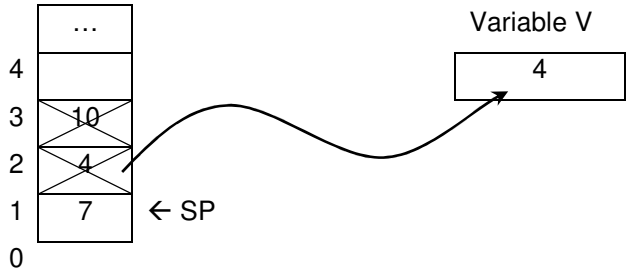
4- Insertion du 3e élément (valeur 10) :



5- Retrait d'un élément : La valeur de l'élément retiré (le dernier élément inséré) est sauvegardée dans une variable V, le sommet de pile SP, se déplace d'une position.



6- Retrait d'un deuxième élément : La valeur de l'élément retiré est sauvegardée dans une variable V, le sommet de pile SP, se déplace d'une position.



7.3- Modèle

On définit sur les piles une machine abstraite munie de l'ensemble des opérations suivantes :

①- CreerPile(P) : Créer une pile vide. Permet d'initialiser la pile P.

②- Empiler(P, val) : Ajouter la valeur val en sommet de la pile P.

③- Depiler(P, val) : Retirer dans la variable val l'élément en sommet de de la pile P.

④- PileVide(P) : Une fonction booléenne permettant de tester si la pile P est vide.

⑤- PilePleine(P) : Une fonction booléenne permettant de tester si la pile P est pleine.

7.4- Implémentation

Une Pile peut être implémentée au moyen de :

- Un tableau (Manière statique)
- Une Liste Linéaire chaînée (Manière dynamique)

7.4.1- Implémentation d'une Pile en utilisant une Liste

Dans ce cas, la Pile est exactement une Liste où les insertions et les suppressions se font en tête. On n'utilisera alors que les procédures (sur les listes) ***InsererTete*** et ***SupprimerTete***.

Le type Pile peut être déclaré comme suit :

```
Type   Liste = ^Element
       Element = Enregistrement
           Val : Entier
           Suiv : Liste
       Fin
       Pile = Liste    // Une pile est exactement une liste.
Var    P : Pile        // Déclaration d'une Pile P.
```

Le modèle défini sur les Piles peut être implémenté comme suit :

```
Procédure CreerPile(Var P :Pile)
Début
    P ← Nil         // Initialisation d'une pile vide.
Fin
Procédure Empiler(Var P : Pile ; Val : Entier)
Début
    Si NON PilePleine(P) Alors
        InsererTete(P, Val)
    /* Sinon
        Ecrire("Empiler – Erreur : Pile Pleine") */
    FSi
Fin
Procédure Depiler(Var P : Pile ; Var Val : Entier)
Début
    Si NON PileVide(P) Alors
        Val ← P^.Val
        SupprimerTete(P)
    /* Sinon
        Ecrire("Depiler - Erreur : Pile vide") */
    FSi
Fin
Fonction PileVide(P :Pile) : Booléen
Début
    PileVide ← P = Nil
Fin
```

```
Fonction PilePleine(P :Pile) : Booléen
Début
    PilePleine ← Faux     // On suppose que la taille de la mémoire est illimitée.
Fin
```

7.4.2- Implémentation d'une Pile en utilisant un tableau

En utilisant un tableau pour implémenter une Pile, on a besoin d'un tableau et d'un indice (SP : Sommet de pile) qui indique quel est l'élément qui est en tête de la pile.

Le type Pile peut être déclaré comme suit :

```
Const N = 50 // Taille Max de la pile.
Type  Pile = Enregistrement
          TAB : Tableau[1 .. N] d'entier
          SP : Entier     // Sommet de Pile
      Fin
Var   P : Pile          // Déclaration d'une Pile P.
```

Le modèle défini sur les piles peut être implémenté comme suit :

```
Procédure CreerPile(Var P :Pile)
Début
    P.SP ← 0       // Il n'y a aucun élément dans la pile.
Fin
Procédure Empiler(Var P : Pile ; Val : Entier)
Début
    Si NON PilePleine(P) Alors
        P.SP ← P.SP + 1
        P.TAB[ P.SP ] ← Val
    /* Sinon
        Ecrire("Empiler – Erreur : Pile Pleine") */
    FSi
Fin
Procédure Depiler(Var P : Pile ; Var Val : Entier)
Début
    Si NON PileVide(P) Alors
        Val ← P.TAB[ P.SP ]
        P.SP ← P.SP – 1
    /* Sinon
        Ecrire("Depiler - Erreur : Pile vide") */
    FSi
Fin
```

```
Fonction PileVide(P :Pile) : Booléen
Début
    PileVide ← P.SP = 0  // Tableau vide.
Fin
Fonction PilePleine(P :Pile) : Booléen
Début
    PilePleine ← P.SP = N        // Tableau plein, il contient N éléments.
Fin
```

7.5- Synthèse

- ✓ Une pile est une structure abstraite.
- ✓ Dans une pile les insertions et les suppressions d'éléments se font à la même extrémité. Donc ses opérations suivent le principe LIFO (Last In First Out, Dernier entré, Premier Servi).
- ✓ On définit sur les piles une machine abstraite munie de l'ensemble des opérations suivantes : CreerPile (Créer une pile vide), Empiler (Ajouter un élément), Depiler (Retirer un élément), PileVide(Vérifier si la pile est vide), PilePleine (Vérifier si la pile est pleine).
- ✓ Une pile peut être implémenté en utilisant une LLC (Liste Linéaire Chaînée) pour la version dynamique ou avec un tableau pour la version statique.

8. FILES

8.1- Définition, principe, domaine d'application

Une file d'attente peut être définie comme une collection d'éléments dans laquelle tout nouvel élément est inséré à la fin et tout élément ne peut être supprimé que du début. Elle applique le principe « FIFO » pour « First In First Out » qui veut dire « premier entré, premier servi »

La file d'attente est très utilisée dans les systèmes d'exploitation des ordinateurs et surtout dans les problèmes de simulation.

8.2- Exemple

Pour expliciter le fonctionnement d'une file, prenons l'exemple suivant : On suppose qu'on a une file et on souhaite insérer les trois éléments (des valeurs entières) 7, 4 et 10 et puis en retirer deux éléments. Chaque insertion se fait à la fin et chaque retrait se fait par le début. On peut schématiser ces opérations comme suit :

1- Etat initial de la file : Au départ, la file est vide, elle ne contient aucun élément.

1	2	3	4	…

NombreElements = 0

2- Insertion du 1er élément (valeur 7) :

1	2	3	4	…
7				

NombreElements = 1

3- Insertion du 2e élément (valeur 4) :

1	2	3	4	…
7	**4**			

NombreElements = 2

4- Insertion du 3e élément (valeur 10) :

1	2	3	4	…
7	**4**	**10**		

NombreElements = 3

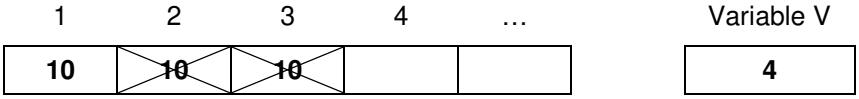
5- Retrait d'un élément : La valeur de l'élément qui se trouve en première position dans la file est sauvegardée dans une variable V. Les autres éléments sont décalés d'un pas. Le nombre d'éléments est décrémenté de un.

1	2	3	4	…
4	**10**	~~**10**~~		

Variable V
7

NombreElements = 2

6- Retrait d'un deuxième élément : même chose que 5-.



NombreElements = 1

8.3- Modèle

On définit sur les files une machine abstraite munie de l'ensemble des opérations suivantes :

①- CreerFile(F) : Créer une file vide. Permet d'initialiser la file F.

②- Enfiler(F, val) : Ajouter val en queue (à la fin) de la file F.

③- Defiler(F, val) : retirer dans la variable val l'élément qui est en tête de la file F.

④- FileVide(F) : Une fonction booléenne permettant de tester si la file F est vide.

⑤- FilePleine(F) : Une fonction booléenne permettant de tester si la file Fest pleine.

8.4- Implémentation

Une file peut être implémentée au moyen de :

- Un tableau (Manière statique)
- Une Liste Linéaire chaînée (Manière dynamique)

8.4.1- Implémentation d'une File en utilisant une Liste

Dans ce cas la file est exactement une liste ou les insertions se font à la fin et les suppressions se font en tête. On n'utilisera alors que les procédures (sur les listes) **InsererQueue** et **SupprimerTete**.

Le type File peut être déclaré comme suit :

```
Type   Liste = ^Element
       Element = Enregistrement
           Val : Entier
           Suiv : Liste
       Fin
       File = Liste    // Le type File est exactement une Liste.
Var    F : File        // F est une File (Implémentée par une liste)
```

Implémentation du modèle (Version 1)

Le modèle défini sur les files peut être implémenté comme suit :

```
Procédure CreerFile(Var F :File)
Début
    F ← Nil // Initialisation d'une file vide.
Fin
Procédure Enfiler(Var F : File ; Val : Entier)
Début
    Si NON FilePleine( F ) Alors
        InsererQueue(F, Val)
```

```
        /* Sinon
        |       Ecrire("Enfiler - Erreur : File Pleine") */
        FSi
Fin
Procédure Defiler(Var F : File ; Var Val : Entier)
Début
        Si NON FileVide(F) Alors
        |       Val ← F^.Val
        |       SupprimerTete(F)
        /* Sinon
        |       Ecrire("Defiler - Erreur : File vide") */
        FSi
Fin
Fonction FileVide(F :File) : Booléen
Début
        FileVide ← F = Nil              // Ou
Fin
```

```
Si F = Nil Alors
|       FileVide ← Vrai
Sinon
|       FileVide ← Faux
FSi
```

```
Fonction FilePleine(F :File) : Booléen
Début
        FilePleine ← Faux // On suppose que la taille de la mémoire est illimitée.
Fin
```

Remarque : Pour travailler efficacement avec une file, il est préférable de sauvegarder la tête et la queue. Cela permet d'insérer directement après l'élément qui est en fin de la liste, sans avoir besoin de parcourir toute la liste pour insérer un nouvel élément à la fin de la liste (file). Dans ce cas, la file sera considérée comme un enregistrement composé de deux champs de type pointeur (Liste) :

①- Le premier (appelé Tete) pointe sur le premier élément de la liste,

②- Le deuxième (appelé Queue) pointe sur le dernier élément de la liste.

Le nouveau type File peut être déclaré comme suit :

```
Type    Liste = ^Element
        Element = Enregistrement
        |       Val : Entier
        |       Suiv : Liste
        Fin
        File = Enregistrement
        |       Tete, Queue : Liste
        Fin
Var     F : File // Déclaration d'une File F.
```

Implémentation du modèle (Version 2)

Le modèle défini sur les files devient :

```
Procédure CreerFile(Var F :File)
Début
    F.Tete ← Nil   // Initialisation d'une file vide.
    F.Queue ← Nil
Fin
Procédure Enfiler(Var F : File ; Val : Entier)
Début
    Si NON FilePleine( F ) Alors
        InsererQueue( F.Queue, Val )
        Si F.Tete = Nil Alors
            F.Tete ← F.Queue // Après insertion du premier élément seulement.
        FSi
    /* Sinon
        Ecrire("Enfiler - Erreur : File Pleine") */
    FSi
Fin
Procédure Defiler(Var F : File ; Var Val : Entier)
Début
    Si NON FileVide(F) Alors
        Val ← F.Tete^.Val
        SupprimerTete(F.Tete)
        Si F.Tete = Nil Alors
            F.Queue ← Nil // Après suppression du dernier élément seulement.
        FSi
    /* Sinon
        Ecrire("Defiler - Erreur : File vide") */
    FSi
Fin
Fonction FileVide(F :File) : Booléen
Début
    FileVide ← F.tete = Nil        Ou
Fin
Fonction FilePleine(F :File) : Booléen
Début
    FilePleine ← Faux     // On suppose que la taille de la mémoire est illimitée.
Fin
```

```
Si F.tete = Nil Alors
    FileVide ← Vrai
Sinon
    FileVide ← Faux
FSi
```

8.4.2- Implémentation d'une File en utilisant un tableau

En utilisant un tableau pour implémenter une file, on a deux modes de fonctionnement :

1- Utilisation des décalages : A chaque défilement (suppression de l'élément en tête (Tab[1])), tous les autres éléments seront décalés vers la gauche.

2- Utilisation circulaire du tableau : A chaque défilement, les éléments du tableau ne sont plus décalés, mais l'indice du début des éléments dans le tableau change. Il ne sera pas toujours 1.

Implémentation du modèle (Version 1 :Utilisation des décalages)

```
Const N = 50 // Taille Max de la file *)
Type   File = Enregistrement
            TAB : Tableau[1 .. N] d'entier       // Eléments dans la file.
            NB : Entier    // Nombre d'éléments dans la file.
       Fin
Var    F : File          // Déclaration d'une File F)
```

Le modèle défini sur les files peut être implémenté comme suit :

```
Procédure CreerFile(Var F :File)
Début
    F.NB ← 0 // Aucun élément n'est dans la file = file vide.
Fin
Procédure Enfiler(Var F : File ; Val : Entier)
Début
    Si NON FilePleine(F) Alors
        F.NB ← F.NB + 1
        F.TAB[ F.NB ] ← Val
    /* Sinon
        Ecrire("Enfiler - Erreur : File Pleine") */
    FSi
Fin
Procédure Defiler(Var F : File ; Var Val : Entier)
Var I : Entier
Début
    Si NON FileVide(F) Alors
        Val ← F. TAB[ 1 ] (* Retirer le premier élément *)
        F.NB ← F.NB – 1
        (* Décaler tous les autres éléments vers la gauche *)
        Pour I ← 1 à F.NB Faire
            F.TAB[ I ] ← F. TAB[ I + 1]
        FPour
```

```
        /* Sinon
            Ecrire("Defiler - Erreur : File vide") */
        FSi
Fin
```

```
Fonction FileVide(F :File) : Booléen
Début
        FileVide ← F.NB = 0          Ou
Fin
```

```
Si F.NB = 0 Alors
        FileVide ← Vrai
Sinon
        FileVide ← Faux
FSi
```

```
Fonction FilePleine (F :File) : Booléen
Début
        FilePleine ← F.NB = N        Ou
        (* Si le tableau est plein *)
Fin
```

```
Si F.NB = N Alors
        FilePleine ← Vrai
Sinon
        FilePleine ← Faux
FSi
```

Implémentation du modèle (Version 2 :Utilisation circulaire du tableau)

On a besoin de sauvegarder les indices du début et la fin des éléments (comme pour la tête et la queue d'une liste) et laisser une case vide dans le tableau pour différencier le cas où la file est (presque) pleine (contient N-1 élément) et le cas où la file est vide (où les indices début et fin sont égaux).

Le type file devient :

```
Const N = 50 (* Taille Max de la file *)
Type File = Enregistrement
                TAB : Tableau[1 .. N] d'entier
                iDebut : Entier (* Indice début des éléments *)
                iFin : Entier (* Indice fin des éléments *)
            Fin
Var F : File (* Déclaration d'une File F *)
```

Le modèle défini sur les files devient :

```
Procédure CreerFile(Var F :File)
Début
        F.iDebut ← 1 (* N'importe quelle valeur ∈[1 ..N] à condition que F.iDebut = F.iFin *)
        F.iFin ← F.iDebut
Fin
Procédure Enfiler(Var F : File ; Val : Entier)
Début
        Si NON FilePleine(F) Alors
            F.TAB[ F.iFin ] ← Val
            F.iFin ← (F.iFin MOD N) + 1 (* Revenir au début du tableau si nécessaire *)
```

```
        /* Sinon
            Ecrire("Enfiler - Erreur : File Pleine") */
        FSi
Fin
Procédure Defiler(Var F : File ; Var Val : Entier)
Début
        Si NON FileVide(F) Alors
            Val ← F. TAB[ F.iDebut ] (* Retirer l'élément en tête de la file*)
            F.iDebut ← (F.iDebut MOD N) + 1
            // Revenir au début du tableau si nécessaire
        /* Sinon
            Ecrire("Defiler - Erreur : File vide") */
        FSi
Fin
Fonction FileVide(F :File) : Booléen
Début
        FileVide ← F.iDebut = F.iFin
Fin
Fonction FilePleine (F :File) : Booléen
Début
        FilePleine ← (F.iFin + N – F.iDebut) MOD N = N – 1 (* iFin peut être < iDebut *)
Fin
```

8.5- File avec priorité

Une file d'attente avec priorité est une file d'attente dans laquelle l'opération de défilement récupère l'élément le plus prioritaire. Définir le modèle et l'implémenter.

Le type File peut être déclaré comme suit :

```
Type   Liste = ^Element
       Element = Enregistrement
                    Val : Entier (* L'information utile *)
                    Prio : Entier (* La priorité de la valeur Val *)
                    Suiv : Liste
                Fin
       File = Liste
```

On peut imaginer deux scénarios :

1- Les insertions se font en queue (normalement) et la fonction de défilement cherche l'élément le plus prioritaire (l'élément avec la valeur Max de la priorité). Si plusieurs éléments

ont la même priorité Max, c'est le premier élément avec la priorité Max qui sera retiré car les éléments sont insérés dans leurs ordres d'arrivée.

2- Les insertions se font par ordre de priorité (c'est exactement une insertion dans une liste triée sur la valeur du champ Prio au lieu du champ Val et les suppressions se font (normalement) en tête de la liste.

Remarque : l'implémentation d'une file avec priorité fera l'objet d'un exercice dans la partie Exercices corrigés.

8.6- Synthèse

- ✓ Une file est une structure abstraite. Elle peut être définie comme une collection d'éléments dans laquelle tout nouvel élément est inséré à la fin et tout élément ne peut être retiré que du début. Elle applique le principe « FIFO » pour « First In First Out » qui veut dire « premier entré, premier servi »
- ✓ On définit sur les Files une machine abstraite munie de l'ensemble des opérations suivantes : CreerFile (Créer une file vide), Emfiler (Ajouter un élément), Defiler (Retirer un élément), FileVide(Vérifier si la file est vide), FilePleine (Vérifier si la file est pleine).
- ✓ Une pile peut être implémenté en utilisant une LLC (Liste Linéaire Chaînée) pour la version dynamique ou avec un tableau pour la version statique.
- ✓ Pour l'implémentation dynamique de la file, une file n'est autre qu'une liste où les insertions se font à la fin et les suppressions se font en tête. Mais pour faciliter les insertions à la fin, il est conseillé d'utiliser deux pointeurs : un pointeur pour la tête de la liste (qui pointe sur le 1er élément) et un deuxième pour la queue de la liste (qui pointe sur le dernier élément).
- ✓ Pour l'implémentation statique de la file, un tableau est utilisé. La solution la plus directe est d'insérer chaque nouvel élément à la fin du tableau (derrière le dernier élément inséré) et en cas de suppression de l'élément en tête (à l'indice 1), tous les autres éléments qui le suivent sont décalés vers la gauche. Comme dans une file d'attente réelle (à un bureau de poste, par exemple), quand la première personne va au guichet, toutes les autres personnes avancent d'un pas.
- ✓ Une file d'attente avec priorité est une file d'attente dans laquelle l'opération de défilement récupère l'élément le plus prioritaire.

9. Les arbres (introduction)

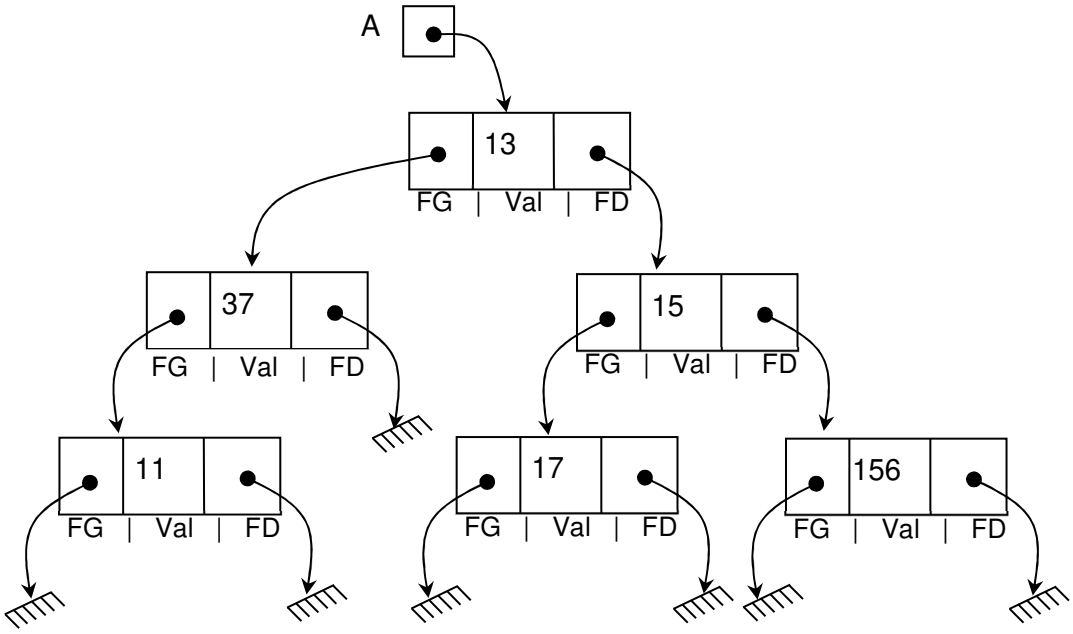
Définition

Un arbre est une structure de données hiérarchique, généralement dynamique.
Un arbre peut être considéré comme une liste chaînée non linéaire d'éléments (maillons, nœuds, cellules). Ces derniers forment un graphe orienté où chaque nœud a au plus 1 prédécesseur (appelé parent) et n (n ≥ 0) successeurs (appelés fils).

