République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université Aboubekr BELKAÏD - TLEMCEN
Faculté des Sciences

Département d'Informatique

TITRE

POLYCOPIÉ DE TRAVAUX PRATIQUES CORRIGES ALGORITHMIQUE

Adressé aux étudiants niveau : Licences (L 2)
Ingénieur (ING 2)

Domaine : Algorithmique
Filière : Licence - Ingénieur

Etabli Par :

Mr. ETCHIALI Abdelhak

Mr. MERZOUG Mohammed

Année 2024-2025

AVANT-PROPOS

Ce polycopié de travaux pratiques avec rappels de cours est destiné aux étudiants de deuxième année informatique et à toutes les filières qui ont, dans leur canevas, le cours d'algorithmique et de programmation en langage C. Il contient un ensemble de rappels de cours et de travaux pratiques avec leurs solutions en langage C.

Les travaux pratiques proposés inclus les fonctions, la récursivité, les différents types de tri et les structures de données (les listes chaînées, les piles, les files et les arbres).

L'objectif de ce manuscrit est d'initier les étudiants à la résoution de problèmes en proposant différents algorithmes et programmes en langage C en se référant aux notions de bases données en rappels de cours.

Ce polycopié est accompagné d'un CDROM qui contient les solutions de l'ensemble des TPs proposés dans ce manuscrit.

TABLE DES MATIERES

TP N°1 : LES FONCTIONS

I. Rappels de Cours

1- Pourquoi les fonctions ?

Lorsqu'une partie de code doit être exécutée plusieurs fois à des endroits différents ou réutilisée ultérieurement, la solution est de définir cette partie du code séparément au code principal et de l'invoquer autant de fois selon le besoin. Cette dernière est appelée **fonction**.

La notion de fonction est très intéressante en algorithmique car elle permet de :

- décomposer le problème en plusieurs sous-problèmes distincts et les résoudre sous forme de fonctions,
- réutiliser les fonctions déjà développés soit dans le programme soit ailleurs.

2- Définition

Toute fonction doit être définie à l'extérieur duprogramme principal. Elle se compose de :

- l'**en-tête** qui est constitué : du **nom de la fonction** (identificateur), la liste de ses **paramètres formels** et le **type de retour** qu'elle renvoie.
- **déclarations locales** (constantes ou variables)
- le **corps** de la fonction est composé, d'une ou d'un ensemble d'**instructions** ; avec au moins une instruction **de retour** qui renvoie le résultat de la fonction. Le coprs de la fonction est délimité par un **Début** et une**Fin**

```
Fonction Nom_de_la_fonction (paramètres formels) :type de retour
     Déclarations locales
     liste des variables et constantes
     Début(Corps de la fonction)
     <instruction 1>
     <instruction 2>
           …
           Type de retour
     Fin (Corps de la fonction)
```

3- Appel de la fonction

Pour appeler une fonction, il suffit d'écrire son nom suivi des valeurs de ses paramètres effectifs tout en respectant l'ordre et le type des paramètres formels.

```
Nom_de_la_fonction (paramètres effectif)
```

Lors de l'appel :

- le **1er** paramètre effectif passe sa valeur au **1er** paramètre formel,
- le **2ème** paramètre effectif passe sa valeur au **2ème** paramètre formel
- ...
- Jusqu'au dernier paramètre de la fonction.

Un appel de fonction provoque l'exécution complète du corps de cette fonction.

Exemple

Paramètres formels

```
Fonction Passage_parametres (v1, v2:entier ; v3: reel) : entier
```

Appel de la fonction

Paramètres effectives

```
Passage_parametres (10, 30, 15, 56)
```

4- Passage de paramètres

Il y a deux manières pour effectuer le passage de paramètres:

1. **Passage par valeur :** Il consiste à copier la valeur de la variable passée en paramètre dans une variable locale. Aucune modification de la variable locale dans la fonction appelée ne peut modifier la variable passée en paramètre, parce que ces modifications ne s'appliquent qu'à l'intérieur du corps de la fonction (la copie de la variable).
2. **Passage par adresse :** il consiste à passer comme paramètre la variable elle-même. Toute modification apportée par la fonction appelée entraine la modification de la variable passée en paramètre.

Conseils à prendre

- Le choix du ***Nom_de_la_fonction*** doit correspondre à la tâche réalisée par cette fonction.

- **Les Paramètres sont ordonnés** et mis entre deux parenthèses. Ces dernières sont obligatoires même si aucun paramètre n'existe.
- Il est indispensable de respecter l'ordre et les types des paramètres au moment de l'appel de la fonction (passage de paramètres).
- **Variables locales et globales** : Les variables globales sont connues dans tous le programme, par contre les variables locales sont connues uniquement dans les fonctions où elles sont déclarées. On parle ici de la portée des variables.

II. Enoncé TP N° 01 : Les Fonctions

Exercice 1

On considère un tableau T de taille N composé d'un ensemble d'entiers. Ecrire une fonction *Supprimer_Doublons* qui permet de supprimer tous les éléments redondants du tableau T en respectant les conditions suivantes :

- la taille du tableau **T** devient $\boldsymbol{M \leq N}$.
- aucun tableau intermédiaire ne doit être utilisé ;
- l'ordre des éléments reste celui de la séquence de départ.

Exemple

Soit le tableau T à N=10 éléments :

Indice	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
T	15	4	19	4	8	11	11	3	4	19

Après suppression des éléments redondants on obtient un tableau T à M = 6 éléments :

Indice	**0**	**1**	**2**	**3**	**4**	**5**
T	15	4	19	8	11	3

Exercice 2

1. Ecrire une fonction nommée *NB_Chiffres* qui permet de calculer le nombre de chiffres d'un nombre entier positif.
2. Ecrire une fonction nommée *Trier_Nombres* qui permet de trier un tableau d'entiers par ordre croissant suivant le nombre de chiffres de ses éléments en faisant appel à la fonction *NB_Chiffres*.

Exemple

Les éléments du tableau suivant sont triés par ordre croissant suivant le nombre de chiffres.

1	31	24	235	215	1223	12236	5648975

Exercice 3

Une rotation dans un tableau consiste à glisser un certain nombre d'éléments à gauche ou à droite. Ecrire une fonction *Rotation* qui permet d'effectuer une rotation dans un tableau **A** en indiquant le **Pas** et le **Sens** de la rotation à travers un entier **R** (un nombre négatif correspond à une rotation vers la gauche, un nombre positif à une rotation vers la droite).

Exemple

Soit le tableau :

A	5	9	10	7	4	13

Pour **R=+2** le sens de la rotation **à droite** et **Pas = 2**

A	4	13	5	9	10	7

Pour **R= - 3** le sens de la rotation **à gauche** et **Pas = 3**

A	7	4	13	5	9	10

Exercice 4

Ecrire une fonction *Tasse* qui détecte et supprime les éléments nuls d'un tableau **T** en décalant vers la gauche les éléments qui se trouvent après (la taille du tableau diminue).

Exemple

Soit le tableau **T** de taille **N = 8** :

2	0	1	5	0	0	31	21

Après exécution de la fonction *Tasse* on obtient un tableau de taille **M= 5** :

2	1	5	31	21

Exercice 5

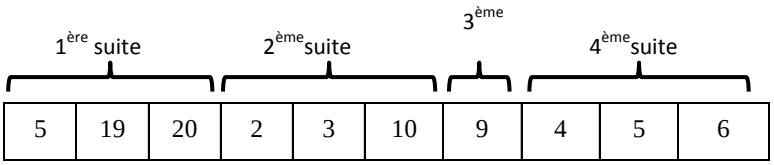
Ecrire une fonction *Suite_Croissante* qui permet de calculer le nombre des suites croissantes dans un tableau d'entiers **T** de taille **N** et de les afficher (une suite croissante est un sous ensemble du tableau initial qui contient des éléments successivement croissants).

Exemple

Soit le tableau **T** de taille **N = 10**:

5	19	20	2	3	10	9	4	5	6

Le nombre de suites croissantes est 4.



Exercice 6

1. Ecrire une fonction nommée *rech_element* qui permet de rechercher un entier **X** dans un tableau d'entiers **T**.
2. Ecrire une fonction nommée *rech_tab* qui permet de calculer le nombre d'éléments communs entre deux tableaux d'entiers **A** et **B.** cette fonction possèdera comme arguments les deux tableaux et leurs tailles respectives et devra faire appel à la fonction *rech_element*.

Exercice 7

Un carré est dit Magique si la somme de chacune de ses lignes, de chacune de ses colonnes et de ses deux diagonales est la même.

Ecrire une fonction *Carre_Magique* qui vérifie si un carré de dimensions **N** (N lignes, N colonnes) est magique.

Exemple : Soit le carré Magique d'ordre **N=3**

2	7	6
9	5	1
4	3	8

Exercice 8

Ecrire une fonction qui permet de trier une matrice **Mat** de dimensions **N**x**M**:

- suivant ses lignes ou suivant ses colonnes selon le caractère choisi (‘**L**’ pour les lignes ou ‘**C**’ pour les colonnes)
- et en fonction de la somme des valeurs que contient chaque ligne ou chaque colonne, dans l’ordre croissant ou décroissant selon le caractère choisi (‘**D**’ pour décroissant, ‘**C**’ pour croissant).

TP N°2 : LA RECURSIVITE

I. Rappels de Cours

1- Introduction

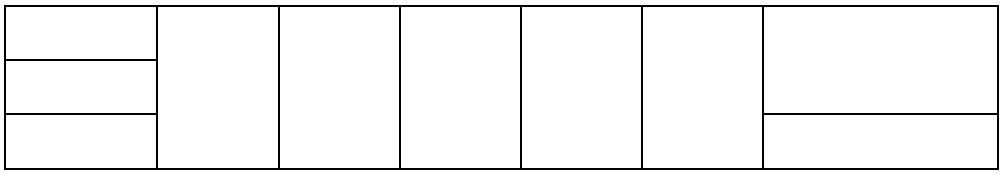
Le processus récursif est un mode de raisonnement qui favorise la décomposition de problèmes en des sous-problèmes de même nature, d'une manière récurrente. Ce processus est inspiré de la méthode classique « Diviser pour Reigner » pour résoudre les problèmes complexes.

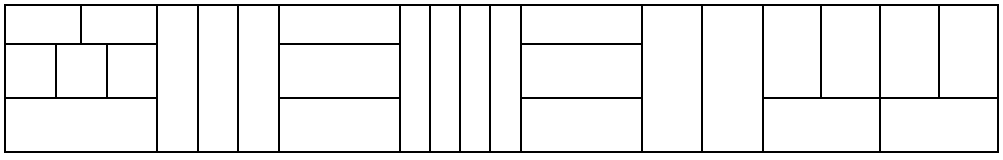
Exemple

Le rectangle ci-dessous représente le problème initial :

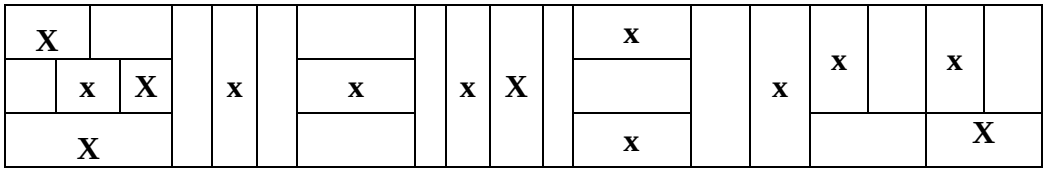
Par récursivité, on peut le décomposer en 03 sous problèmes à résoudre, de même nature que le problème initial :

De la même manière, on peut réitérer le processus de décomposition autant de fois et appliqué le principe pour résoudre le problème initial au sous problèmes:





Le processus de décomposition s'arrête lorsque tous les sous problèmes générés ont une solution évidente.