En pratique un arbre est représenté de haut en bas. Ce qui permet de ne pas orienter les sens des arcs.

Si le nombre de successeurs de tout nœud est au plus égal à 2, l'arbre est dit binaire. En général, les deux fils sont appelés FD (Fils Droit) et FG (Fils Gauche).

Ci-dessous un exemple d'arbre binaire contenant 6 éléments :



PARTIE 3– EXERCICES AVEC SOLUTIONS

Introduction

Remarque 1 : Opérations sur les nombres entiers DIV et MOD et comment extraire les chiffres un par un à partir de la droite pour réaliser les différentes fonctions : nombre de chiffres, somme de chiffres, produit des chiffres, existence d'un chiffre …
Remarque 2 : sur le choix de la version itérative ou récursive selon le contexte de l'examen
Remarque 3 : sur l'option de mettre toutes les variables globales au début de l'algorithme.
Remarque 4 : sur le rappel des procédures et fonctions. Comme les fonctions et procédures sont écrites en réponse à une question, donc, j'ai opté pour le rappel du prototype de la fonction ou procédure suivi de trois points pour dire que la fonction ou la procédure doit être réécrite entièrement à l'endroit du prototype.
Remarque 5 : Les exercices sont écrits selon un enchaînement logique, pour répondre à une question, il est possible (et/ou nécessaire) d'utiliser les fonctions et/ou procédures des questions précédentes à conditions de respecter leurs déclarations.

Exercice 1 : Multiples de 2 et Multiples de 3

(Adapté à partir de l'examen du S2 du 20/05/2015)

Q1) Ecrire une fonction ***multiple2*** qui vérifie si un entier **N** est multiple de **2**.

Q2) Ecrire une fonction ***multiple3*** qui vérifie si un entier **N** est multiple de **3**.

Q3) En utilisant les deux fonctions ***multiple2*** et ***multiple3***, écrire un algorithme principal qui lit un nombre entier et qui précise s'il est pair, multiple de 3 et/ou multiple de 6. Si ce nombre n'est ni pair, ni multiple de 3, ni multiple de 6, on n'affiche rien (exemple 3).

Exemple 1 :

Donnez un entier : 12
Il est pair
Il est multiple de 3
Il est divisible par 6

Exemple 2 :

Donnez un entier : 9
Il est multiple de 3

Exemple 3 :

Donnez un entier : 5

(On n'affiche rien)

Solution

Q1) Fonction multiple2(N : Entier) : Booléen
Début
    Si N **Mod** 2 = 0 Alors
        multiple2← Vrai
    Sinon
        multiple2← Faux
    FSi
    // OU multiple2 ← N **Mod** 2 = 0
Fin

Q2) Fonction multiple3(N : Entier) : Booléen

```
Début
    Si N Mod 3 = 0 Alors
        multiple3← Vrai
    Sinon                          // OU multiple3 ← N Mod 3 = 0
        multiple3← Faux
    FSi
Fin
```

Q3) Algorithme Affichage

```
Var     N : Entier // Déclaration des variables globales
/* Rappel des fonctions */
Fonction multiple2(N : Entier) : Booléen …
Fonction multiple3(N : Entier) : Booléen
Début /* Algorithme principal */
    Ecrire("Donnez un entier : ")
    Lire( N )
    Si multiple2(N) Alors
        Ecrire("Il est pair")
    FSi
    Si multiple3(N) Alors
        Ecrire("Il est multiple de 3")
    FSi
    Si multiple2(N) et multiple3(N) Alors
        Ecrire("Il est divisible par 6")
    FSi
Fin /* Algorithme principal */
```

Exercice 2 : Calcul d'une somme de fractions

(Adapté à partir de l'examen du S2 du 17/05/2016)

On souhaite calculer la somme suivante : $S = 1 + \frac{2^2}{2!} + \frac{3^3}{3!} + \cdots \frac{n^n}{n!},\ \ \mathrm{n} > 3$

Q1) Ecrire une **fonction itérative *Facto(N)*** qui permet de calculer la factorielle d'un entier **N**.

Q2) Ecrire une **fonction récursive *Puiss(x, n)*** qui permet de calculer **x** à la puissance **n** (**x** est un réel) en sachant que :

$$x^n = \begin{cases} 1 & Si\ n = 0 \\ x^{n/2} \times x^{n/2} & Si\ n > 0\ et\ \ n\ \ pair \\ x^{(n-1)/2} \times x^{(n-1)/2} \times x & Si\ n > 0\ et\ \ n\ \ impair \end{cases}$$

Q3) Ecrire une **procédure *Somme()*** qui lit un entier **N > 3** et calcule la somme **S**.

Solution

Q1) Fonction Facto(N : Entier) : Entier

```
Var     F, I : Entier
Début
        F ← 1
        Pour I ← 2 à N Faire
                F ← F * I
        FPour
        Facto ← F
Fin
```

Q2) Fonction Puiss(x : Réel, n : Entier) : Réel

```
Var     R : Réel
Début
        Si n = 1 Alors
                R ← 1
        Sinon
                R ← Puiss( x, n Div 2)
                R ← R * R
                Si n Mod 2 = 1 Alors
                        R ← R * x
                FSi
        FSi
        Puiss ← R
Fin
```

Q3) Procédure Somme()

```
Var     N, I : Entier
        S : Réel
Début
        Répéter
                Ecrire("Donner un entier N > 3")
                Lire( N )
        Jusqu'à N > 3
        S ← 0
        Pour I ← 1 à N Faire
                S ← S + Puiss( I, I ) / Facto( I )
        FPour
        Ecrire("La somme S = ", S)
Fin
```

Exercice 3 : Nombres Amis

(Adapté à partir de l'examen de rattrapage du S1 du 14/03/2016)

Deux nombres entiers strictement positifs ***N1*** et ***N2*** et ***N1 ≠ N2*** sont dits **nombres amis** si la somme des diviseurs propres de ***N1*** (*N1 est exclus de la somme*) est égale à ***N2*** et la somme des diviseurs propres de ***N2*** (*N2 est exclus de la somme*) est égale à ***N1***.

Exemple : **220** et **284** sont amis car :

- Les diviseurs propres de **220** sont : **1, 2, 4, 5, 10, 11, 20, 22, 44, 55 et 110** (**220** n'est pas pris en compte) et **1 + 2 + 4 + 5 + 10 + 11 + 20 + 22 + 44 + 55 + 110 = 284**
- Les diviseurs propres de **284** sont : **1, 2, 4, 71, 142** (**284** n'est pas pris en compte) et **1 + 2 + 4 + 71 + 142 = 220**

Q1) Ecrire une fonction **SomDiviseurs(N)** qui permet de calculer la somme des diviseurs propres d'un entier strictement positif **N** (*N est exclus de la somme*).

Q2) Ecrire une fonction **Amis**(***N1, N2)*** qui vérifie si deux entiers **N1** et **N2** sont amis ou pas.

Q3) Ecrire un algorithme principal qui permet de trouver et compter tous les nombres amis inférieurs à un nombre **NMax** (Exemple **NMax = 10000**). Chaque deux nombres amis, ne doivent pas être affichés en double. Exemple : 220 et 284 sont amis il ne faut pas les afficher deux fois : (220 et 284) et (284 et 220).

Solution

Q1) Fonction **SomDiviseurs**(N : Entier) : Entier

```
Var     S, D : Entier
Début
    S ← 0 // N est exclus de la somme ; Si N = 1 alors, il faut que la somme soit nulle.
    Pour D ← 1 à N div 2 Faire
        Si N Mod D = 0 Alors // Si D est un diviseur de N.
            S ← S + D
        FSi
    FPour
    SomDiviseurs ← S
Fin
```

Q2) Fonction **Amis**(N1, N2 : Entier) : Booléen

```
Début
    Si (SomDiviseurs(N1) = N2) ET (SomDiviseurs(N2) = N1) alors
        Amis ← Vrai
    Sinon
        Amis ← Faux
    FSi
    // Ou tout simplement : Amis←(SomDiviseurs(N1) = N2) ET (SomDiviseurs(N2) = N1)
Fin
```

Q3) Algorithme NombresAmis

```
Var     NMax, N1, N2, SD, Cpt : Entier /* Déclaration des variables globales */
/* Rappel des fonctions et procédures */
Fonction SomDiviseurs(N : Entier) : Entier …
Fonction Amis(N1, N2 : Entier) : Booléen …
Début /* Algorithme principal */
    Répéter
        Ecrire("Donner la valeur de la limite NMAX : ")
        Lire(NMax)
    Jusqu'à NMax > 0
    Cpt ← 0
    Pour N1 ← 1 à (NMAX – 1) Faire
        SD ← SomDiviseurs(N1)
        Si (SD < NMAX) ET (SD > N1) Alors
            /* - SD < NMAX alors les nombres (N1 et SD) sont des amis éventuels.
            - Si SD > NMAX alors SD est en dehors de la limite.
            - La condition (SD > N1) ou (SD < N1) permet d'éviter les doublons. */
            SD ← SomDiviseurs(SD)      // N2 = SD
            Si SD = N1 Alors
                Ecrire(N1, " et ", SD, " sont amis")
                Cpt ← Cpt + 2 // les nombres amis sont affichés deux à deux.
            FSi
        FSi
    FPour
    Ecrire("Il existe ", Cpt, " nombres amis")
Fin
```

Exercice 4 : Nombres Palindromes

(Adapté à partir de l'examen de remplacement du S2 du 18/06/2018)

Q1) Écrire une **fonction *ImageMiroir(N)*** qui permet de calculer l'**image miroir** d'un entier positif **N** en inversant ses chiffres. C'est-à-dire, le premier chiffre devient dernier, le dernier devient premier, le deuxième devient avant dernier et l'avant dernier devient deuxième et ainsi de suite. **Exemple** : Si **N = 12345** Alors son image miroir est **54321**

Un nombre **palindrome** est un nombre dont les chiffres peuvent se lire de droite à gauche ou de gauche à droite. En d'autres termes, c'est un nombre égal à son image miroir.

Exemples : **1, 77, 323, 1221, 58485, 956659** sont des nombres palindromes.

Q2) Écrire une **fonction *Palindrome(N)*** qui vérifie si **N** est palindrome ou pas.

Q3) Écrire une **procédure *Afficher(NBChiffres)*** qui affiche tous les nombres palindromes qui ont une longueur de **NBChiffres**.

Exemples : **101, 111, 717** sont des nombres palindromes de trois chiffres **(NBChiffres = 3)**.

Q4) Écrire un algorithme principal permettant d'afficher tous les nombres palindromes qui ont au moins NMin chiffres et au plus NMax chiffres.

Solution

Q1) Fonction **ImageMiroir**(N : Entier) : Entier

```
Var     M : Entier
Début
        M ← 0
        TQ N ≠ 0 Faire
                M ← M * 10 + N Mod 10
                N ← N Div 10
        FTQ
        ImageMiroir ← M
Fin
```

Q2) Fonction **Palindrome**(N : Entier) : Booléen

```
Début
        Palindrome ← N = ImageMiroir(N)
        /*      Si N = ImageMiroir(N) Alors Palindrome ← Vrai
                Sinon Palindrome ← Faux FSi         */
Fin
```

Q3) Procédure **Afficher**(NBChiffres : Entier)

```
Var     N, NMin, NMax : Entier
Début
        NMin ← 1
        Pour N ← 1 à (NBChiffres – 1) Faire
                NMin ← NMin * 10
        Fpour
        NMax ← NMin * 10 – 1
        Pour N ← NMin à NMax Faire
                Si Palindrome(N) Alors
                        Ecrire(N)
                Fsi
        Fpour
Fin
```

Q4) Algorithme NombresPalindromes

```
Var     NMin, NMax, NBChiffres : Entier // Déclaration des variables globales
Fonction ImageMiroir(N : Entier) : Entier …
Fonction Palindrome(N : Entier) : Booléen …
Procédure Afficher(NBChiffres : Entier) …
Début
    Repéter
        Ecrire("Donner deux entiers NMin > 1 et NMax > 1")
        Lire(NMin, NMax)
    Jusqu'à (NMin > 1) ET (NMax > 1)
    Pour NBChiffres ← NMin à NMax Faire
        Afficher( NBChiffres )
    FPour
Fin
```

Exercice 5 : Nombres d'Armstrong

(Adapté à partir de l'Interrogation S2 du 22/06/2019)

Q1) Pour un entier positif **N**, écrire une **fonction récursive *NombreChiffres(N)*** qui calcule le nombre de chiffres de **N**. **Exemple** : Si **N = 371** alors ***NombreChiffres(371)*** retourne **3**.

Q2) Pour un entier **A**, écrire une **fonction itérative *puiss(A, N)*** qui permet de calculer **A** à la puissance **N** (**N ≥ 0**), *c'est-à-dire* : $\mathbf{A^N = A * A * \dots * A}$, **en respectant la méthode suivante** :

1. On initialise le résultat à **1** (Parce que $\mathbf{A^0 = 1}$),
2. Si **N** est impair on multiplie le résultat précédent par **A** et on remplace **N** par (**N - 1**),
3. Si **N** est pair on remplace **A** par $\mathbf{A^2}$ et **N** par (**N** / **2**),
4. On répète les étapes **2** et **3** jusqu'à ce que **N** devienne nul.

Exemple Pour calculer $\mathbf{A^N = 3^7}$ on fait comme suit :

- On initialise le résultat à **1**,
- N = 7, N est impair, on multiplie le résultat précédent par **3** et on remplace N par N -1 → **N = 6**
- N = 6, N est pair on remplace A par A^2 → $\mathbf{A = 3^2 = 9}$ et on remplace N par N/2 → **N = 3**
- N = 3, N est impair, on multiplie le résultat précédent par 9 et on remplace N par N -1 → **N = 2**
- N = 2, N est pair on remplace A par A^2 → $\mathbf{A = 9^2 = 81}$ et on remplace N par N/2 → **N = 1**
- N = 1, N est impair, on multiplie le résultat précédent par 81 et on remplace N par N -1 → **N = 0**
- N = 0, on arrête les calculs. Au total on a : $\mathbf{A^N} = 3 \times 9 \times 81 = 3^1 \times 3^2 \times 3^4 = 3^7$

Un nombre entier strictement positif **N** à **k** chiffres est dit **nombre d'Armstrong** si la somme des chiffres de **N** élevés à la puissance **k** est égale à **N** lui-même.

Ci-dessous quelques exemples :

- Si **k = 1** : **1, 2, 3, 4, 5, 6, 7, 8, 9** sont tous des nombres d'Armstrong car $\mathbf{1^1 = 1, 2^1 = 2, \dots, 9^1 = 9.}$

- Si **k = 3** : **371** est un nombre d'Armstrong car $3^3 + 7^3 + 1^3 = 27 + 343 + 1 = 371$.
- Si **k = 4** : **1634** est un nombre d'Armstrong car $1^4 + 6^4 + 3^4 + 4^4 = 1 + 1296 + 81 + 256 = 1634$.

Q3) Écrire une **fonction itérative *ARM(N)*** qui permet de vérifier si un entier strictement positif **N** est un nombre d'Armstrong ou non.

Q4) Écrire une **procédure itérative *Affichage(kMin, kMax, Cpt)*** qui permet d'afficher et compter les nombres d'Armstrong qui ont au moins ***kMin*** chiffres et au plus ***kMax*** chiffres.

Exemple : Si on souhaite chercher les nombres d'Armstrong qui ont entre **3 et 5** chiffres, la procédure ***Affichage*** doit afficher les nombres suivants : **153, 370, 371, 407, 1634, 8208, 9474, 54748, 92727, 93084.**

Cette procédure doit aussi compter les nombres d'Armstrong trouvés, donc elle doit retourner le nombre **10**.

Q5) Écrire l'algorithme principal qui permet de lire deux entiers strictement positifs ***k1*** et ***k2*** tels que ***k1 ≤ k2*** puis affiche tous les nombres d'Armstrong qui ont entre ***k1*** et ***k2*** chiffres ainsi que leur nombre.

Solution

Q1) Fonction **NombreChiffres** (N : Entier) : Entier

```
Début
    Si N <10 Alors          // S'il y a un seul chiffre
        NombreChiffres ← 1
    Sinon
        NombreChiffres ← 1 + NombreChiffres (N Div 10)
    FSi
Fin
```

Q2) Fonction **puiss**(A, N : Entier) : Entier

```
Var     Resultat : Entier
Début
    Resultat ← 1
    TQ N ≠ 0 Faire
        Si N Mod 2 = 0 Alors
            A ← A * A
            N ← N Div 2
        Sinon
            Resultat ← Resultat * A
            N ← N – 1
        FSi
    FTQ
    puiss ← Resultat
Fin
```

Q3) Fonction **ARM**(N : Entier) : Booléen

```
Var     NC, K, Somme : Entier
Début
    K ← NombreChiffres(N)
    NC ← N // Garder une copie de N
    Somme ← 0
    TQ NC ≠ 0 Faire
        Somme ← Somme + puiss( NC Mod 10, K)
        NC ← NC Div 10
    FTQ
    ARM ← Somme = N // ≡ Si Somme = N Alors ARM ← Vrai Sinon ARM ← Faux FSi
Fin
```

Q4) Procédure **Affichage**(kMin, kMax : Entier ; **Var** Cpt : Entier)

```
Var     N, NMin, NMax : Entier
Début
    NMin ← puiss( 10, kMin - 1)
    NMax ← puiss( 10, kMax) - 1
    Cpt ← 0
    Pour N ← NMin à NMax Faire
        Si ARM( N ) Alors
            Ecrire( N )
            Cpt ← Cpt +1
        FSi
    FPour
Fin
```

Q5) Algorithme NombresArmstrong

```
Var     k1, k2, Cpt : Entier /* Déclaration des variables globales */
/* Rappel des fonctions et procédures */
Fonction NombreChiffres (N : Entier) : Entier …
Fonction puiss(A, N : Entier) : Entier …
Fonction ARM(N : Entier) : Booléen …
Procédure Affichage(kMin, kMax : Entier ; Var Cpt : Entier) …
Début /* Algorithme principal */
    Répéter
        Ecrire("Donner k1 et k2 les nombres de chiffres minimum et maximum des nombres à tester : ")
        Lire(k1, k2)
    Jusqu'à k1 > 0 et k1 ≤ k2)
```

```
    Affichage(k1, k2, Cpt)
    Ecrire("Il y a ", Cpt, " nombres d'Armstrong qui ont entre", k1, " et ", k2, " chiffres")
Fin /* Algorithme principal */
```

Exercice 6 : Nombres Distincts Et Nombre Bien Ordonnés

(Adapté à partir de l'examen du S2 du 09/07/2019)

Partie I : Un entier positif **N** est dit **distinct** s'il est composé de chiffres tous différents. C'est-à-dire, **N** est composé de chiffres que chacun d'eux n'apparait qu'une seule fois.

Exemples : - **257** est distinct car composé des chiffres **2, 5 et 7** qui sont tous différents.
- **32064** est distinct car composé des chiffres **3, 2, 0, 6 et 4** qui sont tous différents.
- **881** n'est pas distinct car le chiffre **8** apparaît deux (2) fois.

Q1) Ecrire une **fonction *NBExiste(C, N)*** qui permet de compter combien de fois le chiffre **C** ($0 \leq C \leq 9$) apparait dans l'entier **N**.

Q2) Ecrire une **fonction *estDistinct(N)*** qui vérifie si un entier **N** est **distinct.**

Q3) Écrire une **procédure *Affichage1( )*** qui permet de compter et d'afficher tous les nombres distincts de 3 chiffres.

Partie II : Un entier **N** est **bien ordonné** si ses chiffres forment, de gauche à droite, une suite strictement croissante.

Exemples : - L'entier de 3 chiffres, **125** est bien ordonné car **1 < 2 < 5**.
- L'entier de 4 chiffres, **2476** n'est pas bien ordonné car 2 < 4 < 7 **mais 7 > 6.**

Q4) Ecrire une **fonction *estBien(N)*** (itérative ou récursive) qui vérifie si un entier **N** est **bien ordonné.**

Q5) Écrire une **procédure *Affichage2( )*** qui permet de compter et d'afficher tous les nombres bien ordonnés.

Q6) Écrire un ***algorithme principal*** qui permet de : ①- Afficher tous les nombres distincts de 3 chiffres ainsi que leur nombre, ②- Afficher tous les nombres bien ordonnés ainsi que leur nombre.

Solution

Q1) Fonction NBExiste(C, N : Entier) : Entier
Var Cpt : Entier
Début

```
    Cpt ← 0
    Répéter
        Si C = N Mod 10 Alors
            Cpt ← Cpt + 1
        FSi
        N ← N Div 10
    Jusqu'à N = 0
```

```
        NBExiste ← Cpt
Fin
```

Q2) Fonction estDistinct(N : Entier) : Booléen

```
Var Rep : Booléen
Début
        Rep ← Vrai
        TQ (N ≠ 0) ET (Rep = Vrai) Faire      // OU TQ (N ≠ 0) ET Rep Faire
            Si NBExiste(N Mod 10, N Div 10) ≠ 0 Alors
                Rep ← Faux
            Sinon
                N ← N Div 10
            FSi
        FTQ
        estDistinct ← Rep
Fin
```

Q3) Procédure Affichage1( )

```
Var N, Cpt : Entier
Début
        Cpt ← 0
        Pour N ← 100 à 999 Faire
            Si estDistinct( N ) Alors
                Ecrire( N )
                Cpt ← Cpt + 1
            FSi
        FPour
        Ecrire( Cpt )
Fin
```

Q4) <u>Solution 1</u> : La version itérative

```
Fonction estBien (N : Entier) : Booléen
Var Rep : Booléen
        C1 : Entier
Début
        Rep ← Vrai
        TQ (N > 9) ET (Rep = Vrai) Faire      // OU TQ (N > 9) ET Rep Faire
            C1 ← N Mod 10
            N ← N Div 10
```

```
        Si C1 ≤ (N Mod 10) Alors
            Rep ← Faux
        FSi
    FTQ
    estBien ← Rep
Fin
```

Q4) <u>Solution 2</u> : La version récursive

```
Fonction estBien(N : Entier) : Booléen
Var C1 : Entier
Début
    Si N < 10 Alors
        estBien ← Vrai
    Sinon
        C1 ← N Mod 10
        N ← N Div 10
        Si C1 ≤ (N Mod 10) Alors
            estBien ← Faux
        Sinon
            estBien ← estBien( N )
        FSi
    FSi
Fin
```

Q5) Procédure Affichage2( )

```
Var N, Cpt : Entier
Début
    Cpt ← 0
    Pour N ← 1 à 123456789 Faire
    // Ou Pour N ← 10 à 123456789 Faire
    // Une séquence doit contenir au moins deux éléments !
        Si estBien( N ) Alors
            Ecrire( N )
            Cpt ← Cpt + 1
        FSi
    FPour
    Ecrire( "Il existe ", Cpt, " nombres bien ordonnés")
Fin
```

Q6) Algorithme NombresDistincts

```
// Rappel de toutes les fonctions et procédures
Fonction NBExiste(C, N : Entier) : Entier …
Fonction estDistinct(N : Entier) : Booléen …
```

```
Procédure Affichage1( ) …
Fonction estBien(N : Entier) : Booléen …
Procédure Affichage2( ) …
Début
    Affichage1( )
    Affichage2( )
Fin
```

Exercice 7 : Nombres Automorphes

(Adapté à partir de l'examen de rattrapage du S2 du 14/09/2019)

Q1) Pour un entier positif **N**, écrire une **fonction *Carre(N)*** qui calcule le carré de **N**.

Un entiers **N** strictement positif (**N > 0**) est dit automorphe si son carré ($\mathbf{N^2}$) se termine par le nombre **N** lui-même.

Exemples : Les nombres **25, 76 et 890625** sont des nombres automorphes parce que :

- $\mathbf{25^2}$ = 6**25** (N^2 se termine par N = 25)
- $\mathbf{76^2}$ = 57**76** (N^2 se termine par N = 76)
- $\mathbf{890625^2}$ = 793212**890625** (N^2 se termine par N = 890625)

Q2) Écrire une **fonction *Droite(N1, N2)*** (itérative ou récursive) qui vérifie si un entier **N1** est la partie droite de l'entier **N2**. On suppose que **N2** contient au moins le même nombre de chiffres que **N1**.

Pour le faire, on procède comme suit : On extrait les chiffres de **N1** et de **N2**, un par un, à partir de la droite et on vérifie que chaque deux chiffres extraits, à la même étape, sont identiques jusqu'à ce que tous les chiffres de **N1** soient testés.

Exemples : Si N1= 25 et N2= 625, l'appel de fonction ***Droite(25, 625)*** retourne **VRAI**.
Si N1= 15 et N2= 625, l'appel de fonction ***Droite(15, 625)*** retourne **FAUX**.

Q3) Écrire une **fonction *Automorf(N)*** qui vérifie si l'entier **N** est automorphe ou pas.

Q4) Écrire une **procédure *Affichage(N, Cpt)*** qui permet de compter et d'afficher tous les nombres automorphes qui ont **N** Chiffres.

Exemple : Si **N = 3** , la procédure affiche **376 et 625.** Donc, il existe **2** nombres automorphes de **3** chiffres.

Q5) Écrire ***un algorithme principal*** qui lit un entier strictement positif **N** et affiche tous les nombres automorphes qui ont **au plus N** chiffres ainsi que leur nombre.

Exemple : Si **N = 4** , l'algorithme affiche **1, 5, 6, 25, 76, 376, 625 et 9376.** Donc, il existe **8** nombres automorphes qui ont au plus **4** Chiffres.

Solution

Q1) Fonction ***Carre***(N : Entier) : Entier

```
Début
    Carre ← N * N
Fin
```

Q2) Solution 1 : La version itérative

```
Fonction Droite(N1, N2 : Entier) : Booléen
Var Rep : Booléen
Début
    Rep ← Vrai
    Répéter
        Si N1 Mod 10 ≠ N2 Mod 10 Alors
            Rep ← Faux
        Sinon
            N1 ← N1 Div 10
            N2 ← N2 Div 10
        FSi
    Jusqu'à (N1 = 0 ) OU ( Rep = Faux)
    Droite ← Rep
Fin
```

Q2) Solution 2 : La version récursive

```
Fonction Droite(N1, N2 : Entier) : Booléen
Début
    Si N1 < 10 Alors
        Droite ← N1 = N2 Mod 10
    Sinon
        Si N1 Mod 10 ≠ N2 Mod 10 Alors
            Droite ← Faux
        Sinon
            Droite ← Droite(N1 Div 10, N2 Div 10)
        FSi
    FSi
Fin
```

Q3) Fonction Automorf(N : Entier) : Booléen

```
Début
    Si Droite(N, Carre( N ) ) Alors          // ≡ Si Droite(N, N * N ) Alors
        Automorf ← Vrai
    Sinon
        Automorf ← Faux
    FSi
    // ≡ Automorf ← Droite(N, N * N) // N * N ≡ Puiss(N, 2)
Fin
```

Q4) Procédure Affichage(N : Entier ; **Var** Cpt : Entier)

```
Var NMin, NMax, I : Entier
Début
    NMin ← 1
    Pour I ← 1 à N – 1 Faire       // Calcul de 10 ^ (N – 1)
        NMin ← NMin * 10
    Fpour
    NMax ← NMin * 10 – 1           // Calcul de (10 ^ N ) – 1
    Cpt ← 0
    Pour I ← NMin à NMax Faire
        Si Automorf( I ) Alors
            Ecrire( I )
            Cpt ← Cpt +1
        FSi
    FPour
Fin
```

Q4) Algorithme NombresAutomorphes

```
Var     N, NBChiffres, Cpt, Cpt2 : Entier      // Déclaration des variables globales
// Rappel des fonctions & procédures
Fonction Carre(N : Entier) : Entier …
Fonction Droite(N1, N2 : Entier) : Booléen …
Fonction Automorf(N : Entier) : Booléen …
Procédure Affichage( N : Entier ; Var Cpt : Entier ) …
Début
    Répéter
        Ecrire("Donner le nombre de chiffres maximal à tester : ")
        Lire( N )
    Jusqu'à N > 0
    Cpt ← 0
    Pour NBChiffres ← 1 à N Faire
        Affichage( NBChiffres, Cpt2 )
        Cpt ← Cpt + Cpt2
    FPour
    Ecrire("Il existe ", Cpt, " nombres automorphes qui ont au plus ", N, " chiffres")
Fin
```

Exercice 8 : Nombres Colombiens (Auto-nombres)

(Adapté à partir de l'interrogation du S2 du 12/05/2018)

Un entiers **N** strictement positif (**N > 0**) est dit colombien (ou auto-nombre) s'il ne peut pas être décomposé en : la somme d'un autre entier et des chiffres de ce dernier.

Exemples : - Si **N = 15, 15** n'est pas colombien parce que **15 = 12 + (1+2).**

- Si **N = 20, 20** est colombien parce qu'il n'existe pas d'entier tel que la somme de cet entier et de ses chiffres soit égale à 20.

Q1) Écrire une **fonction récursive *SommeChiffres(N)*** qui calcule la somme des chiffres d'un entier positif **N**. **Exemple** : Si **N = 12** alors ***SommeChiffres*** retourne **1 + 2 = 3**.

Q2) Écrire une **fonction itérative *Colombien(N)*** qui permet de vérifier si un entier strictement positif **N** est colombien ou pas.

Q3) Écrire une **procédure itérative *Affichage(N)*** qui permet de trouver et afficher les N premiers nombres colombiens. **Exemple** : Si **N = 10** alors les 10 premiers nombres colombiens sont : **1, 3, 5, 7, 9, 20, 31, 42, 53, 64.**

Q4) Écrire un algorithme principal qui permet de lire un entier positif **N** puis affiche les **N** premiers nombres colombiens.

Solution

Q1) Fonction ***Carre***(N : Entier) : Entier

```
Fonction SommeChiffres(N : Entier) : Entier
Début
    Si N = 0 Alors
        SommeChiffres ← 0
    Sinon
        SommeChiffres ← N Mod 10 + SommeChiffres(N Div 10)
    FSi
Fin
```

```
// Solution 2
Si N <10 Alors
    SommeChiffres ← N
Sinon
    SommeChiffres ← N Mod 10 + SommeChiffres(N Div 10)
FSi
Fin
```

Q2) Fonction Colombien(N : Entier) : Booléen

```
Var     Nombre : Entier
        B : Booléen
Début
    B ← Vrai
    Nombre ← 1
    TQ (Nombre < N) et (B = Vrai) Faire
        Si Nombre + SommeChiffres(Nombre) = N Alors
            B ← Faux
        Sinon
            Nombre ← Nombre +1
        FSi
    FTQ
```

```
    Colombien ← B
Fin
```

Q3) Procédure Affichage(N : Entier)

```
Var     Nombre, Cpt : Entier
Début
    Ecrire("Les ", N, " premiers nombres colombiens sont : ")
    Nombre ← 1
    Cpt ← 0
    TQ Cpt < N Faire
        Si Colombien(Nombre) Alors
            Ecrire(Nombre)
            Cpt ← Cpt +1
        FSi
        Nombre ← Nombre + 1
    FTQ
Fin
```

Q4) Algorithme NombresColombiens

```
// Déclaration des variables globale
Var N : Entier
// Déclaration des fonctions et procédures
Fonction SommeChiffres(N : Entier) : Entier
Début …Fin
Fonction Colombien(N : Entier) : Booléen
Début …Fin
Procédure Affichage(N :Entier)
Début …Fin
Début // Corps de l'algorithme principal
    Répéter
        Ecrire("Combien de nombres colombiens souhaitez-vous afficher ?")
        Lire(N)
    Jusqu'à N > 0 (* à la limite N ≥ 0 *)
    Affichage(N)
Fin // Fin du corps de l'algorithme principal
```

Exercice 9 : Codages et Conversions

(Adapté à partir de l'examen du S2 du 03/06/2018)

Q1) Pour convertir un entier positif **N** codé en décimal vers le binaire, il faut faire des divisions successives par deux (2) jusqu'à ce que le résultat de la division devienne nul

(zéro). La représentation binaire de cet entier est la concaténation des restes des divisions successives.

Exemple : Si **N = 13** Alors : **13** / 2 = **6** et le reste est **1**

6 / 2 = **3** et le reste est **0**

3 / 2 = **1** et le reste est **1**

1 / 2 = **0** et le reste est **1**

→ **13 (en décimal) = 1101 (en binaire)**

Écrire une **fonction *Binaire(N)*** qui convertit un entier **N** (écrit en décimal) vers le binaire.

Q2) En généralisant l'algorithme précédent, écrire une **fonction *Conversion(N, B)*** qui permet de convertir un entier **N** codé en décimal vers toute base **B** comprise entre **2** et **9** (c'est-à-dire **2 ≤ B ≤ 9**).

Q3) Écrire une **procédure *Affichage(N)*** qui permet de lire un entier positif **N** et de l'afficher dans toutes les bases comprises entre **2** et **9**.

Exemple : Si **N = 13**, la procédure affiche les messages suivants :

13 en base 2 = **1101**	**13** en base 3 = **111**	**13** en base 4 = **31**
13 en base 5 = **23**	**13** en base 6 = **21**	**13** en base 7 = **16**
13 en base 8 = **15**	**13** en base 9 = **14**	

Q4) Pour convertir un entier codé en une base **B** comprises entre **2** et **9** (**2 ≤ B ≤ 9**) vers le décimal, on calcule la somme des chiffres de ce nombre multiplié chacun par la base **B** élevée à la puissance correspondante à la position du chiffre (positions numérotées à partir de la droite).

Exemples : Soit **NB** le nombre entier écrit en base **B** et **N** le même nombre écrit en décimal.

Si B= 2 et NB= 1101 → N= $1*2^3 + 1*2^2 + 0*2^1 + 1*2^0 = 8 + 4 + 0 + 1 =$ **13** (en décimal)

Si B= 8 et NB= 15 → N = $1*8^1 + 5*8^0 = 8 + 5 =$ **13** (en décimal)

Si B= 9 et NB= 5 → N = $5*9^0 =$ **5** (en décimal)

Écrire une **fonction récursive *Decimal(NB, B)*** qui permet de convertir un entier **NB** codé en une base **B** vers le décimal.

Solution

Q1) Fonction **Binaire**(N :Entier) : Entier

```
Var     R, P : Entier
Début
    R ← 0
    P ← 1
```

```
        TQ N ≠ 0 Faire
            R ← R + (N Mod 2)* P
            N ← N Div 2
            P ← P * 10
        FTQ
        Binaire ← R
Fin
```

Q2) Fonction **Conversion**(N, B :Entier) : Entier

```
Var     R, P : Entier
Début
        R ← 0
        P ← 1
        TQ N ≠ 0 Faire
            R ← R + (N Mod B)* P
            N ← N Div B
            P ← P * 10
        FTQ
        Conversion ← R
Fin
```

Q3) Procédure **Affichage**(Var N : Entier)

```
Var     Base : Entier
Début
        Répéter
            Lire( N )
        Jusqu'à N ≥ 0
        Pour Base ← 2 à 9 Faire
            Ecrire( N, " en base ", Base, " = ", Conversion( N, Base) )
        FPour
Fin
```

Q4) Fonction **Decimal**(NB, B :Entier) : Entier

```
Début
        Si NB < B Alors
            Decimal ← NB
        Sinon
            Decimal ← NB Mod 10 + B * Decimal( NB Div 10, B)
        FSi
Fin
```

Exercice 10 : Fécondité d'un nombre

(Adapté à partir de l'examen de rattrapage du S2 du 20/06/2018)

A partir d'un entier **N** strictement positif (**N>0**), on construit une suite dans laquelle chaque nouveau terme est égal à la somme du terme précédent et le produit de ses chiffres. Cette suite s'arrête quand le chiffre zéro (0) apparait dans un terme. On appelle **fécondité de N** le nombre de termes de cette suite.

Exemples : Si **N= 23**, la suite est : **23, 23+(2*3)= 29, 47, 75, 110 ➔ fécondité de 23 = 5.**

Si **N= 405**, la suite est : **405 (Un seul terme. 405 contient un zéro)➔ fécondité de 405 = 1.**

Q1) Écrire une fonction ***ZeroExiste(N)*** qui permet de vérifier si un entier positif **N** contient au moins un zéro.

Q2) Écrire une fonction **récursive *ProduitChiffres(N)*** qui calcule le produit (multiplication) des chiffres d'un entier positif **N**.

Exemple : Si **N = 723**, la fonction ***ProduitChiffres*** retourne **7 * 2 * 3 = 42**.

Q3) Écrire une fonction ***Fecondite(N)*** qui calcule la fécondité de **N**.

Q4) Écrire une procédure ***Rechercher(N, FMax, iMin, iMax)*** qui permet de rechercher le nombre **N** qui a la fécondité maximale (FMax) dans l'intervalle [iMin .. iMax].

Exemple : Dans l'intervalle [100 .. 200], le nombre N = 187 a la fécondité maximale = 28.

Solution

```
Fonction ZeroExiste(N : Entier) : Booléen
Var     B : Booléen
Début
    |   Répéter
    |   |       B ← N Mod 10 = 0
    |   |       N ← N Div 10
    |   Jusqu'à (B = Vrai) Ou (N = 0)
    |   ZeroExiste ← B
Fin
```

Q2)

```
Fonction ProduitChiffres(N : Entier) : Entier
Début
    |   Si N ≤ 9 Alors // Cas des nombres de 0 à 9
    |   |       ProduitChiffre ← N
    |   Sinon
    |   |       ProduitChiffre ← (N Mod 10) * ProduitChiffre(N Div 10)
    |   FSi
Fin
```

Q3) Fonction Fecondite(N : Entier) : Entier

```
Var     Cpt : Entier
Début
        Cpt ← 1; //N initial est compté aussi
        TQ ZeroExiste(N) = Faux Faire
                N ← N + ProduitChiffres(N)
                Cpt ← Cpt+1
        FTQ
        Fecondite ← Cpt
Fin
```

Q4) Procédure Rechercher(Var N, FMax : Entier ; iMin, iMax : Entier)

```
Var     Nombre, F : Entier
Début
        FMax ← 0
        Pour Nombre ← iMin à iMax Faire
                F ← Fecondite(Nombre)
                Si F > FMax Alors
                        FMax ← F
                        N ← Nombre
                FSi
        FPour
Fin
```

Exercice 11 : Nombres Frères

(Adapté à partir de l'interrogation du S2 du 22/04/2017)

Deux entiers **N1** et **N2** strictement positifs (**N1 > 0**, **N2 > 0** et **N1 ≠ N1**) sont dits frères si chaque chiffre de **N1** apparaît au moins une fois dans **N2** et inversement.

Exemple

- Si **N1 = 1164** et N2 = **641** → N1 et N2 sont **frères** car les chiffres 1, 6, 4 apparaissent dans les deux nombres 1164 et 641.
- Si **N1 = 905** et **N2 = 5909** → N1 et N2 sont **frères** car les chiffres 9, 0, 5 apparaissent dans les deux nombres 905 et 5909.
- Si N1 = **405** et N2 = **554** → N1 et N2 ne sont **pas frères** car les chiffres 4 et 5 apparaissent dans les deux nombres 405 et 554 **mais le chiffre 0 apparait dans 405 mais pas dans 554**.

Q1) Ecrire une **fonction itérative *Existe(C, N)*** qui vérifie si un chiffre **C** apparait dans un entier **N** ou non.

Q2) Ecrire une **fonction itérative *Tous(N1, N2)*** qui vérifie si tous les chiffres d'un entier **N1** apparaissent dans un autre entier **N2**.

Q3) Ecrire une **procédure *NombresFreres()*** qui lit deux entiers strictement positifs **N1** et **N2** puis affiche un message indiquant s'ils sont frères ou non.

Q4) Ecrire une **procédure *Affichage()*** qui permet de trouver tous les nombres frères de quatre (4) ou cinq (5) chiffres.

Q4) Un entier **N** est dit **distinct** s'il est composé de chiffres tous différents.

Ecrire une fonction ***distinct(N)*** (itérative ou récursive) qui permet de vérifier si un entier **N** est distinct ou non.

Q5) Ecrire une **fonction récursive** pour la question **Q1**.

Q6) Ecrire une **fonction récursive** pour la question **Q2**.

Solution

Q1) Fonction Existe(C, N : Entier) : Booléen

```
Var     Ret : Booleén
Début
        Ret = Faux
        Répéter
                Si C = N MOD 10 Alors
                        Ret ← Vrai
                Sinon
                        N ← N DIV 10
                FSi
        Jusqu'à N = 0 OU Ret = Vrai
        Existe ← Ret
Fin
```

Solution 2

```
TantQue (N ≠ 0) et (Ret = Faux) Faire
        …
FTQ
```

Q2) Fonction Tous(N1, N2 :Entier) : Booléen

```
Var     Ret : Booléen
Début
        Ret ← Vrai
        Répéter
                Si NON Existe(N1 MOD 10,N2) Alors
                        Ret ← Faux
                Sinon
                        N1 ← N1 DIV 10
                FSi
        Jusqu'à N1 = 0 OU Ret = Faux
        Tous ← Ret
Fin
```

Solution 2

```
TantQue (N ≠ 0) et (Ret = Vrai) Faire
        Si Existe(N1 MOD 10,N2) = Faux Alors
        …
FTQ
```

Q3) Procédure NombresFreres()

```
Var     N1, N2 : Entier
Début
        Répéter
            Ecrire("Donner deux entiers strictement positifs N1 et N2")
            Lire(N1, N2)
        Jusqu'à (N1 > 0) et (N2 > 0) et (N1 ≠ N2)
        Si Tous(N1, N2) et Tous(N2, N1) Alors
            Ecrire(N1, " et ", N2, " sont des nombres frères")
        Sinon
            Ecrire(N1, " et ", N2, " ne sont des nombres frères")
        FSi
Fin
```

Q4) <u>Solution 1 :</u>

```
Procédure Affichage()
Var N1, N2 : Entier
Début
        Ecrire("Liste des nombres frères de 4 ou 5 chiffres :")
        Pour N1 = 1000 à 99999 Faire
            Pour N2 = N1 + 1 à 99999 Faire // Il n'y aura pas de doublons !
                Si Tous(N1, N2) et Tous(N2, N1) alors
                    Ecrire(N1, N2)
                FSi
            Fpour
        FPour
Fin
```

<u>Solution 2 :</u>

```
Procédure Affichage()
Var     N1, N2 : Entier
Début
        Ecrire("Liste des nombres frères de 4 ou 5 chiffres :")
        Pour N1 = 1000 à 99999 Faire
            Pour N2 = 1000 à 99999 Faire
                Si Tous(N1, N2) et Tous(N2, N1) et (N1 < N2) alors
                    // N1 < N2 Pour éviter un affichage double (N1, N2) puis (N2, N1)
                    Ecrire(N1, N2)
                FSi
            Fpour
        FPour
Fin
```

Q5) Une **fonction récursive** pour la question **Q1**.