X : solution évidente d'un sous problème

La solution du problème initial consiste à synthétiser de manière récursive toutes les solutions évidentes de tous les sous problèmes générés auparavant par le processus de décomposition.

2- Définition

Une fonction est dite récursive lorsqu'elle est définie en fonction d'elle-même. Pendant l'exécution, une fonction récursive fait appel à elle-même.

3- Exemple explicatif : TOURS de HANOI

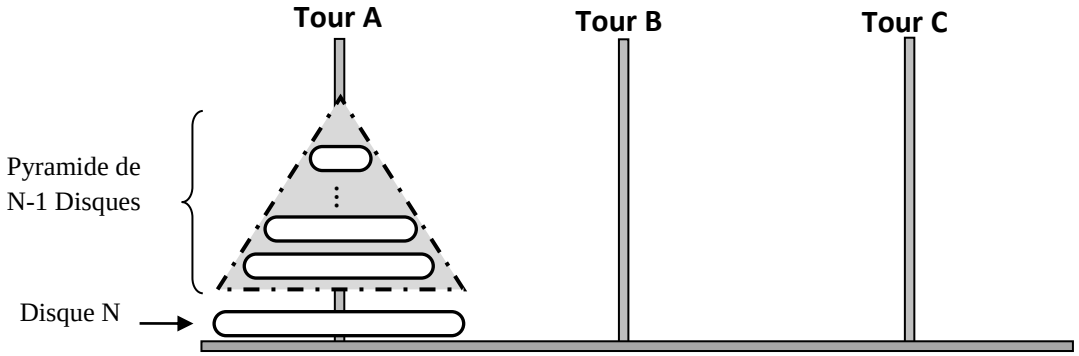
On dispose d'un plateau sur lequel sont fixés trois tours **A, B, C**. Au départ, les **N** disques, de tailles différentes et toutes distinctes, sont enfilés sur **la tour A** d'une façon décroissante de telle sorte que chaque disque ait un diamètre inférieur à son prédécesseur.

L'objectif, est de déplacer les **N** disques de **la tour A** vers **la tour C** en utilisant uniquement **la tour B**, tout en respectant les règles suivantes :

- On ne peut déplacer qu'un seul disque par étape et cela doit toujours être le plus petit disque qui se trouve au sommet.
- A chaque étape, la règle d'organisation des disques sur les tours décrite précédemment doit être respectée. (il est interdit de placer un disque de grande taille sur un disque de petite taille).

Solution récursive adoptée

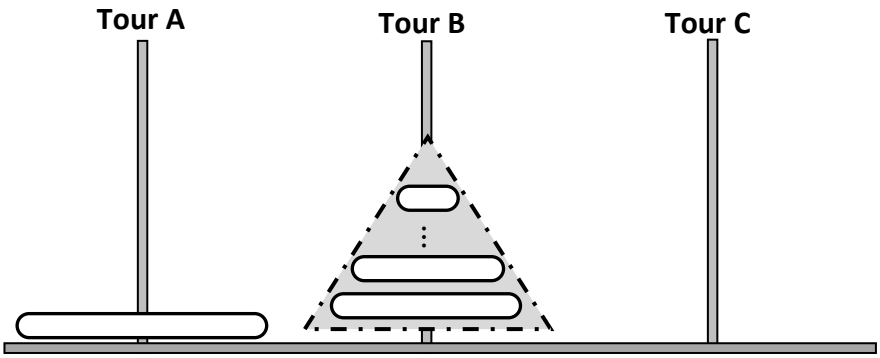
Ce problème peut être analysé d'une manière récursive en le décomposant en sous problèmes de même nature dont la représentation est la suivante :



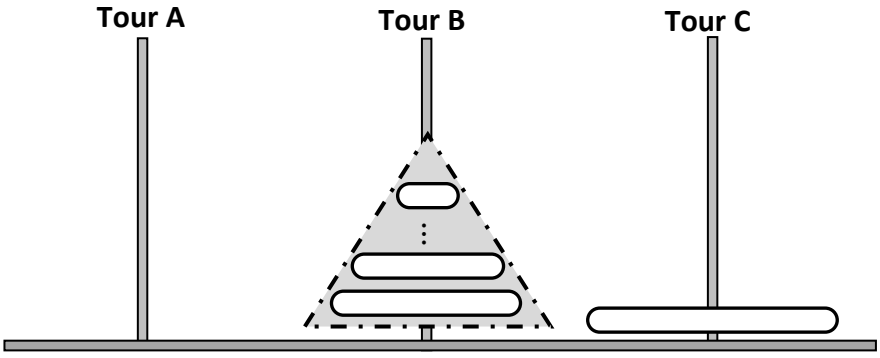
A. Soit une pile de **N** disques, composée du disque le plus grand (Disque **N)** et d'une pyramide de **N-1**disques.

Afin de déplacer le contenu de la tour **A** (Disque N et la pyramide de N-1 disques) vers la tour **C,** on suit les étapes suivantes tout en respectant les contraintes de déplacements :

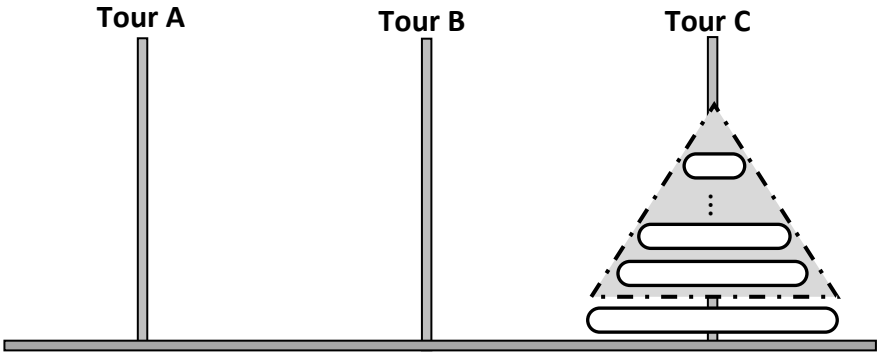
a- Passer la pyramidede **N-1 disques** de la tour **A** à la tour **B**. C'est le seul déplacement possible.



b- Passer le grand disque **N** de la tour **A** à la tour **C**. C'est le seul déplacement possible.