Solution 1 :

```
Fonction Existe(C, N : Entier) : Booléen
Début
    Si N < 10 Alors
        Existe ← C = N
    Sinon
        Si C = N MOD 10 alors
            Existe ← Vrai
        Sinon
            Existe ← Existe(C, N DIV 10)
        FSi
    FSi
Fin
```

```
Si C = N Alors
    Existe ← Vrai
Sinon
    Existe ← Faux
FSi
```

Solution 2 :

```
Fonction Existe(C, N : Entier) : Booléen
Début
    Si N = 0 alors // N > 0, Si N=0 alors il est le résultat des divisions successives par 10
        Existe ← Faux
    Sinon
        Si C = N MOD 10 alors
            Existe ← Vrai
        Sinon
            Existe ← Existe(C, N DIV 10)
        FSi
    FSi
Fin
```

Q6) Une **fonction récursive** pour la question **Q2**.

Solution 1 :

```
Fonction Tous(N1,N2 : Entier) : Booléen
Début
    Si N1 < 10 Alors        // Un seul chiffre
        Tous← Existe(N1, N2)
    Sinon
        Si NON Existe(N1 MOD 10, N2) Alors
            Tous← Faux
```

```
        Sinon
            Tous← Tous(N1 DIV 10, N2)
        FSi
    FSi
Fin
```

Solution 2 :

```
Fonction Tous(N1, N2 :Entier) :Booléen
Début
    Si N1 = 0 alors // Au départ N1>0, si N1=0 ≡ Tous les chiffres de N1 existent dans N2
        Tous ← Vrai
    Sinon
        Si NON Existe(N1 MOD 10, N2) alors
            Tous ← Faux
        Sinon
            Tous ← Tous(N1 DIV 10, N2)
        FSi
    FSi
Fin
```

Exercice 12 : Nombres Premiers et Résistants

(Adapté à partir de l'examen du S2 du 25/05/2017)

Un nombre entier **N (N > 1)** est premier si :

①- **N** n'est pas pair sauf s'il est égal à deux (2),

②- **N** n'admet aucun diviseur impair compris entre trois (3) et sa racine carrée.

Q1) En respectant cette définition, écrire une fonction ***Premier(N)*** qui permet de vérifier si un entier **N** est premier ou pas.

Un nombre premier est dit **résistant à droite** s'il reste premier en lui supprimant ses chiffres un par un à partir de la droite. **Exemple** : **7193** est résistant à droite car tous les nombres **7193**, **719** (3 est supprimé), **71** (9 est supprimé), **7** (1 est supprimé) sont premiers.

Q2) Ecrire une **fonction récursive *ResistDroite*** qui vérifie si un nombre premier **N** est résistant à droite.

Un nombre premier est dit **résistant à gauche** s'il reste premier en lui supprimant ses chiffres un par un à partir de la gauche. **Exemple** : **9137** est résistant à gauche car tous les nombres **9137**, **137** (9 est supprimé), **37** (1 est supprimé), **7** (3 est supprimé) sont premiers.

Q3) Ecrire une **fonction itérative *ResistGauche*** qui vérifie si un nombre premier **N** est résistant à gauche.

Un nombre premier est dit **résistant** s'il est à la fois résistant à droite et résistant à gauche.

Exemple : **3137** est un nombre premier résistant.

Q4) Ecrire une **procédure *Affichage(Max)*** qui permet de rechercher tous les nombres premiers inférieurs à un entier **Max** et pour chaque nombre premier trouvé lui associer le message le plus adapté parmi les messages suivants :

①- "N est résistant." | ②- "N est résistant à droite."

③- "N est résistant à gauche." | ④- "N est premier."

Exemples :

Si N = ?	**Message à afficher**	**Explication**
N = 10		N n'est pas premier (aucun message n'est affiché).
N = 41	41 est premier	N est premier seulement.
N = 7193	7193 est résistant à droite	N est premier et résistant à droite, mais n'est pas résistant à gauche.
N = 9137	9137 est résistant à gauche	N est premier et résistant à gauche, mais n'est pas résistant à droite.
N = 3137	3137 est résistant	N est premier, résistant à droite et résistant à gauche.

Q5) Ecrire une **fonction itérative** pour la question **Q2**.

Solution

Q1) Fonction Premier(N : Entier) : Booléen

```
Var     P : Booléen
        D : Entier
Début
    Si N ≤ 1 Alors // Par convention le 1 n'est pas premier
        P ← Faux
    Sinon
        Si N MOD 2 = 0 Alors
            P ← N = 2      // Le deux (2) est le seul nombre premier pair
        Sinon
            D ← 3 // On ne vérifie que les nombres impairs à partir de trois (3)
            P ← Vrai
            TQ (D*D ≤ N) ET (P= Vrai) Faire // Ou TQ (D ≤ SQRT(N)) ET P Faire
                Si N MOD D = 0 Alors
                    P ← Faux // D est un diviseur de N.
                Sinon
                    D ← D+2 // Passer au nombre impair suivant.
                FSi
            FTQ
        FSi
    FSi
    Premier ← P
Fin
```

```
Si N = 2 Alors
    P ← Vrai
Sinon
    P ← Faux
FSi
```

Q2) Fonction ResistDroite(N : Entier) : Booléen

```
Début
    Si N < 10 Alors // Si N est composé d'un seul chiffre
        ResistDroite ← Premier(N)
    Sinon
        Si Premier(N) Alors
            // Supprimer un chiffre à droite et continuer
            ResistDroite ← ResistDroite(N DIV 10)
        Sinon
            ResistDroite ← Faux
        FSi
    FSi
Fin
```

Q3) <u>Solution 1 :</u>

```
Fonction ResistGauche(N : Entier) : Booléen
Var     P, N2 : Entier
        RG : Booléen
Début
    P ← 10 // P ≡ Puissances de 10 : 10, 100, …
    Répéter
        N2 ← N MOD P // Extraire la partie droite du nombre N.
        RG ← Premier(N2)
        P ← P * 10
    Jusqu'à N2 = N ou NON RG
    ResistGauche ← RG
Fin
```

<u>Solution 2 :</u>

```
Fonction ResistGauche(N : Entier) : Booléen
Var P, N2 : Entier
    RG : Booléen
Début
    RG ← Vrai
    P ← 10 // P ≡ Puissances de 10 : 10, 100, …
    N2 ← N MOD P  // Extraire la partie droite du nombre N.
    TantQue (N2 ≠ N) et RG Faire
        Si Premier(N2) Alors
            P ← P * 10
```

```
            N2←N MOD P // Nouvelle partie droite de N
        Sinon
            RG←Faux
        FSi
    FTQ
    ResistGauche ← Premier(N2) et RG
Fin
```

Q4) Procedure Affichage(Max : Entier)

```
Var     N : Entier
        RD, RG : Booléen
Début
    Pour N ← 2 à (Max – 1) Faire
        Si Premier(N) Alors
            RD ← ResistDroite(N)
            RG ← ResistGauche(N)
            Si RD et RG alors
                Ecrire(N," est résistant")
            Sinon
                Si RD alors
                    Ecrire(N," est résistant à droite")
                Sinon
                    Si RG alors
                        Ecrire(N," est résistant à gauche")
                    Sinon
                        Ecrire(N," est premier")
                    FSi
                FSi
            FSi
        FSi
    FPour
Fin
```

Q5) Fonction ResistDroite(N : Entier) : Booléen

```
Var     RD : Booléen
Début
    Répéter
        RD ← Premier(N)
        N ← N Div 10 // Supprimer un chiffre à droite.
    Jusqu'à (N = 0) ou NON RD
    ResistDroite ← RD
Fin
```

Exercice 13 : Chiffre de chance

(Adapté à partir de l'examen de rattrapage du S2 du 22/06/2017)

Pour trouver le **chiffre de chance** d'une personne, on calcule la somme des chiffres de sa date de naissance. Au nombre obtenu, on refait le même traitement jusqu'à ce qu'on trouve un nombre composé d'un seul chiffre. Ce nombre est le chiffre de chance de la personne.

Exemple : Si la date de naissance est **27/09/1999** (Lue au clavier **27091999)**.

- On calcule la somme des chiffres de la date : **2+7+0+9+1+9+9+9 = 46**
- **46** est composé de deux chiffres, on refait le même traitement : **4 + 6 = 10**.
- **10** est composé de deux chiffres, on refait le même traitement : **1 + 0 = 1**.
- **1** est composé d'un seul chiffre, c'est le chiffre de chance recherché.

Q1) Ecrire une **fonction récursive Somme** qui calcule la somme des chiffres d'un nombre entier **N**.

Q2) Ecrire une **fonction itérative Chance** qui calcule le chiffre de chance à partir d'une date.

Q3) Ecrire une **un algorithme principal** qui permet de lire la date de naissance d'une personne et affiche son chiffre de chance.

Remarque : La date de naissance est lue à partir du clavier sous la forme d'un entier (**jjmmaaaa** avec **jj = jour**, **mm = mois**, **aaaa = année**). Pour l'exemple précédent on lit la date, au clavier, comme **27091999**.

Q4) Ecrire une **fonction itérative** pour la question **Q1**.

Q5) Ecrire une **fonction récursive** pour la question **Q2**.

Solution

Q1) Fonction Somme(N : Entier) : Entier

```
Début
    Si N < 10 Alors // Si un seul chiffre
        Somme ← N
    Sinon
        Somme ← N MOD 10 + Somme(N DIV 10)
    FSi
Fin
```

Q2) Fonction Chance(N : Entier) : Entier

```
Début
    TantQue N ≥ 10 Faire
        N ← Somme( N )
    FTQ
    Chance ← N
Fin
```

Q3) Algorithme ChiffreDeChance

```
Var Date, CC : Entier // CC = Chiffre de Chance
Fonction Somme(N : Entier) : Entier … // Rappel des fonctions
Fonction Chance(N : Entier) : Entier …
Début
    Répéter
        Ecrire("Donner une date de naissance sous la forme jjmmaaaa")
        Lire( Date )
    Jusqu'à Date > 0 // Une date est un entier strictement positif.
    CC ← Chance( Date )
    Ecrire("Le chiffre de chance de ", Date, " est = ", CC)
Fin
```

Q4) Fonction Somme(N : Entier) : Entier

```
Var     S : Entier
Début
    S ← 0
    TQ N ≠ 0 Faire
        S ← S + N Mod 10
        N ← N Div 10
    FTQ
    Somme ← S
Fin
```

Q5) Fonction Chance(N : Entier) : Entier

```
Début
    Si N < 10 Alors // Si N est composé d'un seul chiffre. N est positif.
        Chance ← N
    Sinon
        Chance ← Chance( Somme( N ) ) // Changer N par la somme de chiffres de N
    FSi
Fin
```

Exercice 14 : Nombres de Kaprekar

(Adapté à partir de l'examen de rattrapage du S2 du 08/06/2016)

Un entier strictement positif **N (N > 0)** à **k** chiffres est dit nombre de **Kaprekar** si lorsqu'on élève **N** au carré, la somme du nombre composé des **k** chiffres de droite et du nombre formé par le reste des chiffres redonne le nombre d'origine.

Exemples : • 9^2 = 81 et 8 + 1 = 9 • 55^2 = 3025 et 30 + 25 = 55

• 703^2 = 494209 et 494 + 209 = 703 • 2223^2 = 4941729 et 494 + 1729 = 2223

Q1) Ecrire une **fonction récursive NBChiffres** qui permet de calculer le nombre de chiffres d'un entier strictement positif **N**.

Q2) Ecrire une **fonction itérative Kaprekar** qui permet de vérifier si un entier **N** est un nombre de Kaprekar ou non.

Q3) Ecrire une **fonction itérative** pour la question **Q1**.

Solution

Q1) Fonction NBChiffres(N : Entier) : Entier

```
Début
    Si N < 10 Alors // Nombre composé d'un seul chiffre.
        NBChiffres ← 1
    Sinon
        NBChiffres ← 1 + NBChiffres(N Div 10) // Compter un chiffre et retirer le.
    FSi
Fin
```

Q2) Fonction Kaprekar(N : Entier) : Booléen

```
Var   N1, N2, I, P : Entier
Début
    P ← 1
    Pour I ← 1 à NBChiffres( N ) Faire
        P ← P * 10
    FPour
    N1 ← (N * N) Mod P
    N2 ← (N * N) Div P
    Si N = N1 + N2 Alors
        Kaprekar ← Vrai
    Sinon                    // OU Kaprekar ← N = N1 + N2
        Kaprekar ← Faux
    FSi
Fin
```

Q3) Fonction NBChiffres(N : Entier) : Entier

```
Var     NBC : Entier
Début
        NBC ← 0
        Répéter
                NBC ← NBC + 1 // Compter le chiffre
                N ← N Div 10 // Retirer un chiffre
        Jusqu'à N = 0
        NBChiffres ← NBC
Fin
```

```
NBC ← 1 // Solution 2
TQ N > 9 Faire
        NBC ← NBC + 1
        N ← N Div 10
FTQ
```

Exercice 15 : LLC – Opérations sur les ensembles – Part1

(Adapté à partir de l'examen du S2 du 09/07/2019)

(Adapté à partir de l'examen de remplacement du S2 du 23/07/2019)

Un ensemble est une collection d'éléments qui ne peut pas contenir le même élément plus d'une fois et que l'ordre entre les éléments n'a pas d'importance.

Dans cet exercice, on considère des ensembles de nombres entiers positifs.

On souhaite implémenter quelques opérations sur les ensembles en utilisant les Listes Linéaires Chaînées.

Q1) Déclarer le type ***Liste*** permettant de manipuler des ensembles d'entiers.

Q2) Un ensemble ne peut pas contenir un élément plus d'une fois. Ecrire une fonction **Existe(L, N)** permettant de vérifier si l'entier **N** existe déjà dans l'ensemble **L** ou non.

Q3) L'ordre entre les éléments d'un ensemble n'est pas important, on peut toujours insérer les nouveaux éléments au début de la liste en vérifiant qu'ils n'existent pas déjà dans la liste. Ecrire la **procédure *InsererDebut(L, N)*** qui permet d'ajouter un nouvel élément **N** à l'ensemble **L**.

On suppose que les procédures et fonctions suivantes existent :

- ***Procédure CreerEnsemble(Var L : Liste)*** : permet de créer un ensemble d'entiers positifs ***L*** à partir du clavier.
- ***Fonction Longueur(L : Liste) : Entier*** **:** permet de calculer le nombre d'éléments d'un ensemble **L**.
- ***Procédure Afficher(L : Liste)*** : permet d'afficher les éléments d'un ensemble **L**.

Q4) Ecrire une fonction ***Identiques(L1, L2)*** qui permet de vérifier si les deux ensembles **L1** et **L2** sont identiques. C'est-à-dire **L1** et **L2** contiennent les mêmes éléments.

Q5) Écrire un ***algorithme principal*** permettant de : ①- Construire deux ensembles **E1** et **E2** contenant des entiers positifs ②- Afficher les deux ensembles construits ③- Vérifier si les deux ensembles sont identiques ou pas.

Solution

Q1) Déclaration du type Liste :

```
Type   Liste = ^Element
       Element = Enregistrement
           Val : Entier
           Suiv : Liste
       Fin
```

Q2) Fonction Existe(L : Liste ; N : Entier) : Booléen // **Solution 1 : Version itérative**

```
Var Rep : Booléen
Début
    Rep ← Faux
    TQ (L ≠ Nil) et (Rep = Faux) Faire
        Si L^.Val = N Alors
            Rep ← Vrai
        Sinon
            L ← L^.Suiv
        FSi
    FTQ
    Existe ← Rep
Fin
```

Fonction Existe(L : Liste ; N : Entier) : Booléen // **Solution 2 : Version Récursive**

```
Début
    Si L = Nil Alors
        Existe ← Faux
    Sinon
        Si L^.Val = N Alors
            Existe ← Vrai
        Sinon
            Existe ← Existe(L^.Suiv, N)
        FSi
    FSi
Fin
```

Q3) Procédure InsererDebut(Var L : Liste ; N : Entier)

```
Var P : Liste
Début
    Si NON Existe(L, N) Alors
        Allouer(P)
        P^.Val ← N
        P^.Suiv ← L
        L ← P
    FSi
Fin
```

Q4) Fonction Identiques(L1, L2 : Liste) : Booléen

```
Var Rep : Booléen
Début
    Rep ← Faux
    Si Longueur(L1) = Longueur(L2) Alors
        Rep ← Vrai
        TQ (L1 ≠ Nil) et (Rep = Vrai) Faire
            Si Non Existe(L2, L1^.Val) Alors
                Rep ← Faux
            Sinon
                L1 ← L1^.Suiv
            FSi
        FTQ
    FSi
    Identiques ← Rep
Fin
```

Q5) Algorithme LesEnsembles

```
Type   Liste = ^Element       // Déclaration du type Liste
       Element = Enregistrement
           Val : Entier
           Suiv : Liste
       Fin
Var    E1, E2 : Liste  // Déclaration des variables globales
Fonction Existe(L : Liste ; N : Entier) : Booléen …   // Rappel des procédures & fonctions
Procédure InsererDebut(Var L : Liste ; N : Entier) …
Procédure CreerEnsemble(Var L : Liste ) …
Fonction Longueur( L : Liste ) : Entier …
```

```
Procédure Afficher(L : Liste) …
Fonction Identiques(L1, L2 : Liste) : Booléen …
Début
    E1 ← Nil
    E2 ← Nil
    CreerEnsemble(E1)
    CreerEnsemble(E2)
    Afficher(E1)
    Afficher(E1)
    Si Identiques(E1, E2) Alors
        Ecrire("Les deux ensembles sont identiques")
    Sinon
        Ecrire("Les deux ensembles ne sont pas identiques")
    FSi
Fin
```

Exercice 16 : LLC – Opérations sur les ensembles – Part2

(Adapté à partir de l'examen de rattrapage du S2 du 22/06/2017)

- Un ensemble est une collection d'éléments (des nombres entiers), par exemple L = {2, 1, 7}.
- Un ensemble ne peut pas contenir le même élément plus d'une fois.
- L'ordre entre les éléments d'un ensemble n'est pas important.

Dans cet exercice, on considère des ensembles de nombres entiers positifs.

On souhaite implémenter quelques opérations sur les ensembles en utilisant les listes linéaires chaînées.

Q1) Déclarer le type **Liste** permettant de manipuler des ensembles d'entiers.

Q2) Un ensemble ne peut pas contenir un élément plus d'une fois, on a besoin d'une fonction **Existe(L, val)** permettant de vérifier si l'élément **val** existe déjà dans l'ensemble **L**.

Ecrire la fonction **Existe** de manière **récursive**.

Q3) L'ordre entre les éléments d'un ensemble n'est pas important, on peut toujours insérer les nouveaux éléments au début de la liste. Ecrire la procédure **InsererTete(L, val)** qui permet d'ajouter un nouvel élément à l'ensemble **L** en s'assurant que **val** n'appartient pas déjà à l'ensemble **L**.

Q4) Ecrire une procédure **ConstruireEnsemble(L)** qui permet de lire des valeurs entières à partir du clavier et qui construit l'ensemble **L**. La saisie au clavier s'arrête quand une valeur **négative (< 0)** est saisie. Cette valeur négative ne doit pas être ajoutée à l'ensemble **L**.

Q5) L'union de deux ensembles **L1** et **L2** (notée **L = L1 ∪ L2**) est l'ensemble **L** qui contient tous les éléments qui appartiennent à **L1** ou appartiennent à **L2**. Ecrire la procédure **Union(L1, L2, L)** qui permet de calculer l'union entre **L1** et **L2**.

Q6) L'intersection de deux ensembles **L1** et **L2** (notée **L = L1 ∩ L2**) est l'ensemble **L**, qui contient tous les éléments appartenant à la fois à **L1** et à **L2**, et seulement ceux-là. Ecrire la procédure **Intersection(L1, L2, L)** qui permet de calculer l'intersection entre **L1** et **L2**.

Q7) Ecrire un algorithme principal permettant de construire deux ensembles puis afficher leur union et leur intersection.

Solution

Q1) Déclaration du type Liste :

```
Type   Liste = ^Element
       Element = Enregistrement
           Val : Entier
           Suiv : Liste
       Fin
```

Q2) Fonction **Existe**(L :Liste, val :Entier) : Booléen

```
Début
    Si L = Nil Alors
        Existe ← Faux
    Sinon
        Si L^.Val = val Alors
            Existe ← Vrai
        Sinon
            Existe ← Existe(L^.Suiv, val)
        FSi
    FSi
Fin
```

Q3) Procédure **InsererTete**(**Var** L :Liste, val :Entier)

```
Var    P : Liste
Début
    Si NON Existe(L, val) Alors
        Allouer(P)
        P^.Val ← val
        P^.Suiv ← L
        L ← P
    FSi
Fin
```

Q4) Procédure **ConstruireEnsemble**(**Var** L :Liste)

```
Var     val : Entier
Début
    L ← Nil
    Répéter
        Ecrire("Donner un entier à ajouter à l'ensemble L")
        Lire(val)
        Si val ≥ 0 Alors
            Si NON Existe(L, val) Alors
                InsererTete(L, val)
            FSi
        FSi
    Jusqu'à val < 0
Fin
```

Q5) Procédure **Union**(L1, L2 : Liste, **Var** L :Liste)

```
Var     P : Liste
Début
    L ← Nil
    // Insérer les éléments de la 1e liste.
    P ← L1
    TantQue P ≠ Nil Faire
        InsererTete(L, P^.Val)
        P ← P^.Suiv
    FTQ
    // Ajouter les éléments de la 2e liste qui n'existent pas dans la 1e liste.
    TantQue L2 ≠ Nil faire
        Si NON Existe(L1, L2^.Val) Alors
            InsererTete(L, L2^.Val)
        FSi
        L2 ← L2^.Suiv
    FTQ
Fin
```

Q6) Procédure **Intersection**(L1, L2 : Liste, **Var** L :Liste)

```
Début
    L ← Nil
    // Insérer dans L, les éléments qui existent dans les deux listes
    TantQue L1 ≠ Nil Faire
        Si Existe(L2, L1^.Val) Alors
            InsererTete(L, L1^.Val)
        FSi
        L1 ← L1^.Suiv
    FTQ
Fin
```

Q7) Algorithme LesEnsembles

```
Type   Liste = ^Element       // Déclaration du type Liste
       Element = Enregistrement
           Val : Entier
           Suiv : Liste
       Fin
Var    E1, E2, E3, E4 : Liste // Déclaration des variables globales
Fonction Existe(L :Liste, val :Entier) : Booléen … // Rappel des procédures & fonctions
Procédure InsererTete(Var L :Liste, val :Entier) …
Procédure ConstruireEnsemble(Var L : Liste ) …
Procédure Union(L1, L2 : Liste, Var L :Liste) …
Procédure Intersection(L1, L2 : Liste, Var L :Liste)
Procédure Afficher(L : Liste) …
Début
    ConstruireEnsemble(E1)
    ConstruireEnsemble(E2)
    Ecrire("Voici l'ensemble E1 :")
    Afficher(E1)
    Ecrire("Voici l'ensemble E2 :")
    Afficher(E1)
    Union(E1, E2, E3)
    Ecrire("L'union des deux ensembles donne :")
    Afficher(E3)
    Intersection(E1, E2, E4)
    Ecrire("L'intersection des deux ensembles donne :")
    Afficher(E4)
Fin
```

Exercice 17 : LLC – Opérations sur les ensembles – Part3

(Adapté à partir de l'examen de remplacement du S2 du 23/07/2019)

Q1) Un ensemble ne peut pas contenir un élément plus d'une fois. Ecrire une procédure **InsererElement(L, N)** permettant de vérifier si l'entier **N** n'existe pas dans l'ensemble **L** avant de l'insérer à la fin de l'ensemble **L**.

Q2) On appelle **cardinal** d'un ensemble le nombre d'éléments de cet ensemble. Ecrire une fonction ***Card(L)*** permettant de compter les éléments de l'ensemble **L**.

Q3) Ecrire une fonction ***Disjoint(L1, L2)*** qui permet de vérifier si les deux ensembles **L1** et **L2** sont disjoints. C'est-à-dire **L1** et **L2** n'ont aucun élément en commun. En d'autres termes leur intersection est vide.

Solution

Q1) Procédure **InsererElement**(Var L : Liste ; N : Entier)

```
Var     Existe : Booléen
        P, Dernier : Liste
Début
        P ← L
        Dernier ← Nil
        Existe ← Faux
        TQ (P ≠ Nil) ET (Existe = Faux) Faire
                Si P^.Val = N Alors
                        Existe ← Vrai
                Sinon
                        Dernier ← P
                        P ← P^.Suiv
                FSi
        FTQ
        Si NON Existe Alors
                Allouer(P)
                P^.Val ← N
                P^.Suiv ← Nil
                Si L = Nil Alors
                        L ← P
                Sinon
                        Dernier^.Suiv ← P
                FSi
        FSi
Fin
```

Q2) Fonction Card(L : Liste) : Entier // **Solution 1 : Version itérative**

```
Var     Cpt : Entier
Début
        Cpt ← 0
        TQ L ≠ Nil Faire
                Cpt ← Cpt + 1
                L ← L^.Suiv
        FSi
        Card ← Cpt
Fin
```

Fonction Card(L : Liste) : Entier // **Solution 2 : Version récursive**

```
Début
        Si L = Nil Alors
                Card ← 0
        Sinon
                Card ← 1 + Card ← ( L^.Suiv )
        FSi
Fin
```

Q3) Fonction Disjoint(L1, L2 : Liste) : Booléen

```
Var     Dj : Booléen
        P : Liste
Début
        Dj ← Vrai
        TQ (L1 ≠ Nil) et (Dj = Vrai) Faire
                P ← L2
                TQ (P ≠ Nil) et (Dj = Vrai) Faire
                        Si P^.Val = L1^.Val Alors
                                Dj ← Faux
                        Sinon
                                P ← P^.Suiv
                        FSi
                FTQ
                L1 ← L1^.Suiv
        FTQ
        Disjoint ← Dj
Fin
```

Exercice 18 : Jeux d'éliminations

(Adapté à partir de l'examen du S2 du 03/06/2018)

Soient **N** et **K** deux entiers strictement positifs. Construire une liste dans laquelle sont enregistrés les nombres **1, 2, 3, ..., N dans cet ordre**.

En commençant à partir du premier élément, supprimer successivement tous les **K^{ièmes}** éléments de la liste, en effectuant un parcours circulaire de celle-ci. Dès qu'un élément est supprimé, son suivant est considéré comme le premier élément de la liste et le comptage recommence. Si l'élément en fin de liste est supprimé, le comptage recommence à partir du début de la liste. Ce procédé d'élimination continue jusqu'à ce que la liste devienne vide.

Exemple : Si **N = 8** et **K = 3**, la suppression des éléments contenant les **huit** (**8**) entiers se fait dans cet ordre : **3, 6, 1, 5, 2, 8, 4, 7**.

Ce procédé peut être illustré par un groupe composé à l'origine de **N** enfants formant un cercle, que l'on élimine (fait sortir) successivement en désignant le **K^{ième}** de ceux restants dans le cercle que l'on continue de parcourir.

Dans cet exercice on a besoin d'une liste linéaire chaînée simple afin de trouver l'ordre d'élimination des nombres de **1 à N**.

Q1) Déclarer le type **Liste** permettant de manipuler des listes d'entiers.

Q2) Écrire une procédure **InsererFin(L, V)** permettant d'insérer un entier **V** à la fin de la liste **L**.

Q3) Écrire une procédure **CreerListe(L, N)** permettant d'insérer les nombres **1, 2, 3, …, N dans cet ordre** dans la liste **L**.

Q4) On suppose que la procédure **SupprimerValeur(L, V)** existe et elle permet de supprimer la valeur **V** de la liste **L** ; Écrire une procédure **Eliminations(L, K, L2)** permettant d'éliminer les éléments de la liste **L** selon la méthode décrite au début de cet exercice en les insérant dans la liste **L2**.

Q5) Écrire une **procédure récursive Afficher(L)** qui permet d'afficher les éléments d'une liste **L**.

Q6) Écrire un **algorithme principal** permettant de modéliser le jeu de cet exercice en effectuant les opérations suivantes :

①- Saisir deux entiers strictement positifs **N** et **K**.

②- Construire une liste **L1** contenant les nombres **1, 2, 3, ..., N**.

③- Appliquer le procédé d'éliminations sur la liste **L1** en créant la liste **L2**.

④- Afficher la liste **L2**.

Solution

Q1) Type Liste = ^Element

Element = Enregistrement

Val : Entier

Suiv : Liste

Fin

Q2) Procédure **InsererFin** (Var L : Liste ; V : Entier)

```
Var   P, Dernier : Liste
Début
    Allouer(P)
    P^.Val ← V
    P^.Suiv ← Nil
    Si L = Nil Alors
        L ← P
    Sinon
        Dernier ← L
        TQ Dernier^.Suiv ≠ Nil Faire
            Dernier ← Dernier^.Suiv
        FTQ
        Dernier^.Suiv ← P
    FSi
Fin
```

Q3) Procédure **CreerListe** (Var L : Liste ; N : Entier)

```
Var   Nombre : Entier
Début
    L ← Nil
    Pour Nombre ← 1 à N Faire
        InsererFin( L, Nombre )
    FPour
Fin
```

Q4) On suppose que la procédure **SupprimerValeur(L, V)** existe et elle permet de supprimer la valeur **V** de la liste **L**.

```
Procédure Eliminations(Var L : Liste ; K : Entier ; Var L2 :Liste)
Var   P : Liste
      Cpt, V : Entier
Début
    L2 ← Nil
```

```
        Cpt ← 0                                          // Solution 2
        TQ L ≠ Nil Faire                                 TQ L^.suiv ≠ Nil Faire
            P ← L                                            …
            TQ P ≠ Nil Faire
                Cpt ← Cpt + 1
                Si Cpt = K Alors
                    V ← P^.Val
                    P ← P^.Suiv
                    InsererFin(L2, V)
                    SupprimerValeur(L, V)
                    Cpt ← 0
                Sinon
                    P ← P^.Suiv
                FSi
            FTQ
        FTQ                                              FTQ
                                                         InsererFin(L2, L^.Val)
                                                         SupprimerValeur(L, L^.Val)
Fin
```

Q5) Procédure Afficher(L :Liste) // **Solution 1**

```
Début
    Si L ≠ Nil Alors
        Ecrire(L^.Val)
        Afficher(L^.Suiv)
    FSi
Fin
```

Procédure Afficher(L : Liste) // **Solution 2**

```
Début
    Si L= Nil alors
        Ecrire("Liste Vide")
    Sinon
        Si L^.Suiv = Nil Alors
            // C'est le dernier élément. On l'affiche et on s'arrête.
            Ecrire(L^.Val)
        Sinon
            // Ce n'est pas le dernier élément. On l'affiche et on continue.
            Ecrire(L^.Val)
```

```
            Afficher(L^.Suiv)
        FSi
    FSi
Fin
```

Q6) Algorithme JeuxEliminations

```
Type Liste = ^Element // Déclaration du type Liste
     Element = Enregistrement
         Val : Entier
         Suiv : Liste
     Fin
Var  L1, L2 : Liste  // Déclaration des variables
     N, K : Entier
Procédure InsererFin (Var L : Liste ; V : Entier) … // Rappel des procédures
Procédure CreerListe (Var L : Liste ; N : Entier) …
Procédure SupprimerValeur(Var L : Liste ; V : Entier) …
Procédure Eliminations(Var L : Liste ; K : Entier ; L2 : Liste) …
Procédure Afficher(L :Liste) …
Début
    Repéter
        Ecrire("Donner deux entiers strictement positifs (>0)" )
        Lire(N, K)
    Jusqu'à (N > 0) ET (K > 0)
    CreerListe (L1, N)
    Eliminations(L1, K, L2)
    Afficher(L2)
Fin
```

Exercice 19 : Nombres Premiers – Crible d'Eratosthène

(Adapté à partir de l'examen de rattrapage du S2 du 14/09/2019)

Dans cet exercice, on considère des listes de nombres entiers.

Q1) Déclarer le type **Liste** permettant de manipuler des listes d'entiers.

Q2) Ecrire une procédure **InsererValeur(L, N)** permettant d'insérer l'entier **N** **au début** de la liste **L**. Il n'est pas nécessaire de vérifier si **N** existe déjà dans la liste.

Q3) Ecrire une procédure **SupprimerValeur(L, N)** permettant de supprimer l'entier **N** de la liste **L** s'il existe. **N** peut exister au début, au milieu ou à la fin de la liste.

Une méthode (le crible d'Eratosthène) pour trouver tous les nombres premiers inférieurs à un entier **NMax** procède comme suit : On crée une liste qui contient tous les nombres entiers compris entre **2** et **NMax**. On supprime de la liste tous les nombres qui sont multiples de 2,

puis à chaque fois on supprime les multiples du plus petit entier restant. On s'arrête quand le carré du plus petit entier restant devient supérieur à **NMax**.

À la fin, les nombres qui restent dans la liste sont tous des nombres premiers.

Exemple : Si **NMax = 30** on fait comme suit :

- On crée la liste **L** = { 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30 }
- On supprime les multiples de 2. **L** devient : { 2, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29 }
- Le plus petit entier restant est 3, on supprime les multiples de 3.

  L devient { 2, 3, 5, 7, 11, 13, 17, 19, 23, 25, 29 }
- Le plus petit entier restant est 5, on supprime les multiples de 5.

  L devient { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 }
- Le plus petit entier restant est 7, $7^2 = 49$ est supérieur à **NMax = 30**, on s'arrête. Les nombres qui sont dans la liste sont tous premiers **({ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 })**.

Q4) Écrire un ***algorithme principal*** permettant de :

①- Lire un entier **N** supérieur ou égal à 2,

②- Chercher les nombres premiers inférieurs à **N** avec la méthode décrite ci-dessus,

③- Afficher les nombres premiers trouvés.

Remarque importante : La seule procédure autorisée est **AfficherListe(L)**. L'utilisation des autres procédures et fonctions vues en cours ou en TD est **interdite**.

Solution

Q1) Déclaration du type Liste :

```
Type   Liste = ^Element
       Element = Enregistrement
           Val : Entier
           Suiv : Liste
       Fin
```

Q2) Procédure **InsererValeur**(**Var** L : Liste ; N : Entier)

```
Var    P : Liste
Début
    Allouer(P)
    P^.Val ← N
    P^.Suiv ← L
    L ← P
Fin
```

Q3) Procédure **SupprimerValeur**(**Var** L : Liste ; N : Entier) // **Solution 1 : Version itérative**

```
Var     P, Q : Liste
        Existe : Booléen
Début
        P ← Nil
        Q ← L
        Existe ← Faux
        TQ (Q ≠ Nil) ET (NON Existe) Faire
                Si Q^.Val = N Alors
                        Existe ← Vrai
                Sinon
                        P ← Q
                        Q ← Q^.Suiv
                FSi
        FTQ
        Si Existe Alors
                Si Q = L Alors
                        L ← L^.Suiv
                Sinon
                        P^.Suiv ← Q^.Suiv
                FSi
                Q^.Suiv ← Nil /* Isoler le nœud à supprimer. C'est optionnel */
                Libérer( Q )
        FSi
Fin
```

Procédure **SupprimerValeur**(**Var** L : Liste ; N : Entier) // **Solution 2 : Version récursive**

```
Var     P : Liste
Début
        Si L ≠ Nil Alors
                Si L^.Val = N Alors
                        P ← L
                        L ← L^.Suiv
                        P^.Suiv ← Nil // Isoler le nœud à supprimer. C'est optionnel.
                        Libérer( P )
                Sinon
                        SupprimerValeur(L^.Suiv, N)
                FSi
        FSi
Fin
```

Q4) Algorithme CribleEratosthène

```
Type   Liste = ^Element // Rappel du type Liste
       Element = Enregistrement
           Val : Entier
           Suiv : Liste
       Fin
Var    L, P1, P2 : Liste
       N, N1, N2 : Entier
       Stop : Booléen
// Rappel des procédures & fonctions
Procédure InsererValeur(Var L : Liste ; N : Entier) …
Procédure SupprimerValeur(Var L : Liste ; N : Entier) …
Procédure Afficher(L : Liste) …
Début
    Répéter
        Ecrire("Donner la limite maximale pour chercher les nombres premiers : ")
        Lire( N )
    Jusqu'à N ≥ 2
    L ← Nil
    N1 ← N
    TQ N1 ≥ 2 Faire
        InsererValeur( L, N1 ) // Création de la liste 2, 3, …, N. On commence par N.
        N1 ← N1 – 1
    FTQ
    Stop ← Faux
    P1 ← L
    TQ (P1 ≠ Nil) ET (NON Stop) Faire  // Le crible d'Eratosthène
        N1 ← P1^.Val
        Si N1 * N1 > N Alors
            Stop ← Vrai    // Tous les diviseurs ont été déjà testés
        Sinon
            P2 ← P1^.Suiv
            TQ P2 ≠ Nil Faire
                N2 ← P2^.Val
                P2 ← P2^.Suiv
                Si N2 Mod N1 = 0 Alors
                    SupprimerValeur(L, N2)
                FSi
            FTQ
        FSi
```

```
        P1 ← P1^.Suiv
    FTQ
    AfficherListe( L )
    TQ L ≠ Nil Faire        // Libérer l'espace alloué dynamiquement
        SupprimerValeur( L, L^.Val)
    FTQ
Fin
```

Exercice 20 : Décomposition en facteurs premiers.

(Adapté à partir de l'examen de rattrapage du S2 du 20/06/2018)

Pour construire la liste des facteurs premiers d'un entier **N** (**N > 1**) on procède comme suit :

1- Au départ la liste est vide (L = { }),

2- On divise **N** successivement par les entiers **P = 2, 3, 4, 5, 6, …** en testant si le nombre **P** est un diviseur de **N**. Si oui, on ajoute **P** à la fin de la liste et on remplace **N** par **N/P**.

3- On s'arrête quand le nombre **P** à tester devient supérieur à la racine carrée du nombre à décomposer.

Exemple : Pour construire la liste du nombre **N = 45** on procède comme suit :

- L = {}.
- On divise 45 par P= 2 : 2 ne divise pas 45. On passe au nombre suivant (P= 3).
- On divise 45 par P= 3 : 3 divise 45 (45 = 3 x 15) ➔ On ajoute 3 à la fin de liste et on remplace 45 par 15 (La liste devient L = {3}).
- On divise 15 par P= 3 : 3 divise 15 (15 = 3 x 5) ➔ On ajoute 3 à la fin de liste une deuxième fois et on remplace 15 par 5 (La liste devient L = {3, 3}).
- 3 est supérieur à la racine carrée de 5, on s'arrête et on ajoute 5 à la fin de la liste.

➔ Le nombre N = 45 donne la liste L = {3, 3, 5} (parce que 45 = 3 x 3 x 5).

Q1) Écrire une procédure **récursive InsererFin(L, V)** permettant d'insérer un entier **V** à la **fin** de la liste **L**. **Note** : L'utilisation de la procédure **InsérerTete(L, Val)** n'est pas autorisée.

Q2) Écrire une procédure **CreerListe(N, L)** permettant de créer la liste **L** qui contient les facteurs premiers d'un nombre entier positif **N**.

Q3) Écrire une fonction **Origine(L)** permettant de retrouver le nombre entier d'origine qui a permet de construire la liste **L**. On suppose que la liste **L** n'est pas vide.

Exemple : Si **L = {3, 3, 5}**, la fonction **Origine** retourne **45**.

Q4) Écrire une fonction **récursive Exposant(P, L)** permettant de calculer le nombre d'occurrences du facteur (nombre) **P** dans la liste **L**. **Exemple** : Si **L = {3, 3, 5}** alors l'appel **Exposant(3, L)** retourne **2**, l'appel **Exposant(5, L)** retourne **1**.

Q5) Écrire une procédure **PPCM(L1, L2, L3)** permettant de construire la liste L3 correspondant au PPCM des deux nombres représentés par les deux listes L1 et L2.

Rappel : Le PPCM (Plus Petit Commun Multiple) de deux nombres décomposés en facteurs premiers est égal au produit de tous les facteurs premiers communs ou non, chacun d'eux n'est pris qu'une seule fois, avec son exposant le plus grand.

Exemple : Si L1= {2, 2, 3, 5} (N1= 2x2x3x5 = 60) et L2= {3, 3, 5} (N2= 3x3x5 = 45), alors la procédure PPCM retourne L3= {2, 2, 3, 3, 5} (N3= 2x2x3x3x5 = 180)

Q6) Écrire une procédure ***PGCD(L1, L2, L3)*** permettant de construire la liste L3 correspondant au PGCD des deux nombres représentés par les deux listes L1 et L2.

Rappel : Le PGCD (Plus Grand Commun Diviseur) de deux nombres décomposés en facteurs premiers, est égal au produit de tous les facteurs premiers communs à ces deux nombres, chacun d'eux n'est pris qu'une seule fois, avec son exposant le plus petit.