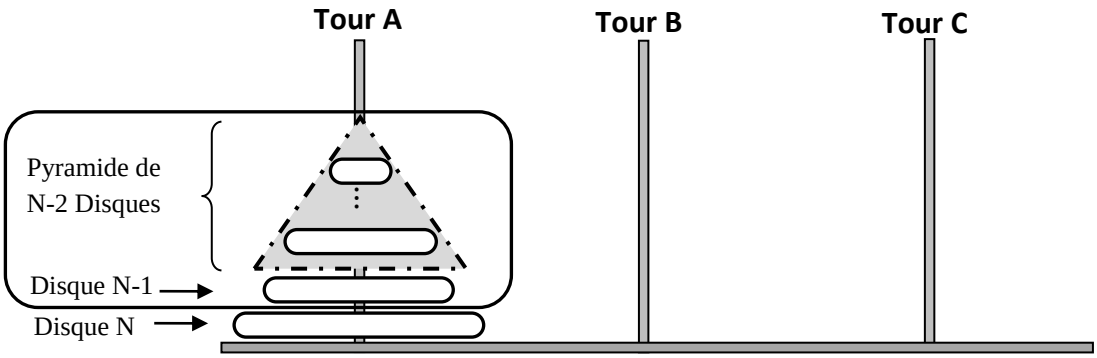


c- Passer **la pyramide** de la tour **B** à la tour **C**. Arrivant à ce stade, le problème est résolu.



Le système appliqué pour transférer la pyramide de **N-1** disques peut être réitéré pour une pyramide de **N-2** disques et ainsi de suite jusqu'à arriver à une pyramide de **1** disque.

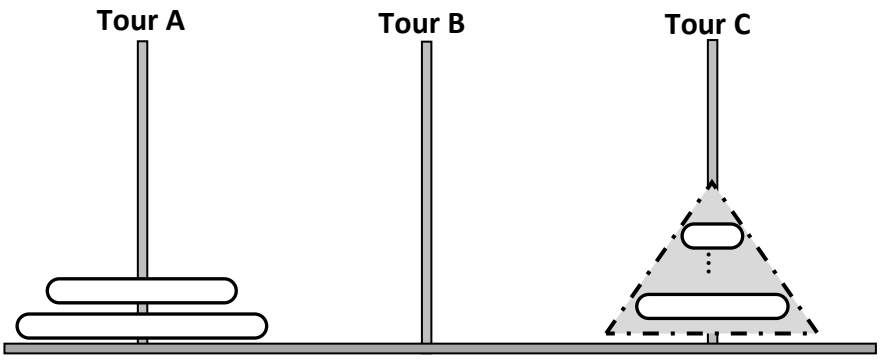
B. Décomposer la pyramide de **N-1** disques en une nouvelle pyramide de **N-2** disque et du disque de taille **N-1**.



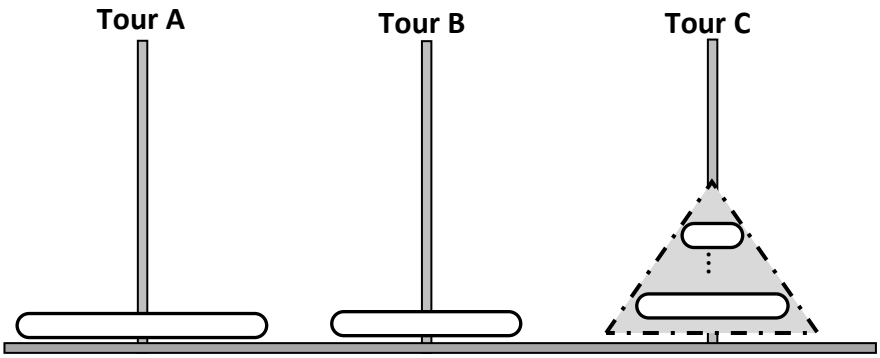
En appliquant les trois opérations précédentes (**a**, **b** et **c**), la résolution du problème devient comme suit :

1. Passer la pyramide de **N-1** disques**,** composée de la pyramide **N-2** disques et du disque **N-1**, de la tour **A** à la tour **B** en utilisant la tour **C** comme intermédiaire.

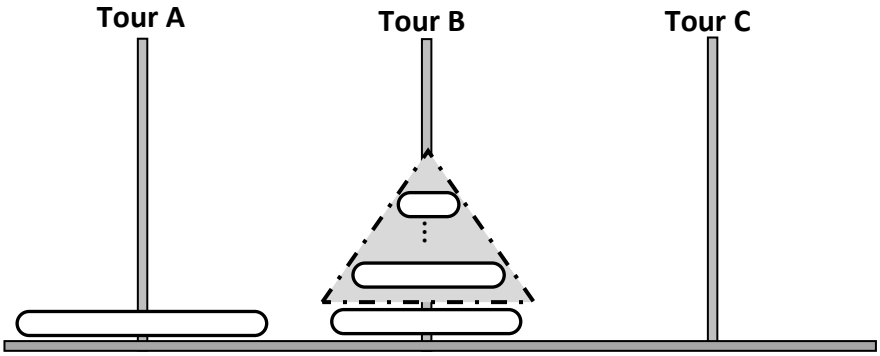
- Passer la pyramide **N-2** disques de la tour **A** à la tour **C**.



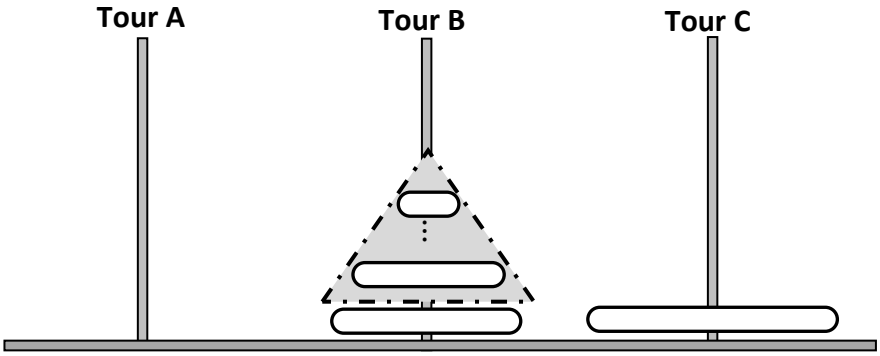
- Passer le disque **N-1** de la tour **A** à la tour **B**.



- Passer la pyramide **N-2** de la tour **C** à la tour **B**.

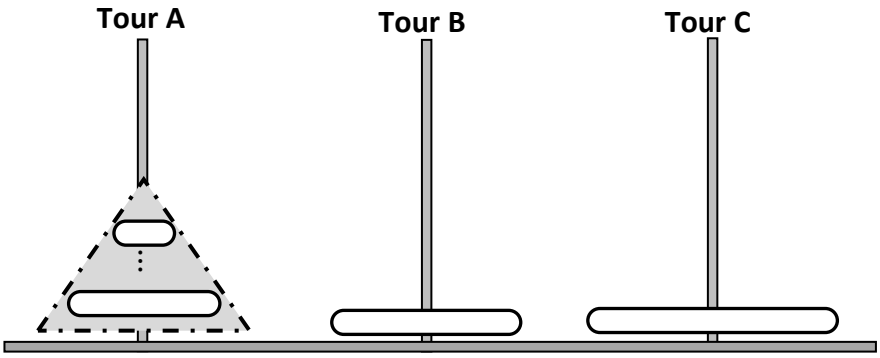


2. Passer le grand disque **N** de la tour **A** à la tour **C**.

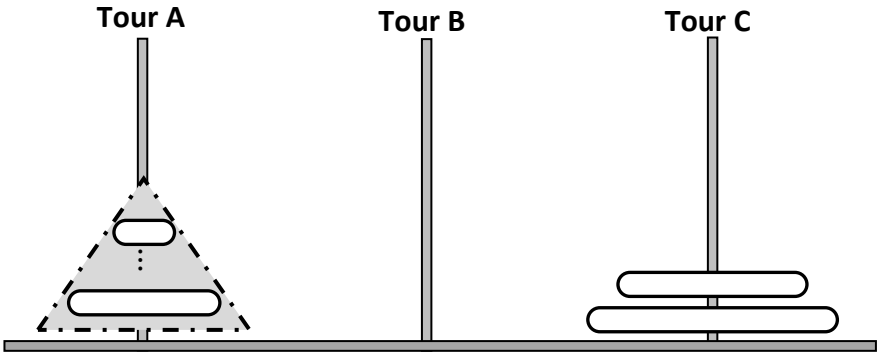


3. Passer la pyramide de **N-1** disques**,** composée de la pyramide **N-2** disques et du disque **N-1**, de la tour **B** à la tour **C** en utilisant la tour **A** comme intermédiaire.

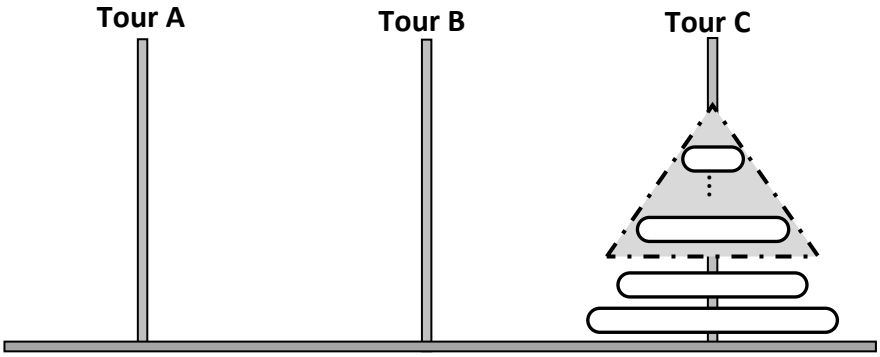
- Passer la pyramide **N-2** de la tour **B** à la tour **A**.



- Passer le disque **N-1** de la tour **B** à la tour **C**.



- Passer la pyramide de **N-2** de la tour **A** à la tour **C**.



L'étape **B** doit être réitérée jusqu'à obtenir une pyramide de taille **1** disque qui représente le cas de base (la solution évidente), définit par le mouvement : **passer de la tour de départ ou intermédiaire vers la tour finale.**

Algorithme

Fonction_HANOÏ (TourA, TourB, TourC, Nb_Disques)

*//{déplacer **Nb_Disques** de la Tour **A** vers la Tour **B** en utilisant la Tour **C** (comme tour auxiliaire)}*

Paramètres :

Nb_Disques : entier

TourA, TourB, TourC: caractère

Début

Si (Nb_Disques == 1) alors {condition d'arrêt}

Afficher ("Déplacez un disque de ",TourA, " vers ",TourB)

Sinon

//{déplacer tous les disques sauf le (Nb_Disques)ième de la tour de départ vers la tour auxiliaire}

Fonction_HANOÏ (TourA, TourB, TourC, Nb_Disques -1)

//{déplacer le (Nb_Disques)ième disque vers la tour destinataire}

Afficher ("Déplacez un disque de ", TourA, " vers ", TourB)

//{déplacer tous les disques de la tour auxiliaire vers la tour destinataire}

Fonction_HANOÏ (TourC, TourB, TourA, Nb_Disques -1)

Fin Si

Fin

Conseils à prendre

Les règles suivantes doivent être respectées pour développer une fonction récursive :

- Le cas de base qui représente la solution évidente, doit être connu et défini dès le début.
- La taille du problème doit diminuer après chaque appel de la fonction $(\boldsymbol{n \rightarrow n-1})$.