Exemple : Si L1= {2, 2, 3, 5} (N1= 2x2x3x5 = 60) et L2= {3, 3, 5} (N2= 3x3x5 = 45), alors la procédure PGCD retourne la liste L3= {3, 5} (car le PGCD de 60 et 45 est égal 3x5 = 15).

Q7) Écrire un **algorithme principal** permettant de lire deux entiers **supérieurs à 1** et d'afficher leur PPCM et leur PGCD.

Solution

Q1) Procédure **InsererFin** (**Var** L : Liste ; V : Entier)

```
Var     P : Liste
Début
    Si L = Nil Alors
        Allouer(P)
        P^.Val ← V
        P^.Suiv ← Nil
        L ← P
    Sinon
        InsererFin(L^.Suiv, V)
    Fsi
Fin
```

Q2) Procédure **CreerListe** (N : Entier ; **Var** L : Liste)

```
Var     P : Entier
Début
    L ← Nil
    P ← 2
    TQ P*P ≤ N Faire
        Si N Mod P = 0 Alors
            InsererFin(L, P)
            N ← N Div P
        Sinon
```

```
            P ← P + 1
        Fsi
    FTQ
    InsererFin(L, N)
Fin
```

Q3) Fonction **Origine**(L : Liste) : Entier

```
Var     P : Entier
Début
    P ← 1
    TQ L ≠ Nil Faire
        P ← P * L^.Val
        L ← L^.Suiv
    FTQ
    Origine ← P
Fin
```

Q4) Fonction **Exposant**(P :Entier ; L : Liste) : Entier

```
Début
    Si L = Nil Alors
        Exposant ← 0
    Sinon
        Si L^.Val = P Alors
            Exposant ← 1 + Exposant(P, L^.Suiv)
        Sinon
            Exposant ← Exposant(P, L^.Suiv)
        Fsi
    Fsi
Fin
```

Q5) Procedure **PPCM**(L1, L2 :Liste ; **Var** L3 : Liste)

```
Var     P, Cpt1, Cpt2, Max : Entier
Début
    L3 ← Nil // Initialisation de la liste L3
    TQ L1 ≠ Nil Faire // Prendre les facteurs communs entre L1 et L2.
        P ← L1^.Val
        Cpt1 ← Exposant(P, L1)
        Cpt2 ← Exposant(P, L2)
        Si Cpt1 > Cpt2 Alors
            Max ← Cpt1
```

```
        Si non
            Max ← Cpt2
        Fsi
        TQ Max > 0 Faire
            InsererFin(L3, P)
            Max ←Max – 1
        FTQ
        TQ Cpt1 > 0 Faire
            L1 ← L1^.Suiv
            Cpt1 ←Cpt1 – 1
        FTQ
    FTQ
    TQ L2 ≠ Nil Faire // Ajouter les facteurs qui se trouvent dans L1 mais pas dans L2.
        P ← L2^.Val
        Cpt1 ← Exposant(P, L1)
        Cpt2 ← Exposant(P, L2)
        Si Cpt1 = 0 Alors // Ce facteur existe dans L2 mais pas dans L1.
            Cpt1 ← Cpt2
            TQ Cpt1 > 0 Faire
                InsererFin(L3, P)
                Cpt1 ←Cpt1 – 1
            FTQ
        Fsi
        TQ Cpt2 > 0 Faire
            L2 ← L2^.Suiv
            Cpt2 ←Cpt2 – 1
        FTQ
    FTQ
Fin
```

Q6) Procedure **PGCD**(L1, L2 :Liste ; **Var** L3 : Liste)

```
Var     P, Cpt1, Cpt2, Min : Entier
Début
    L3 ← Nil // Initialisation de la liste L3
    TQ L1 ≠ Nil Faire // Prendre les facteurs communs entre L1 et L2
        P ← L1^.Val
        Cpt1 ← Exposant(P, L1)
        Cpt2 ← Exposant(P, L2)
```

```
            Si Cpt1 < Cpt2 Alors
                Min ← Cpt1
            Si non
                Min ← Cpt2
            Fsi
            TQ Min > 0 Faire
                InsererFin(L3, P)
                Min ←Min – 1
            FTQ
            TQ Cpt1 > 0 Faire // Passage au facteur premier suivant
                L1 ← L1^.Suiv
                Cpt1 ←Cpt1 – 1
            FTQ
        FTQ
Fin
```

Q7) Algorithme FacteursPremiers

```
Type    Liste = ^Element // Déclaration du type Liste
        Element = Enregistrement
            Val : Entier
            Suiv : Liste
        Fin
Var     L1, L2, L3, L4 : Liste // Déclaration des variables globales
        N1, N2 : Entier
Procédure InsererFin(Var L : Liste ; V : Entier) …
Procedure CreerListe(N : Entier ; Var L : Liste) …
Fonction Origine(L : Liste) : Entier …
Fonction Exposant(P :Entier ; L : Liste) : Entier …
Procedure PPCM(L1, L2 :Liste ; Var L3 : Liste) …
Procedure PGCD(L1, L2 :Liste ; Var L3 : Liste) …
Début
    Repéter
        Ecrire("Donner deux entiers N1 > 1 et N2 > 1")
        Lire(N1, N2)
    Jusqu'à (N1 > 1) ET (N2 > 1)
    CreerListe (N1, L1)
    CreerListe (N2, L2)
    PPCM(L1, L2, L3)
```

```
    Ecrire("PPCM( ", N1, " , ", N2, " ) = ", Origine(L3) )
    PGCD(L1, L2, L4)
    Ecrire("PGCD( ", N1, " , ", N2, " ) = ", Nombre(L4)   )
Fin
```

Exercice 21 : Suite de Recaman

(Adapté à partir de l'examen de du S2 du 25/05/2017)

La suite de Recaman est définie comme suit :

$$\begin{cases} a_0 = 0 & Si\ n = 0 \\ Si\ n > 0 & \begin{cases} a_n = a_{n-1} - n & Si\ (a_{n-1} - n)\ est\ positif\ et\ \boldsymbol{n'existe\ pas\ dans\ la\ suite} \\ & Sinon\ \ a_n = a_{n-1} + n \end{cases} \end{cases}$$

Les 15 premiers termes de cette suite sont : 0, 1, 3, 6, 2, 7, 13, 20, 12, 21, 11, 22, 10, 23, 9.

On utilise une liste triée (ordonnée) d'entiers pour sauvegarder les termes de la suite de Recaman. C'est-à-dire, chaque nouveau terme a_n calculé est inséré dans la liste triée.

Exemple : Pour les 15 premiers termes de la suite, la liste contient dans l'ordre les éléments suivants : 0, 1, 2, 3, 6, 7, 9, 10, 11, 12, 13, 20, 21, 22, 23.

Q1) Ecrire une procédure **AjouterTerme(L, a)** qui permet d'insérer un nouveau terme ***a*** dans la liste triée ***L***. On suppose que le terme ***a*** n'existe pas dans la liste ***L***.

Note : L'utilisation des procédures ***InsererTete*** et ***InsererQueue*** n'est pas autorisée.

Q2) Ecrire une **fonction récursive TermeExiste(L, a)** qui vérifie si un terme ***a*** existe déjà dans la liste triée ***L***.

Q3) Ecrire une procédure **detruireListe(L)** permettant de supprimer tous les éléments d'une liste ***L***.

Q4) Ecrire une procédure **Recaman2(N)** qui calcule et affiche les **N** premiers termes de la suite de Recaman en utilisant une liste triée. A la fin, cette liste doit être détruite.

Q5) Ecrire un algorithme principal permettant de lire un entier **N** et d'afficher les **N** premiers termes de la suite de Recaman.

Solution

Q1) Procédure **AjouterTerme**(**Var** L : Liste, a : Entier)

```
Var     P, Prec, Q : Liste
        Stop : Booléen
Début
    ALLOUER(P) // Créer un nouvel élément.
    P^.Valeur ← a
    Si L = Nil Alors // Insertion en tête
        P^.Suiv ← Nil
        L ← P
    Sinon
```

```
        Si L^.Valeur > a Alors // Insertion en tête
            P^.Suiv ← L
            L ← P
        Sinon // Insertion au milieu ou à la fin de la liste. Chercher le bon endroit.
            Prec ← L
            Stop ← Faux
            Q ← L^.Suiv
            TantQue Q ≠ Nil et NON Stop Faire
                Si Q^.Valeur > a Alors
                    Stop ← Vrai
                Sinon
                    Prec ← Q
                    Q ← Q^.Suiv
                FSi
            FTQ
            P^.Suiv ← Q
            Prec^.Suiv ← P
        FSi
    FSi
Fin
```

Q2) Fonction **TermeExiste**(L : Liste, a : Entier) : Booléen

```
Début
    Si L = Nil Alors
        // Liste vide, ou tous les éléments ont été testés
        TermeExiste ← Faux
    Sinon
        Si L^.Valeur > a Alors
            // Le terme ne peut pas exister car la liste est triée
            TermeExiste ← Faux
        Sinon
            Si L^.Valeur = a Alors
                // Le terme est trouvé
                TermeExiste ← Vrai
            Sinon
                // Continuer la recherche …
                TermeExiste ← TermeExiste(L^.Suiv, a)
            FSi
        FSi
    FSi
Fin
```

Q3) Procédure **detruireListe**(**Var** L : Liste)

```
Var     P : Liste
Début
        TantQue L ≠ Nil Faire
                P ← L
                L ← L^.Suiv
                P^.Suiv ← Nil // Isoler l'élément pointé par P. C'est optionnel.
                Libérer(P)
        FTQ
Fin
```

Q4) Procédure **Recaman**(N : Entier)

```
Var     L : Liste
        i, prec, courant : Entier
Début
        Ecrire("Les ", N, "premiers termes de la suite de Recaman sont :")
        L ← Nil // Initialisation de la liste
        Prec ← 0 // Parce que le premier terme a0 = 0
        AjouterTerme(L, prec)
        Ecrire("a0 = ", prec) // Afficher le 1er terme a0
        Pour i ← 1 à (N – 1) Faire
                courant ← prec – i // an = an-1 – n Par definition de la suite.
                Si (courant < 0) OU TermeExiste(L, courant) Alors
                        courant ← prec + I // an = an-1 + n
                FSi
                AjouterTerme(L, courant)
                Ecrire("a", i, " = ", courant) // Afficher les termes ai : a1, a2, a3, …, aN-1
                Prec ← courant
        FPour
        detruireListe(L) // Suppression des éléments de la liste
Fin
```

Q5) Algorithme SuiteRecaman

```
Type    Liste = ^Element // Déclaration du type Liste
        Element = Enregistrement
                Val : Entier
                Suiv : Liste
        Fin
Var     N : Entier // Déclaration des variables globales
```

```
Procédure AjouterTerme(Var L : Liste, a : Entier) …
Fonction TermeExiste(L : Liste, a : Entier) : Booléen …
Procédure detruireListe(Var L : Liste) …
Procédure Recaman(N : Entier) …
Début
    Repéter
        Ecrire("Combien de termes de la suite de Recaman voulez-vous ?")
        Lire(N)
    Jusqu'à (N > 0)
    Recaman( N )
Fin
```

Exercice 22 : Représentation des polynômes

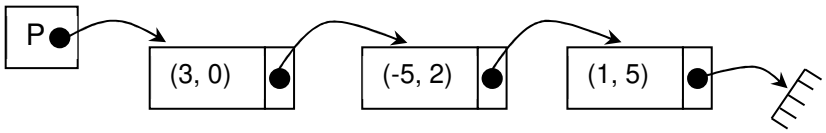
(Adapté à partir de l'examen du S2 du 17/05/2016)

Un polynôme $P(X) = a_0 + a_1X + a_2X^2 + \ldots + a_nX^n$ est une suite de monômes a_iX^i ($\mathbf{a_i}$ est le coefficient, **i** est la puissance).

Exemple : $P(X) = 3 - 5X^2 + X^5$ ($a_0 = 3$, $a_1 = 0$, $a_2 = -5$, $a_3 = 0$, $a_4 = 0$, $a_5 = 1$)

On utilise une liste linéaire chainée pour représenter un polynôme en ne prenant en compte que les monômes avec des coefficients non-nuls.

Le polynôme de l'exemple précédent peut être représenté par la liste suivante :



Q1) Déclarer le type **Liste** permettant de manipuler des polynômes.

Q2) Ecrire une procédure **AjouterMonome(P, a, n)** qui permet d'ajouter le monôme $\mathbf{ax^n}$ au début du polynôme **P**. On suppose que le coefficient $\mathbf{a \neq 0}$, la puissance $\mathbf{n \geq 0}$ et elle n'existe pas dans le polynôme **P**.

Q3) Ecrire une procédure **LirePolynome(P)** qui permet de lire des couples de valeurs (Puissance, Coefficient) à partir du clavier et qui construit le polynôme **P**.

La saisie au clavier doit respecter les règles suivantes :

1. Les puissances sont saisies de la plus grande à la plus petite (Pour l'exemple précédent on saisit les valeurs : **5** 1 **2** -5 **0** 3 -1 (n'importe quelle valeur négative, **voir règle 4**).
2. Si un coefficient est nul (**= 0**), le monôme saisi ne doit pas être ajouté au polynôme.
3. Si une puissance est déjà traitée (existe dans le polynôme ou une puissance plus petite qu'elle est déjà saisie), cette puissance est ignorée (rejetée). Il n'est pas nécessaire de lire le coefficient correspondant.

4. On s'arrête quand une puissance négative **(< 0)** est saisie. Cette valeur négative ne doit pas être ajoutée au polynôme.

Q4) Ecrire une procédure **récursive Afficher(P)** qui affiche le polynôme **P** sous forme de couples (coefficient, puissance). Pour l'exemple précédent, on aura **P= (3, 0) (-5, 2) (1, 5)**. Si **P** est nul on affiche **P= 0**.

Q5) Modifier la procédure **Afficher** pour que l'affichage soit : **P= (1, 5) (-5, 2) (3, 0)**.

Q6) Ecrire une fonction **récursive Evaluer(P, x)** qui permet de calculer la valeur du polynôme **P** pour un point **x** donné. Pour l'exemple précédent, si **x= 3.5** alors **$P(3.5) = 3 - 5*(3.5)^2 + (3.5)^5 = 3 – 5*(12.5) + 525,21875= 466,96875$**

Note : Vous pouvez utiliser la fonction ***Puiss(x, n)*** qui permet de calculer x à la puissance n.

Q7) Ecrire une procédure **Derivee(P, D)** qui permet de calculer le polynôme dérivé **D** du polynôme **P**. Pour l'exemple précédent on aura le polynôme dérivé **$D = -10X + 5X^4$**.

Rappel : la fonction dérivée d'un monôme aX^n est égale à $(n*a)X^{n-1}$

Q8) Ecrire un algorithme principal qui permet de :

①- Définir un polynôme à partir du clavier.

②- Lire une valeur réelle **x** et donner la valeur du polynôme pour ce point.

③- Calculer le polynôme dérivé.

④- Afficher le polynôme dérivé.

⑤- Afficher la valeur du polynôme dérivé pour le point **x**.

Solution

Q1) Type Liste = ^Cellule
Cellule = Enregistrement
Coefficient : Réel
Puissance : Entier
Suivant : Liste
Fin

Var P, L : Liste

Q2) Procédure **AjouterMonome**(**Var** P : Liste, a : Réel, n : Entier)

```
Var     Q : Liste
Début
        Allouer( Q )
        Q^.Coefficient ← a
        Q^.Puissance ← n
        Q^.Suivant ← P
        P ← Q
Fin
```

Q3) Procédure **LirePolynome**(**Var** P : Liste)

```
Var     Puiss, AncPuiss : Entier
        Coeff : Réel
Début
    P ← NIL
    AncPuiss ← - 1
    Répéter
        Ecrire("Donner la puissance du monôme")
        Lire(Puiss)
        Si (Puiss ≥ 0) et ((AncPuiss < 0) OU (Puiss < AncPuiss)) Alors
            Ecrire("Donner la valeur du coefficient")
            Lire(Coeff)
            Si Coeff ≠ 0 Alors
                AjouterMonome(P, Coeff, Puiss)
                AncPuiss ← Puiss
            FSi
        FSi
    Jusqu'à Puiss < 0
Fin
```

Q4) Procédure **Afficher**(P : Liste, Deb : Booléen) // Deb = est ce que c'est la 1ière fois ?

```
Début
    Si Deb = Vrai Alors
        Ecrire("P = ")
        Si P = Nil Alors
            Ecrire(" 0 ")
        FSi
    FSi
    Si P ≠ Nil Alors
        Ecrire("( ", P^.Coefficient, " , ", P^.Puissance, " )")   // Instruction I1
        Afficher(P^.Suivant, Faux)   // Instruction I2
    FSi
Fin
```

/* Dans l'algorithme principal, cette procédure doit être appelée avec le paramètre Deb = Vrai, c'est-à-dire Afficher(P, Vrai) */

Q5) Procédure Afficher2(P : Liste, Deb : Booléen). C'est la même procédure **Afficher** de la question Q4, il faut inverser les deux instructions I1 et I2.

Procédure **Afficher2**(P : Liste, Deb : Booléen)

```
Début
    Si Deb = Vrai Alors
        Ecrire("P = ")
        Si P = Nil Alors
            Ecrire(" 0 ")
        FSi
    FSi
    Si P ≠ Nil Alors
        Afficher(P^.Suivant, Faux)   // Instruction I2
        Ecrire("( ", P^.Coefficient, " , ", P^.Puissance, " )")   // Instruction I1
    FSi
Fin
```

Q6) Fonction **Evaluer**(P :Liste, x : Réel) : Réel

```
Début
    Si P = Nil Alors
        Evaluer ← 0
    Sinon
        Evaluer ← P^.Coefficient * Puiss(x, P^.Puissance) + Evaluer(P^.Suivant, x)
    FSi
Fin
```

Q7) Procédure **Derivee**(P : Liste, **Var** D : Liste)

```
Début
    D ← Nil
    TantQue P ≠ Nil Faire
        Si P^.Puissance ≠ 0 Alors
            AjouterMonome(D, P^.Coefficient * P^.Puissance, P^.Puissance – 1)
            // Remarque : L'ordre des puissances sera inversé
        FSi
        P ← P^.Suivant
    FTQ
Fin
```

Q8) Algorithme Polynomes

```
Type   Liste = ^Cellule // Rappel du type liste.
       Cellule = Enregistrement
           Coefficient : Réel
           Puissance : Entier
           Suivant : Liste
       Fin
```

```
Var     P, D : Liste // Déclaration des variables globales
        x : Réel
Fonction Puiss(x : Réel, n : Entier) : Réel   // permet de calculer x à la puissance n.
/* Liste des procédures & fonctions des questions Q2 à Q7 */
Procédure AjouterMonome(Var P : Liste, a : Réel, n : Entier)   // Q2
Procédure LirePolynome(Var P : Liste)   // Q3
Procédure Afficher(P : Liste, Deb : Booléen)   // Q4
Procédure Afficher2(P : Liste, Deb : Booléen)   // Q5
Fonction Evaluer(P :Liste, x : Réel) : Réel   // Q6
Procédure Derivee(P : Liste, Var D : Liste)   // Q7
Début
        LirePolynome( P )
        Ecrire("Fixer la valeur du point x à tester")
        Lire( x )
        Ecrire("P(x) = ", Evaluer( P, x ) )
        Deriver( P, D )
        Afficher2( D )
        Ecrire("D(x) = ", Evaluer( D, x ) )
Fin
```

Exercice 23 : File d'attente avec priorité

(Adapté à partir de l'examen de rattrapage du S2 du 08/06/2016)

Une file d'attente avec priorité est une collection d'éléments dans laquelle tout nouvel élément est inséré à la fin et l'opération de défilement (suppression) récupère l'élément le plus prioritaire indépendamment de sa position (au début, au milieu ou à la fin).

On souhaite utiliser une liste pour implémenter les files avec priorité.

On définit le type **File** comme suit :

```
Type   File = ^Cellule
       Cellule = Enregistrement
                 Val : Entier    // Champ de données
                 Prio : Entier   // Priorité de la valeur Val
                 Suiv : File     // Un pointeur vers l'élément suivant
       Fin
```

On définit un modèle sur les files avec l'ensemble des procédures et fonctions suivantes :

CreerFile(F) : Permet de créer une file vide.

Enfiler(F, Val, Prio) : Permet d'insérer à la fin de la file **F** la valeur **Val** avec la priorité **Prio**.

Defiler(F, Val, Prio) : Permet de récupérer l'élément le plus prioritaire de la file **F**. Cet élément sera supprimé de la file **F** et ses données seront sauvegardées dans les deux variables **Val** et **Prio**. Si deux éléments ont la même priorité, c'est le premier élément inséré qui sera supprimé.

FileVide(F) : Permet de tester si la file est vide.

FilePleine(F) : Permet de tester si la file est pleine.

Q1) Ecrire toutes les procédures et fonctions du modèle.

Q2) Ecrire un algorithme principal permettant de :

①- Créer une file avec priorité,

②- Insérer les quatre couples (Val, Prio) suivants : (1, 3), (2, 2), (3, 1), (4, 3),

③- Défiler tous les éléments en les affichant.