II. Enoncé TP N° 02 : La Récursivité

Exercice 1

Ecrire une fonction récursive *Nombre_Occ* qui reçoit comme argument une chaîne de caractères et renvoie le nombre d'occurrences de chaque caractère qu'elle contient en affichant une seule fois les caractères.

Exemple

Pour la chaine « rapport » la fonction *Nombre_Occ* retourne :

- caractère « r » = 2
- caractère « a » = 1
- caractère « p » = 2
- caractère « o » = 1
- caractère « t » = 1

Exercice 2

1. Ecrire une fonction récursive *Afficher* qui permet d'afficher les éléments d'un tableau **T** de taille **N**.
2. Ecrire une fonction récursive *Somme* qui permet de calculer la somme des éléments d'un tableau **T** de taille **N**.
3. Ecrire une fonction récursive *Occurrence* qui retourne le nombre d'occurrences d'une valeur **V** dans un tableau **T** de taille **N**.
4. Ecrire une fonction récursive *kiem_chiffre* qui permet de retourner le $K^{ème}$chiffre à partir de la droite d'un nombre entier.
5. Ecrire une fonction récursive *Classer* qui permet de classer un nouveau entier **V** dans un tableau **T** trié de taille **N**.
6. Ecrire une fonction récursive *Commun* qui permet d'afficher les éléments communs entre deux tableaux entiers déjà triés **T1** et **T2**.

Exercice 3

Ecrire une fonction récursive *Trier* qui permet de trier un tableau **T** d'entier de taille **N** par ordre croissant en se basant sur le principe suivant :

- rechercher le plus grand élément de **T** et le placer en fin du tableau,
- recommencer avec la deuxième plus grande valeur du reste de **T** et la placer en avant-dernière position,
- ainsi de suite jusqu'à avoir parcouru tous les éléments de **T**.

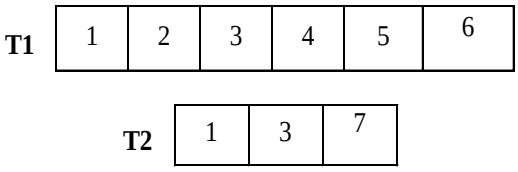
Exercice 4

Soient **T1** et **T2** deux tableaux d'entiers strictement croissants de taille **N** et **M** respectivement. On cherche les éléments communs des deux tableaux.

Ecrire une fonction récursive *Nbr_Commun* permettant d'afficher ces éléments et de retourner leur nombre.

Exemple

Soit les tableaux :



Après exécution de la fonction *Intersection*on obtient :

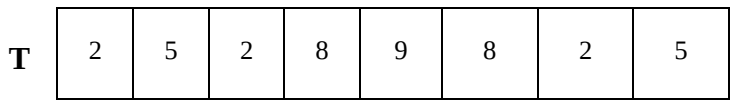
- Eléments communs : **1 et 3**
- Valeur de retour : **2**

Exercice 5

Ecrire une fonction récursive *Verifier_Redondant* qui permet de vérifier si un tableau **T**de taille **N** contient que des éléments redondants (chaque élément est répété au moins 2 fois).

Exemple

Soit le tableau :



Retourne 0
(l'élément 9 n'est pas redondant)

Retourne 1
(tous les éléments sont redondants)

TP N°3 : LE TRI DES TABLEAUX

I. Rappels de Cours

1- Définition

Un tableau (**Tab**) est une structure de donnée qui permet de stocker un certain nombre d'éléments (**Tab[i])** de même type repérés par un indice **i**. Le nombre d'éléments stockés dans un tableau statique est fixe.

2- Déclaration

La déclaration d'un tableau comprend quatre éléments :

- Le nom
- Le nombre de ses dimensions
- La taille (borne inférieure, borne supérieure)
- Le type

```
TypeElements <nomTableau> [Taille 1][Taille 2]…
```

Exemple

- Pour déclarer en langage **C** un tableau **Note** à une dimension de 10 nombres réels, on écrira

```
float Note [10]
```

- Pour déclarer en langage **C** un tableau **Tab** à deux dimensions de nombres entiers, on écrira

```
int Tab [10][20]
```

3- Manipulation

Chaque élément d'un tableau est désigné par un indice qui indique la position de cet élément dans le tableau. Pour accéder à un élément, il suffit d'indiquer entre crochets l'indice de la case contenant cet élément. On choisit la valeur **0** pour la borne inférieure dans le but de faciliter la traduction en langage **C**.

Exemple

int Liste [6] définit un tableau d'entier nommé Liste dont l'indice varie de 0 à 5

Indice	**0**	**1**	**2**	**3**	**4**	**5**
Valeur	21	12	18	67	1	35

Conseils à prendre

- Il est très important d'utiliser correctement les indices (le 1er et le dernier indice);
- L'accès à un élément supérieur ou égal à **N** (Taille du tableau) provoquera systématiquement un résultat faux et généralement une erreur mémoire.

4- Tri des tableaux

Le tri consiste à réarranger un ensemble de **N**éléments d'un tableau **Tab**de telle manière que :

$\mathbf{Tab}[0] \leq \mathbf{Tab}[1] \leq \cdots \leq \mathbf{Tab}[N-1]$: Tri par ordre croissant (ascendant)

$\mathbf{Tab}[0] \geq \mathbf{Tab}[1] \geq \cdots \geq \mathbf{Tab}[N-1]$: Tri par ordre décroissant (descendant)

Exemple

Le résultat du tri croissant du tableau **Tab :**

Tab	21	12	18	67	1	35

Est le suivant :

Tab	1	12	18	21	35	67

Dans ce qui suit, on présente les méthodes de tris les plus utilisées à l'aide d'algorithmes simplifiés.

4.1 Tri à bulle

Cet algorithme porte le nom de tri bulle car, petit à petit, les plus grands éléments du tableau remontent, par le jeu des permutations, en fin de tableau.

La stratégie de cet algorithme est comme suit :

1. On parcourt le tableau en comparant deux à deux les éléments successifs (**Tab**[i-1] et **Tab**[i]) et on les permute s'ils ne sont pas dans l'ordre voulu (croissant ou décroissant).
2. On répète l'**étape 1** tant que des permutations sont possibles.

Algorithme

```
fonction Tri_Bulle(entier[] tab)
   Var  i, j, temp : entier
   début
          pour (i  allant de  n-1 à 1 pas -1) faire
               pour (j  allant de 0 à i-1 pas 1) faire
                    si (tab[j] > tab[j+1]) alors
                          temp ← tab[j];
                         tab[j] ← tab[j+1];
                         tab[j+1] ← temp;
                    fin si
               fin pour
          fin pour
    fin
```

Exemple

Donner le déroulement de l'algorithme de Tri à Bulle au tableau suivant :

Tab	29	9	20	17	7

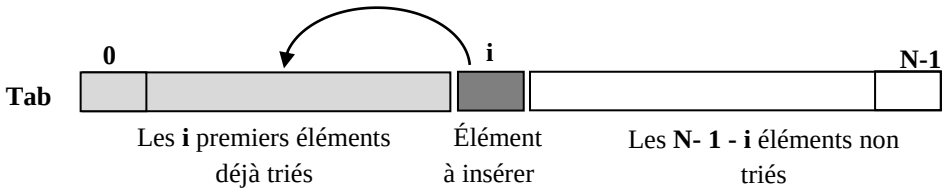
Solution

Itération	Eléments comparés en bleu	Eléments permutés en rouge	Partie triée en vert
i=1	**29** **9** 20 17 7	**9** **29** 20 17 7	
i=2	9 **29** **20** 17 7	9 **20** **29** 17 7	
i=3	9 20 **29** **17** 7	9 20 **17** **29** 7	
i=4	9 20 17 **29** **7**	9 20 17 **7** **29**	9 20 17 7 29
i=5	**9** **20** 17 7 29	9 20 17 7 29	9 20 17 7 29

i=6	9 20 17 7 29	9 17 20 7 29	9 20 17 7 29
i=7	9 17 20 7 29	9 17 7 20 29	9 17 7 20 29
i=8	9 17 7 20 29	9 17 7 20 29	9 17 7 20 29
i=9	9 17 7 20 29	9 7 17 20 29	9 7 17 20 29
i=10	9 7 17 20 29	7 9 17 20 29	7 9 17 20 29

4.2 Tri par insertion

On prend deux éléments qu'on trie dans le bon ordre, puis on insère le 3ème élément à sa place parmi les 2 premiers, ce qi nous donne une liste triée de 3 éléments. Ensuite on insère un 4ème à sa place parmi les 3 autres, etc. La liste est triée au fur et à mesure jusqu'à contenir les N éléments.



Algorithme

```
fonction Tri_Insertion(entier[] tab)
     Var  i,  j, val : entier
     début
          pour (i allant de 1 à n-1 pas 1) faire
               val ← tab[i];
               j ← i;
               Tant Que ((j > 0) et (tab[j-1] > val)) faire
                    tab[j] ← tab[j-1];
                     j ← j-1;
               fin Tant Que
               tab[j] ← val;
          fin pour
     fin
```

Exemple

Donner le déroulement de l'algorithme de Tri par insertion au tableau suivant :

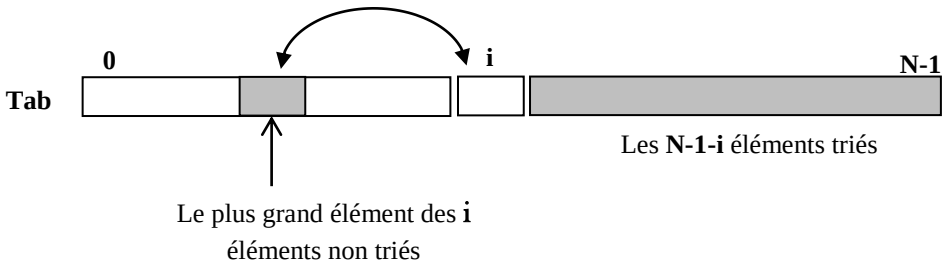
Tab	29	9	20	17	7

Solution

Itération	Partie triée en jaune Partie non triée en bleu	Elément à insérer en rouge	Partie triée en vert
i=1	29 9 20 17 7	29 9 20 17 7	9 29 20 17 7
i=2	9 29 20 17 7	9 29 20 17 7	9 20 29 17 7
i=3	9 20 29 17 7	9 20 29 17 7	9 17 20 29 7
i=4	9 17 20 29 7	9 17 20 29 7	7 9 17 20 29

4.3 Tri par sélection

On sélectionne le plus grand élément du tableau et on le place à la position N (dernière case) par permutation. On réitère la même procédure pour les N-1 éléments restant du tableau.



Algorithme

```
fonction Tri_Selection(entier[] tab)
     Var  i, j, temp, pg : entier
     debut
          i ← n-1;
          Tant Que (i > 0) faire
               pg ← 0;
               pour (j allant de 0 à i pas 1) faire
                    si (tab[j] > tab[pg]) alors
                         pg ← j;
                    fin si
               fin pour
               temp ← tab[pg];
               tab[pg] ← tab[i];
               tab[i] ← temp;
               i ← i-1;
          fin Tant Que
      fin
```

Exemple

Donner le déroulement de l'algorithme de Tri par sélection au tableau suivant :

Tab	29	9	20	17	7

Solution

Itération	Elément cherché en bleu Elément sélectionné en jaune	Eléments permutés en rouge	Partie triée en vert
i=1	**29** 9 20 17 **7**	**7** 9 20 17 **29**	7 9 20 17 **29**
i=2	7 9 **20** **17** 29	7 9 **17** **20** 29	7 9 17 **20** **29**
i=3	7 9 **1** **7** 20 29	7 9 **17** 20 29	7 9 **17** **20** **29**
i=4	7 **9** 17 20 29	7 **9** 17 20 29	**7** **9** **17** **20** **29**

5- Recherche dans les tableaux

5.1 Recherche séquentielle

La recherche séquentielle consiste à parcourir tous les éléments du tableau et les comparer à la valeur de l'élément à chercher.