Q3) Dans quel ordre les quatre couples insérés seront affichés ?

Solution

Q1) On définit le type **File** comme suit :

```
Type   File = ^Cellule
       Cellule = Enregistrement
                 Val : Entier     // Champ de données
                 Prio : Entier    // Priorité de la valeur Val
                 Suiv : File      // Un pointeur vers l'élément suivant
       Fin
```

Q1) Ecriture de toutes les procédures et fonctions du modèle.

```
Procédure CreerFile(Var F : File)
Début
    F ← Nil
Fin
Fonction FileVide(F : File) : Booléen
Début
    FileVide ← F = Nil
Fin
Fonction FilePleine(F : File) : Booléen
Début
    FilePleine ← Faux // On suppose que la taille de la mémoire centrale est illimitée
Fin
Procédure Enfiler(Var F : File ; val, prio : Entier)
Var    P, Dernier : File
Début
    Si NON FilePleine( F ) Alors
        Allouer( P )
```

```
        P^.Val ← val
        P^.Prio ← prio
        P^.Suiv ← Nil
        Si F = Nil Alors
            F ← P
        Sinon
            // Ajouter à la fin de la file (la liste).
            Dernier ← F
            TantQue Dernier^.Suiv ≠ Nil Faire
                Dernier ← Dernier^.Suiv
            FTQ
            Dernier^.Suiv ← P
        FSi
    /* Sinon
        Ecrire("Enfiler - Erreur : File Pleine") */
    FSi
Fin
Procédure Défiler(Var F : File ; Var val, prio : Entier)
Var   P1, P2, AvantPMax, PMax : File
Début
    Si NON FileVide( F ) Alors
        /* On suppose que le 1ier élément est le plus prioritaire */
        AvantPMax ← Nil
        Pmax ← F
        /* On commence par l'élément suivant */
        P2 ← F
        P1 ← F^.Suiv
        TantQue P1 ≠ Nil Faire
            Si P1^.Prio > Pmax^.Prio Faire
                AvantPMax ← P2
                PMax ← P1
            FSi
            P2 ← P1
            P1 ← P1^.Suiv
        FTQ
        val ← PMax^.Val
        prio ← PMax^.Prio
```

```
        Si PMax = F Alors
            F ← F^.Suiv
        Sinon
            AvantPMax^.Suiv ← PMax^.Suiv
        FSi
        PMax^.Suiv ← Nil /* Optionnel */
        Libérer( PMax )
    /* Sinon
        Ecrire("Defiler - Erreur : File vide") */
    FSi
Fin
```

Q2) Algorithme FilesAvecPriorités

```
Type   File = ^Cellule // Rappel du type File
       Cellule = Enregistrement
           Val : Entier     // Champ de données
           Prio : Entier    // Priorité de la valeur Val
           Suiv : File      // Un pointeur vers l'élément suivant
       Fin
Var    F : File
       V, P : Entier
/* Liste des procédures et fonctions */
Procédure CreerFile(Var F : File) …
Fonction FileVide(F : File) : Booléen …
Fonction FilePleine(F : File) : Booléen …
Procédure Enfiler(Var F : File ; val, prio : Entier) …
Procédure Défiler(Var F : File ; Var val, prio : Entier) …
Début
    CréerFile( F )
    Enfiler( F, 1, 3)
    Enfiler( F, 2, 2)
    Enfiler( F, 3, 1)
    Enfiler( F, 4, 3)
    TantQue Non FileVide( F ) Faire
        Défiler( F, V, P)
        Ecrire("Valeur = ", V, "Priorité = ", P)
    FTQ
Fin
```

Q3) Dans quel ordre les quatre couples insérés seront affichés ?

Les éléments seront affichés selon l'ordre des priorités ainsi que l'ordre d'arrivée.

➔ Ordre : Val = 1 Priorité = 3

Val = 4 Priorité = 3

Val = 2 Priorité = 2

Val = 3 Priorité = 1

*** Fin des exercices corrigés ***

PARTIE 4– SERIES DE TD

Série de TD N1– Procédures, Fonctions & Récursivité

Université Mohammed Seddik BENYAHIA – JIJEL –
Faculté des Sciences Exactes et Informatique **2019 / 2020**
Département de Mathématiques et Informatique

– Algorithmique – TD N°4 – Procédures & Fonctions & Récursivité

Exercice 1 : Ordre croissant

Q1) Ecrire une procédure **Trier2** qui prend deux (2) entiers **A** et **B** et de les permuter, si nécessaire, pour que l'état de sortie soit **A ≤ B**. C'est-à-dire que **A** et **B** seront dans l'ordre croissant.

Q2) En s'inspirant du tri à bulles, écrire une procédure **Trier3** sur trois (3) entiers **(A, B et C)** qui appelle **Trier2** et permet d'avoir les trois variables **A, B et C** dans l'ordre croissant **(A ≤ B ≤ C)**.

Q3) Ecrire l'algorithme principal qui permet de lire 3 entiers **(X, Y et Z)** et de les afficher dans l'ordre croissant.

Exercice 2 : Carré Parfait

Q1) Ecrire une procédure **CParfait** qui permet de vérifier si un entier positif **N** est un **carré parfait**.

Cette procédure donne deux résultats : un booléen qui est égal à **vrai** si et seulement si **N** est un carré parfait et un entier correspondant à la partie entière de la racine carré de **N**.

Un entier **N** est **carré parfait** s'il est le carré d'un entier **k**, c'est-à-dire $\mathbf{N = k^2}$. Par exemple, les entiers 0, 1, 4, 9, 16 et 25 sont des carrés parfaits. Pour chercher la valeur de **k**, on calcule la somme des **k** premiers nombres impairs jusqu'à ce que cette somme soit supérieure ou égale à **N**.

- Si **N = 16** alors S = 1+3+5+7 = la somme des 4 premiers nombres impairs ; **S = N** ➔ 16 est carré parfait et $\sqrt{16}$ **= 4.**
- Si **N = 18** alors S = 1+3+5+7+9 = la somme des 5 premiers nombres impairs ; **S > N** ➔18 n'est pas carré parfait et **la partie entière de** $\sqrt{18}$ **est égale à 4.**

Q2) En utilisant la procédure **CParfait**, écrire l'algorithme principal qui permet de : ①**-** Lire un entier positif **N**, ②**-** Vérifier si **N** est un carré parfait et afficher sa racine carrée, sinon afficher la partie entière de sa racine carrée. **Note :** L'utilisation des fonctions SQR (le carré) et SQRT (la racine carrée) n'est pas autorisée.

Exercice 3 : Nombres Premiers

Un nombre entier **N (N > 1)** est premier si les deux conditions suivantes sont vérifiées :

①- **N = 6 * k - 1** ou **N = 6 * k + 1** (Où **k** est un entier strictement positif) sauf si **N = 2** ou **N = 3**,

②- **N** n'admet aucun diviseur impair compris entre trois **(3)** et sa racine carrée **(**$\sqrt{\mathbf{N}}$**)**.

Q1) En respectant cette définition, écrire une fonction **Premier(N)** qui permet de vérifier si un entier **N** est premier ou pas. Par convention, **1** n'est pas premier.

Q2) Ecrire une procédure qui permet d'afficher les **M** petits nombres premiers.

Exemple : Si M = 10 alors les 10 petits nombres premiers sont : 2, 3, 5, 7, 11, 13, 17, 19, 23, 29.

Q3) Modifier la procédure précédente pour qu'elle calcule aussi la somme de ces **M** petits nombres premiers.

Exemple : Si M = 9 alors la somme = 2 + 3 + 5 + 7 + 11+ 13 + 17 + 19 + 23 = 100.

Q4) Ecrire l'algorithme principal qui permet de lire un entier **M** et d'afficher les **M** petits nombres premiers ainsi que leur somme.

Exercice 4 : Fonctions numériques

Q1) Factorielle : Ecrire une fonction **Facto(n)** qui permet de calculer **n ! = 1 * 2 * 3 * …* n.**

Q2) Combinaisons : Ecrire une fonction **Combin(n, p)** qui permet de calculer le nombre de combinaisons défini par $C_n^p = \frac{n!}{p!*(n-p)!}$. **n** et **p** sont deux entiers positifs tels que **p ≤ n**.

Q3) Puissance : Ecrire une fonction **Puiss(x, n)** qui renvoie $\mathbf{x^n}$ **= x * x * … * x** (**x** réel et **n** entier positif).

Q4) Exponentielle : Ecrire une fonction **Expn(x, n)** qui permet de calculer la valeur approchée

$e^x \cong 1+\frac{x}{1}+\frac{x^2}{2!}+\frac{x^3}{3!}+...+\frac{x^n}{n!}$ (**x** est un nombre réel et **n** un nombre entier positif).

Exercice 5 : Récursivité

Q1) Factorielle : Ecrire une fonction **FactoRec(n)** qui permet de calculer **n ! = n * (n-1) !**

(On sait que 0 ! = 1 ! = 1)

Q2) Puissance : Ecrire deux fonctions récursives pour calculer $\mathbf{x^n}$.

- La première **PuissRec(x, n)** en utilisant la définition suivante :

$$x^n = \begin{cases} 1 & Si\ \ n=0 \\ x^{n-1}\times x & Si\ \ n>0 \end{cases}$$

- La deuxième **PuissRec2(x, n)** en utilisant la définition suivante :

$$x^n = \begin{cases} 1 & Si\ \ n=0 \\ x^{n/2}\times x^{n/2} & Si\ \ n>0\ et\ \ n\ est\ pair \\ x^{(n-1)/2}\times x^{(n-1)/2}\times x & Si\ \ n>0\ et\ \ n\ \ est\ impair \end{cases}$$

Q3) L'algorithme d'Euclide permettant de calculer le **PGCD** (Plus Grand Commun Diviseur) de deux entiers strictement positifs **A** et **B** tels que **A > B** est défini comme suit :

$$PGCD(A,B) = \begin{cases} PGCD(B, A \bmod B) & Si\ B\neq 0 \\ A & Si\ B=0 \end{cases}$$

Ecrire une fonction récursive permettant de déterminer le **PGCD** de **A** et **B**.

Q4) Suite de Fibonacci : Ecrire une fonction récursive **Fibo(n)** permettant de calculer le $\mathbf{n^{ieme}}$ terme de la suite de **Fibonacci** définie par : $U_n = \begin{cases} 0 & Si\ n=0 \\ 1 & Si\ n=1 \\ U_{n-1}+U_{n-2} & Si\ n\geq 2 \end{cases}$

Q5) Pour un entier positif N, écrire une fonction <u>itérative</u> puis une autre <u>récursive</u> qui permet :

- Calculer le nombre de chiffres de N.
- Calculer la somme des chiffres de N.
- Calculer le produit (la multiplication) des chiffres de N. **<u>Remarque</u>** : Si le nombre N contient un zéro, dès que le chiffre 0 (zéro) est rencontré, le produit devient nul (égal à zéro), il faut s'arrêter.

Université Mohammed Seddik BENYAHIA – JIJEL –
Faculté des Sciences Exactes et Informatique
Département de Mathématiques et Informatique

2019 / 2020

– Algorithmique – TD N°5 – Les Listes Linéaires Chaînées (LLC)

Dans tous les exercices on utilise des listes d'entiers.

Rappel de cours

Le type **Liste** est déclaré comme suit :

```
Type   Liste = ^Element
       Element = Enregistrement
           | Val : Entier
           | Suiv : Liste
       Fin
Var L, P : Liste /* Deux variables de type Liste */
```

Opérations sur les pointeurs

- ***Allouer(P)*** : permet de créer un nouvel élément (ou cellule) pointé par le pointeur ***P***.
- ***Liberer(P)*** : permet de supprimer l'élément (ou la cellule) pointé par le pointeur ***P.***
- ***Nil :*** est une constante affectée à un pointeur pour indiquer qu'il ne pointe vers aucune donnée. On écrit, par exemple, P ← Nil.

Exercice 1 : Construction et exploitation

Q1) Écrire une procédure ***InsererTete(L, N)*** qui permet d'insérer l'entier ***N*** au début de la liste ***L*** (La liste ***L*** peut être vide).

Q2) Utiliser la procédure ***InsererTete*** pour écrire une autre procédure ***CreerListe*** qui permet de créer une liste d'entiers positifs à partir du clavier. Si une valeur négative est saisie, l'insertion à la liste s'arrête.

Q3) Écrire une procédure ***AfficherListe(L)*** qui permet d'afficher les éléments de la liste ***L***. Si ***L*** est vide, la procédure doit afficher le message '*** Liste vide ***'.

Q4) Écrire une procédure ***InsererQueue(L, N)*** qui permet d'insérer l'entier ***N*** à la fin de la liste ***L*** (La liste ***L*** peut être vide).

Q5) Modifier la procédure écrite à la question **Q2** pour utiliser la procédure ***InsererQueue***.

Q6) Écrire une fonction ***Longueur(L)*** qui permet de calculer la longueur (nombre d'éléments) de la liste ***L***.

Q7) Réécrire les fonctions/procédures des questions **Q6, Q3, Q4** sous forme récursive.

Q8) Modifier la procédure récursive ***AfficherListe(L)*** pour qu'elle affiche les éléments de la liste ***L*** dans l'ordre inverse.

Q9) On suppose, cette fois-ci, que les éléments de la liste sont triés. Écrire une procédure ***InsererListeTriee(L, N)*** qui permet d'insérer un entier ***N*** de telle sorte que la liste reste toujours triée.

*** Q10)** Écrire une procédure ***InsererPos(L, N, pos)*** qui permet d'insérer l'entier ***N*** dans la liste ***L*** à la position ***pos.*** Si la position ***pos*** n'existe pas, l'insertion n'est pas effectuée. **Exemple** : Si la liste contient 2 éléments, il est possible d'insérer un nouvel élément à la position **1, 2** ou **3**. Cependant, Il est impossible d'insérer un nouvel élément à la position **4**.

Q11) Réécrire les deux procédures des questions **Q9, Q10** sous forme récursive.

Exercice 2 : Consultation

Q1) Écrire une fonction (itérative puis récursive) qui permet de :

a. Vérifier si une valeur ***N*** existe dans la liste. Elle retourne **Vrai**, si ***N*** existe, sinon elle retourne **Faux** *(Fonction **Existe(L, N)**)*.

b. Vérifier si une valeur ***N*** existe dans la liste. Elle retourne l'adresse de l'élément contenant la valeur ***N***. Sinon, elle retourne **Nil** *(Fonction **Adresse1(L, N)**)*.

c. Retourner un pointeur sur l'élément qui se trouve à la position ***pos***. Si la position ***pos*** n'existe pas, la fonction retourne **Nil** *(Fonction **Adresse2(L, pos)**)*

d. Calculer le nombre d'occurrence d'un entier ***N*** dans la liste.

e. Chercher le Maximum *(Fonction **Max(L)**)*. On suppose que la liste n'est pas vide.

f. Chercher le Minimum *(Fonction **Min(L)**)*. On suppose que la liste n'est pas vide.

g. Vérifier si la liste est triée.

Q2) Écrire une procédure qui recherche l'élément qui a le plus grand nombre d'occurrences dans la liste.

Exercice 3 : Mise à jour

Q1) Écrire une procédure ***SupprimerTete(L)*** qui permet de supprimer le premier élément de la liste ***L***, s'il existe.

Q2) Écrire une procédure ***SupprimerListe(L)*** qui permet de supprimer tous les éléments de la liste ***L***.

Q3) Écrire une procédure ***SupprimerValeur(L, N)*** qui permet de supprimer toutes les occurrences d'une valeur ***N***.

Q4) Écrire une procédure ***SupprimerPos(L, pos)*** qui permet de supprimer l'élément qui se trouve à la position ***pos***.

Q5) Écrire une procédure ***Fusion(L1, L2, L3)*** qui permet de fusionner deux listes triées ***L1*** et ***L2*** et créer la liste triée ***L3***.

Q6) Écrire une procédure ***Eclater(L, L1, L2)*** qui permet d'éclater la liste ***L*** en deux listes ***L1*** et ***L2***. La liste ***L1*** contient les nombres impairs et la liste ***L2*** les nombres pairs.

Q7) Écrire une procédure ***Inverser(L)*** qui permet d'inverser une liste ***L***. Le premier élément devient le dernier, le deuxième devient l'avant dernier et ainsi de suite. **Remarque** : Il est interdit de créer de nouveaux éléments.

*** Exercice 4 : Piles & Files**

Q1) Implémenter une File d'entiers en utilisant une liste linéaire chaînée.

Q2) Une file d'attente avec priorité est une file d'attente dans laquelle l'opération de défilement récupère l'élément le plus prioritaire. Définir le modèle et l'implémenter.

Q3) Implémenter une File d'entiers en utilisant un tableau.

Q4) Implémenter une Pile d'entiers en utilisant une liste linéaire chaînée.

Q5) Implémenter une Pile d'entiers en utilisant un tableau.

**** Travail facultatif***

PARTIE 5– SERIES DE TP

Série de TP N1– Procédure, Fonctions & Récursivité

Université Mohammed Seddik BENYAHIA – JIJEL –
Faculté des Sciences Exactes et Informatique **2019 / 2020**
Département de Mathématiques et Informatique

– Algorithmique – TP N°5 – PROCEDURES & FONCTIONS –

I– Introduction : Les **procédures et fonctions** sont utilisées pour **simplifier** l'écriture des programmes.

Le programme global est ainsi **décomposé** en plusieurs parties faciles à écrire, à comprendre et à faire évoluer à l'inverse d'un programme de grande taille écrit en un seul bloc.

Le programme (**principal**) décrit alors la méthode générale de résolution du problème traité et les procédures et fonctions traitent les détails.

Cette décomposition permet aussi de **réutiliser** les procédures et fonctions au lieu de les réécrire à chaque fois car souvent les mêmes traitements sont effectués plusieurs fois dans le même programme.

Résumé : ① En langage C, tous les traitements sont effectués par des fonctions. ② Les procédures sont un cas particulier de fonctions. ③ Le programme principal est la fonction ***int main( )***.

II– Les fonctions

✓ Une fonction retourne un seul résultat avec l'instruction **return**. Elle est définie selon la syntaxe :

typeResultat nomFonction(TypeParametre1 parametre1, … , TypeParametreN parametreN)

{ /* Déclarations de variables locales */

/* Liste des instructions*/

return resultat ; }

typeResultat : est l'un des types ENTIER (char, short, int, long, …) , REEL (float, double, …), CARACTERE (char) ou ENREGISTREMENT (struct).

Exemple : Une fonction qui calcule le **maximum** de deux **entiers n1** et **n2** peut être écrite comme suit :

```
int maximum(int n1, int n2) { int max ; // max est une variable locale
    if(n1 > n2) max = n1 ;
    else max = n2 ;
    return max ; }
```

✓ L'instruction **return** permet de retourner le résultat et terminer la fonction (ou la procédure). Si l'instruction **return** est rencontrée, toutes les instructions qui la suivent ne seront pas exécutées.

Exemple : La fonction **maximum** précédente peut être réécrite comme suit :

```
int maximum(int n1, int n2){ if(n1 > n2) return n1 ;
                             return n2 ; }
```

✓ Une fonction est, généralement, utilisée (appelée) dans une expression arithmétique ou logique. Elle peut aussi servir comme paramètre (passé par valeur) à une autre fonction ou procédure.

Exemples d'appels : ① int n = n3 + maximum(n1, n2) ; ② Pour trouver le maximum de trois entiers, on peut écrire : int max = maximum(maximum(n1, n2), n3) ; // Le 1er paramètre est un appel de fonction.

III – Les procédures

✓ Une procédure est une fonction dont le type de retour est **void** (vide). Il signifie que la procédure ne retourne aucune valeur. **Syntaxe de déclaration** : **void** nomProcedure(Liste_des_paramètres)

✓ La procédure est utilisée (appelée) comme une instruction. Elle ne peut pas être utilisée dans une expression arithmétique ou logique, ni passée en paramètre à une autre procédure ou fonction.

✓ Dans une procédure, l'utilisation de l'instruction **return** reste possible, mais elle n'est pas suivie d'un résultat (**return ;**).

Remarque 1 : Les instructions **scanf** et **printf** sont, en réalité, deux fonctions. ① **scanf** : En plus des valeurs lues à partir du clavier, elle retourne le nombre de variables correctement lues à partir du clavier.

② **prinf** : En plus de l'affichage sur écran, cette fonction retourne le nombre de caractères affichés.

Remarque 2 : Si une fonction est appelée comme une procédure (comme une instruction), la fonction sera exécutée mais la valeur de retour n'est pas exploitée (utilisée). **Exemples** : scanf et printf.

IV – Passage de paramètres par valeur (par copie) :

Dans ce type de passage de paramètres, la valeur du paramètre est copiée dans une variable locale sans toucher la valeur d'origine. C'est cette variable locale qui est utilisée dans la fonction (ou la procédure) appelée. Aucune modification de la variable locale dans la fonction ne modifie la variable passée en paramètre, parce que ces modifications ne s'appliquent qu'à une copie de cette dernière.

Exemple : La fonction **facto** qui calcule la factorielle d'un entier positif **n** peut être écrite comme suit :

```
int facto(int n) { int f = 1 ; for( ; n > 1 ; n--) f *= n ; return f ;}
```

Explication : Malgré que la fonction **facto** modifie le paramètre **n** en le décrémentant (**n--**), il retrouvera sa valeur initiale à la fin de la fonction car cette dernière travaille sur une copie de **n** (**n** est passé par valeur).

✓ Un paramètre passé par valeur (par copie) peut recevoir une valeur constante, une variable, une expression ou même un appel à une fonction. **Exemples** : facto(5), facto(x), facto(n1 + n2), facto(maximum(n1, n2)).

V – Passage de paramètres par adresse (par référence) :

Dans ce type, il n'y a pas de copie de la valeur du paramètre mais la copie de son adresse. La procédure (ou fonction) travaille directement avec la variable passée en paramètre, elle peut ainsi modifier sa valeur. Toute modification du paramètre dans la procédure appelée entraîne la modification de la variable passée en paramètre.

✓ Un paramètre passé par adresse à une procédure (**rarement** à une fonction) est précédé par le symbole *.

Exemple : int *a ; La variable **a** est dite un **pointeur** sur un **int**. (***a**) est la **valeur pointée** par le pointeur **a**.

Une procédure qui incrémente la valeur d'un **int a** peut être écrite comme suit :

```
void incrementer(int *a) { (*a)++ ; /* ou *a = *a + 1 ; */ }
```

Exemple d'appel de la procédure : int n = 10 ; incrementer(**&**n) ; printf("%d", n) ; // n aura la valeur 11.

A l'appel, il faut utiliser le symbole **&** (lu adresse) comme déjà utilisé dans scanf.

VI – Variables locales et variables globales

✓ Une **variable globale** est déclarée à l'extérieur du corps de toute fonction, et peut donc être utilisée dans n'importe quelle procédure ou fonction (y compris la fonction **main**). En général, les variables globales sont déclarées au début du programme, immédiatement après les directives **#include** et **#define**.

✓ Une **variable locale** ne peut être utilisée que dans la fonction ou le bloc ({ …bloc… }) où elle est définie.

Remarques : ① Si une variable globale n'est pas initialisée par l'utilisateur, elle est initialisée par défaut à zéro. ② Si une variable locale n'est pas initialisée par l'utilisateur, sa valeur est indéterminée. **Attention** !

VII – LES EXERCICES – LES PROCEDURES ET LES FONCTIONS

Exercice 1 : Tables de multiplication

```
7 × 1 = 7
7 × 2 = 14
7 × 3 = 21
7 × 4 = 28
7 × 5 = 35
7 × 6 = 42
7 × 7 = 49
7 × 8 = 56
7 × 9 = 63
7 × 10 = 70
```

Q1) Ecrire une procédure ***afficherMultiples*** qui permet d'afficher la table de multiplication d'un entier **n** compris entre **1** et **10** (**1 ≤ n ≤ 10**). En face, la table de multiplication de **n = 7.**

Q2) Ecrire le programme principal (la fonction ***int main( )***) qui permet de lire un entier ***n*** compris entre 1 et 10 et d'afficher sa table de multiplication.

Q3) Modifier le programme principal pour afficher toutes les tables de multiplication de 1 à 10.

Exercice 2 : Affichage d'étoiles

Pour chaque procédure, écrire un programme C qui permet de la tester.

Q1) Ecrire une procédure ***ligne(n)*** qui permet d'afficher une ligne de **n** étoiles '*' (astérisques).

Exemple : ********** est une ligne de 10 étoiles.

Q2) En utilisant la procédure ***ligne***, écrire une procédure ***rectangle(n, m)*** qui permet d'afficher un rectangle d'étoile de ***m*** lignes de **n** étoiles. **Exemple** : Ci-dessous un rectangle de **3** lignes de **20** étoiles.

```
********************
********************
********************
```

Q3) En utilisant la procédure ***rectangle***, écrire une procédure ***carre(n)*** qui permet d'afficher un carré d'étoiles de **n** lignes de **n** étoiles. Ci-dessous un carré de 4 lignes de 4 étoiles.

```
****
****
****
****
```

Exercice 3 : Ordre croissant

Q1) Pour deux entiers ***a*** et ***b***, écrire une procédure ***Trier2*** qui permute ***a*** et ***b***, si nécessaire, pour que l'état de sortie soit ***a ≤ b*** (***a*** et ***b*** seront dans l'ordre croissant).

Q2) En utilisant la procédure ***Trier2***, écrire une procédure ***Trier3*** sur 3 entiers ***(a, b et c)*** qui permet d'avoir les trois variables ***a, b et c*** dans l'ordre croissant ***(a ≤ b ≤ c)***.

Q3) Ecrire un programme C qui permet de lire 3 entiers (***x***, ***y***, ***z***) puis les afficher dans l'ordre croissant.

Exercice 4 : Maximum/Minimum

Q1) Ecrire une fonction ***max2(x, y)*** qui retourne le plus grand de deux nombres entiers ***x*** et ***y***.

Q2) Utiliser la fonction ***max2*** pour écrire une deuxième fonction ***max3(x, y, z)*** qui calcule le plus grand de trois nombres entiers ***x***, ***y*** et ***z***.

Q3) Ecrire un programme C qui permet de lire trois entiers ***a***, ***b***, ***c*** et d'afficher leur maximum.

Q4)** Adapter les fonctions précédentes ainsi que le programme principal pour afficher le **minimum** de trois entiers ***a, ***b*** et ***c.***

*** Exercice 5 : Nombres Premiers**

Un nombre est dit premier s'il n'admet que deux diviseurs : 1 et lui-même.

Q1) Ecrire une fonction ***Premier(n)*** qui permet de vérifier si un entier ***n*** est premier ou pas.

Q2) Ecrire une procédure qui permet d'afficher les ***m*** premiers nombres premiers.

Exemple : Si **m = 8**, la procédure doit afficher : 2 3 5 7 11 13 17 19.

Q4) Ecrire un programme C qui permet de lire un entier ***m*** et d'afficher les ***m*** premiers nombres premiers.

Exercice 6 : Fonctions itératives ou récursives ?

Pour chaque fonction, écrire un programme C qui permet de la tester.

Q1) ***Factorielle*** : Ecrire une fonction ***Facto(n)*** qui permet de calculer ***n !*** **= 1 * 2 * 3 * …* n**

Q2) ***Puissance*** : Ecrire une fonction ***Puiss(x, n)*** qui renvoie $x^n = x * x * \ldots * x$ (n fois)

Q3) ***Factorielle*** : Ecrire une fonction récursive ***FactoRec(n)*** qui permet de calculer ***n !*** en sachant que :

$$n ! = n * (n-1) ! \quad \text{et} \quad 0 ! = 1 ! = 1$$

Q4) ***Puissance*** : Ecrire une fonction récursive ***PuissRec(x, n)*** qui permet de calculer x^n en utilisant le fait que :

$$x^n = \begin{cases} 1 & Si\ n = 0 \\ x^{n-1} \times x & Si\ n > 0 \end{cases}$$

Q5) ***Puissance*** : Ecrire une fonction récursive ***PuissRec2(x, n)*** qui permet de calculer x^n en utilisant le fait que :

$$x^n = \begin{cases} 1 & Si\ n = 0 \\ x^{n/2} \times x^{n/2} & Si\ n > 0\ et\ n\ pair \\ x^{(n-1)/2} \times x^{(n-1)/2} \times x & Si\ n > 0\ et\ n\ impair \end{cases}$$

*** Q6)** L'algorithme d'Euclide permettant de calculer le **PGCD** (Plus Grand Commun Diviseur) de deux entiers strictement positifs ***A*** et ***B*** tels que ***A > B*** est défini comme suit :

$$PGCD(A,B) = \begin{cases} PGCD(B,\ A \bmod B) & Si\ B \neq 0 \\ A & Si\ B = 0 \end{cases}$$

Ecrire une fonction récursive permettant de déterminer le ***PGCD*** de ***A*** et ***B***.

*** Q7)** ***Suite de Fibonacci*** : Ecrire une fonction récursive ***Fibo(n)*** permettant de calculer le n^{ieme} terme de la suite de ***Fibonacci*** définie par :

$$U_n = \begin{cases} 0 & Si\ n = 0 \\ 1 & Si\ n = 1 \\ U_{n-1} + U_{n-2} & Si\ n \geq 2 \end{cases}$$

Q8) Pour un entier positif **n (n ≥ 0)**, écrire une fonction itérative puis une autre récursive qui permet :

- Calculer le nombre des chiffres de **n**.
- Calculer la somme des chiffres de **n**.
- Calculer le produit (la multiplication) des chiffres de **n**.
- * Vérifier si un chiffre (0 ≤ chiffre ≤ 9) existe dans le nombre **n**.
- * Calculer le nombre d'occurrences d'un chiffre (0 ≤ chiffre ≤ 9) dans le nombre **n**.

*** : Travail facultatif**

Université Mohammed Seddik BENYAHIA – JIJEL –
Faculté des Sciences Exactes et Informatique **2019 / 2020**
Département de Mathématiques et Informatique

– Algorithmique – TP N°6 – LES LISTES LINEAIRES CHAINEES –

I- Les enregistrements

I.1- Rappel de cours : Un enregistrement est une structure de données permettant de regrouper dans une seule entité un ensemble de données ***de types différents*** associées à un même et seul objet. Chaque donnée est appelée un champ. Chaque champ est identifié par ***un nom*** qui permet d'y accéder directement et ***un type***. Le type d'un champ peut être simple (***int, float, char, …***) ou structuré (***tableau, matrice, enregistrement***).

Exemple : Pour manipuler les informations d'un étudiant, on peut créer un type **Etudiant** comme suit :

Type Etudiant = **Enregistrement**

Nom : **Chaine**
Prenom : **Chaine**
Age : **Entier**
Moyenne : **Réel**

Fin

Var e1, e2 : Etudiant /* e1 et e2 deux variables de type **Etudiant**, elles ont chacune 4 champs : Nom, Prénom, … */

I.2- Les enregistrements en langage C

En langage C, le type enregistrement est déclaré avec le mot clé ***struct***.

Pour déclarer le type Etudiant en langage C, on écrit :

```
struct Etudiant {
      char Nom[40] ; /* une chaine de 39 caractères maximum */
      char Prenom[40] ; /* une chaine de 39 caractères maximum */
      int Age ;
      float Moyenne ;
} ;
struct Etudiant e1, e2 ; /* déclaration de deux variables de type struct Etudiant */
```

Remarques : ① En général, on utilise le mot clé ***typedef*** pour renommer le type **struct** Etudiant en Etudiant comme suit : `typdef struct Etudiant ;`

La déclaration des deux variables e1 et e2 sera simplifiée comme suit :

```
typdef struct Etudiant ;
Etudiant e1, e2 ; /* Deux variables de type Etudiant */
```

② L'instruction ***typedef*** peut être placée avant ou après la déclaration du type `struct Etudiant { … } ;`

I.3- Accès aux champs d'un enregistrement

Pour accéder à un champ, il faut utiliser l'opérateur point (.).

Exemples : - Pour affecter la valeur 20 au champ Age de l'étudiant e1, on écrit : `e1.Age = 20 ;`

- Pour donner la valeur 11.5 au champ Moyenne de l'étudiant e1, on écrit : `e1.Moyenne = 11.5 ;`

II- Les pointeurs en langage C

Rappel : Un pointeur est une variable qui contient l'adresse mémoire d'une autre variable stockée en mémoire.

II.1- Déclaration : Pour déclarer une variable de type pointeur sur un type donné (simple ou structuré), il suffit de précéder sa déclaration par un astérisque (*).

Exemple1 : Pour déclarer un pointeur (appelé ptr) sur un entier (***int***), on peut écrire : `int *ptr ;`

Exemple2 : Pour déclarer un pointeur (appelé p) sur un enregistrement de type Etudiant, on peut écrire :

```
Etudiant *p ;
```

II.2- Accès à la donnée : Pour accéder à la variable pointée, on écrit par exemple : `*ptr = 20 ;` ou

```
printf("%d", *ptr) ;
```

Remarque : Les pointeurs sont utilisés pour le passage de paramètres par adresse (par variable) dans les procédures (et fonctions), comme déjà vu au **TP N°5 – PROCEDURES & FONCTIONS**.

II.3- Accès aux champs d'un enregistrement pointé

Pour accéder à un champ (le champ Age, par exemple) d'un enregistrement pointé (par la variable p), la notation `(*p).Age` reste valable. Cependant, le langage C, propose une notation plus simple selon la syntaxe suivante : `p->Age`. Pour affecter à l'âge la valeur 20, on écrit alors `p->Age = 20` ; au lieu de `(*p).Age = 20 ;`

II.4- Initialisation : La constante **NULL** (en majuscule) est utilisée pour indiquer que le pointeur ne pointe vers aucune donnée valide. On écrit alors : `ptr = NULL ;`

II.5- Allocation mémoire : Pour allouer un espace mémoire à la variable pointée, on utilise l'instruction `malloc` de la bibliothèque `stdlib.h`.

Exemple1 : pour réserver l'espace à un entier on écrit : `ptr = malloc( sizeof( int ) ) ;`

Exemple2 : pour réserver l'espace à un enregistrement de type Etudiant pointée par le pointeur `p`, on écrit :

```
p = malloc( sizeof( Etudiant ) ) ;
```

II.6- Libération de l'espace mémoire : Pour libérer l'espace mémoire (précédemment alloué par `malloc`) et pointé par le pointeur ***ptr***, on utilise l'instruction `free` de la bibliothèque `stdlib.h`.

On écrit alors : `free( ptr ) ;`

III- Les listes en langage C

III.1- Rappel : Une Liste Linéaire Chaînée (LLC) est un ensemble d'éléments (Cellule, Nœud, Maillon) alloués dynamiquement chaînés (reliés) entre eux. Chaque élément est un enregistrement qui contient au moins deux champs : ①- Un champ qui contient l'information (appelé Valeur, **Val**, Info ou Data).

②- Un champ qui est un pointeur sur l'élément suivant (appelé **Suiv** ou Suivant).

III.2- Déclaration : Dans un programme C, le type Liste (d'entiers) peut être déclaré comme suit :

```
typedef struct Element Element;
typedef Element* Liste;
struct Element {
            int val; /* val Contient l'information utile */
            Liste suiv; /* suiv un pointeur vers l'élément suivant */
            };
Liste L, P, Q, Ptr ; /* L, P, Q, Ptr : des variables de type Liste ou pointeur sur Element */
```

III.3- Accès aux champs

✓ On accède au champ **val** par **L->val**. Exemples : `L->val = 25 ;` ou `printf("%d\n", L->val) ;`

✓ On accède au champ **suiv** par **L->suiv**. Exemples : `L->suiv = NULL ;` ou `if(L->suiv != NULL)…`

III.4- Allocation (création)

```
L = malloc( sizeof( Element) ) ;
```

✓ Si l'opération d'allocation s'est bien déroulée, l'instruction ***malloc*** retourne une adresse mémoire valide (valeur non nulle). Sinon, elle retourne NULL.

III.5- Libération (destruction)

✓ L'instruction `free(L) ;` permet de détruire (supprimer) un élément créé avec l'instruction ***malloc*** et pointé par le pointeur ***L***.

IV- Exercice

Objectif : Maîtriser les listes à travers la recherche des nombres premiers.
Dans ce TP, on s'intéresse à une liste d'entiers, mais les principes abordés restent valables pour des listes de n'importe quel type de données.

Le but de cet exercice est de trouver tous les nombres premiers inférieurs à un entier **Max** (par exemple 10^6).
Une méthode rapide pour vérifier si un nombre entier **N** est premier est de tester s'il accepte un diviseur parmi les nombres premiers (déjà connus) qui sont inférieurs à sa racine carrée.
Nous utilisons une liste linéaire chaînée pour stocker les nombres premiers. A chaque fois qu'un nombre premier est découvert (trouvé), il est inséré à la fin de cette liste.

Q1) Écrire une procédure ***InsererTete(L, N)*** qui permet d'insérer un entier ***N*** au début de la liste ***L*** (La liste ***L*** peut être vide).

Q2) Écrire une procédure ***AfficherListe(L)*** qui permet d'afficher les éléments d'une liste ***L***. Si ***L*** est vide, la procédure doit afficher le message '*** Liste vide ***'.

Q3) Tester les deux procédures écrites en **Q1** et **Q2** : Dans le programme principal (fonction ***int main()***), créer une liste vide (***L***) ; Insérer dans ***L***, les nombres suivants : 2, 3, 5, 7 puis afficher le contenu de cette liste. Dans quel ordre les nombres 2, 3, 5 et 7 sont affichés ?

Q4) Écrire une procédure ***InsererQueue(L, N)*** qui permet d'insérer un entier ***N*** à la fin de la liste ***L*** (***L*** peut être vide). **Remarque** : Il est possible d'utiliser la procédure ***InsererTete***, si nécessaire.

Q5) Dans le programme principal, remplacer les insertions en tête de la liste par des insertions à la fin. Cette fois-ci, dans quel ordre les nombres 2, 3, 5 et 7 sont affichés ?

Q6) Écrire une fonction ***Premier(L, N)*** où ***L*** est une liste qui contient les nombres premiers déjà découverts et ***N*** un entier. Cette fonction teste si ***N*** est un nombre premier ou non en vérifiant si ***N*** accepte un diviseur parmi les nombres premiers qui sont déjà dans la liste ***L*** et sont inférieurs à sa racine carrée.

Q7) Écrire une procédure ***Lister_Nombres_Premiers(L, Max)*** qui permet la construction de la liste ***L*** en cherchant tous les nombres premiers compris entre ***2*** et ***Max***.

Le principe de cette procédure est le suivant :

- On ne teste que les nombres impairs, car tous les nombres pairs sont divisibles par **2**.
- Au début la liste ***L*** est vide ***(L = { })***. - On rajoute 3 à la liste ***(L = { 3 })***.
- On commence la recherche par le nombre 5, le premier nombre dans la liste est 3, mais 3 est supérieur à la racine carrée de 5 alors 5 est premier et est ajouté à la liste ***(L = { 3, 5 })***.
- On passe à 7, le premier nombre dans la liste est 3, mais 3 est supérieur à la racine carrée de 7 alors 7 est premier et est ajouté à la liste ***(L = { 3, 5, 7 })***.
- On passe à 9, on le divise par 3, 3 est un diviseur de 9, alors 9 n'est pas premier.
- On passe à 11, on le divise par 3, il n'est pas divisible par 3. Le nombre premier suivant dans la liste est 5, mais 5 est supérieur à la racine carrée de 11 alors 11 est premier et est ajouté à la liste ***(L = { 3, 5, 7, 11 })***.
- On passe à 13, 15, 17, 19, … et on continue, avec le même principe, jusqu'à trouver **tous les nombres premiers impairs** inférieurs ou égaux à ***Max***.
- A la fin, on rajoute au début de la liste, le nombre **2** qui est le **seul nombre premier pair**.

Q8) Écrire une fonction ***Longueur(L)*** qui donne le nombre d'éléments de la liste ***L***.
Combien de nombre premier inférieur à 10^6 ?

Q9) Écrire une fonction ***Somme(L)*** qui donne la somme des éléments de la liste ***L*** (La somme de tous les nombres premiers inférieurs ou égaux à ***Max***).

Q10) Un programme bien conçu, doit impérativement supprimer tous les éléments créés dynamiquement (avec l'instruction **malloc**) en utilisant l'instruction **free**. Écrire une procédure ***SupprimerListe(L)*** qui permet de supprimer tous les éléments de la liste ***L***. Appeler cette procédure à la fin du programme principal.

Q11) Donner une version récursive des fonctions/procédures suivantes : ***AfficherListe***, ***InsererQueue***, ***Longueur, Somme, SupprimerListe.***

PARTIE 6– REFERENCES BIBLIOGRAPHIQUES

[1] M. BELAID, Algorithmique & Programmation en PASCAL, Cours, Exercices, Travaux Pratiques, Corrigés, Bouira-Algérie: Eurl Pages Bleues Internationales, 2008.

[2] D.-E. ZEGOUR, Structures de Données et de Fichiers Programmation PASCAL et C, Alger-Algérie: Editions Chihab, 1996.

[3] J. COURTIN, I. KOWARSKI et J. ARSAC, Initiation à l'algorithmique et aux structures de données 1. Programmation structurée et structures de données élémentaires, Paris - France: Dunod, 1994.

[4] J. COURTIN et I. KOWARSKI, Initiation à l'algorithmique et aux structures de données 2. Récursivité et structures de données avancées, Paris - France: Dunod, 1995.

[5] C. DELANNOY, Apprendre à programmer en Turbo C, Alger-Algérie: Chihab - Eyrolles, 1994.

[6] K. KHALFAOUI, «Introduction à la programmation, Support de Cours,» Université de Jijel, Jijel, 2018.

[7] V. L. Damien Berthet, Algorithmique & programmation en langage C vol.1 : Supports de cours., Istanbul, Turquie : Université Galatasaray, 2014.

[8] V. L. Damien Berthet, Algorithmique & programmation en langage C vol.2 : Sujets de travaux pratiques., Istanbul, Turquie: Université Galatasaray, 2014.

[9] V. L. Damien Berthet, Algorithmique & programmation en langage C vol.3 : Corrigés de travaux pratiques., Istanbul, Turquie: Université Galatasaray, 2014.