Exemple

La fonction *RechSeq* permet de retourner la position de la première occurrence d'une valeur **val** dans un tableau.

Algorithme

```
Fonction RechSeq(T:tab; N, val:entier):entier
      Var i, pos : entier
Début
      Pos  ← -1
      i ← 0
      Tant que(i ≤ N-1 et pos = -1) Faire
            Si (T[i] = val )alors
                  pos← i
            Si non
                  i ← i+1
            Fin Si
      Fin Tant que
      Retourner pos
Fin
```

5.2 Recherche Dichotomique

La recherche dichotomique est effectuée dans un tableau trié de la manière suivante :

- On compare l'élément cherché à celui qui se trouve au milieu du tableau ;
- Si l'élément cherché est plus petit que l'élément du milieu, on continue la recherche ; dans la première moitié du tableau sinon dans la seconde ;
- On recommence ce processus sur la moitié sélectionnée ;
- On s'arrête une fois que l'élément cherché est trouvé ou lorsque l'intervalle de recherche devient nul.

Exemple

- La fonction *RechDicho* permet de retourner la position de la première occurrence d'une valeur **val** dans un tableau.

Algorithme

```
Fonction RechDicho (T : tab; N, val :entier):entier (version itérative)
      VAR i, pos, mil, inf, sup:entier
Début
      pos ← -1
      inf← 0
      sup← N-1
      Tant que (inf ≤ sup et pos = -1)Faire
            mil ← (inf + sup) div 2
            Si (T[mil] = val) alors
                  pos = mil
            Sinon
                  Si (val<T[mil]) alors
                        Sup← mil - 1
                  Sinon
                        inf← mil + 1
                  Fin Si
            Fin Si
      Fin Tant que
      Retourner pos
Fin
```

- Exécuter l'algorithme *RechDicho* pour rechercher l'indice de la valeur **37** dans le tableau ci-dessous :

Indice	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Tab	3	7	11	18	21	32	37	41	53	65

Mil = (inf+sup)/2	Tableau divisé en deux Tab[mil] en bleu élément cherché en rouge	Comparé Tab[mil] à 37	Tableau sélectionné
Mil=(0+9)/2 **Mil =4**	0 1 2 3 4 3 7 11 18 **21** 5 6 7 8 9 32 **37** 41 53 65	**21 < 37**	5 6 7 8 9 **32 37 41 53 65**
Mil=(5+9)/2 **Mil = 7**	5 6 7 32 **37** **41** 8 9 53 65	**41 > 37**	5 6 **32 37**
Mil=(5+6)/2 **Mil =5**	5 **32** 6 **37**	**32 < 37**	6 **37**
Mil=(6+6)/2 **Mil = 6**	6 **3** **7**	**37 = 37**	L'élément cherché **37** se trouve à la **position 6**

II. Enoncé TP N° 03 : Le Tri des Tableaux

Exercice 1

1. Ecrire une fonction *Inserer_Trier* qui permet de remplir un tableau d'entiers de taille **N** trié en ordre croissant, en insérant chaque élément à sa place adéquate pendant la saisie.

Exemple

Donner la taille du tableau : N = **5**

a. 1er élément : 7 (inséré automatiquement dans la case 1 du tableau)

A	**7**				

b. 2ème élément : 4 (on le compare avec le 1er élément et on le met dans la case adéquate)

A	**4**	7			

c. 3ème élément : 5 (on le compare avec ses précédents et on cherche sa place)

A	4	**5**	7		

d. Et ainsi de suite …

2. Ecrire une fonction *Fusionner* qui permet de fusionner deux tableaux entiers **T1(N)** et **T2(M)** triés en ordre croissant dans un tableau **T3 (N+M)** (**T3** doit être trié pendant la fusion).

3. Donner le déroulement de de la fonction fusion pour les deux tableaux suivants :

T1	4	7	11	13	19

T2	3	5	9

Exercice 2

1. Ecrire en langage C les fonctions *Tri_Bulle, Tri_Selection* et *Tri_Insersion.*
2. Pour chaque fonction de la question 1, Afficher les tableaux intermédiaires et le temps d'exécution pour le tableau suivant :

20	9	29	17	3	7

Exercice 3

Le Tri Rapide aussi appelé Tri Binaire d'un tableau est basé sur les étapes suivantes :

1. Choisir une valeur dite *pivot* dans le tableau (pour simplification, la première valeur du tableau est choisie comme *pivot*) ;
2. Permuter les éléments de telle façon que :
   - toutes les valeurs plus petites que le *pivot* soit à sa gauche (sous-tableau à gauche),
   - toutes les valeurs plus grandes que le *pivot* soit à sa droite (sous-tableau à droite),
3. Répéter les étapes 1. et 2. tant que les permutations sont possibles.

Ecrire une fonction récursive *Tri_Rapide* qui respecte cet algorithme.

Exercice 4

1. Ecrire une fonction récursive *Rech_Dicho* qui permet de chercher par dichotomie un entier **X** dans le tableau.
2. Ecrire une fonction récursive *Rech_Trico* qui permet de chercher par tricotomie un entier **X** dans le tableau (c'est le même principe que la recherche par dichotomie sauf qu'au lieu de chercher la valeur dans un des deux sous-tableaux par rapport à la valeur du milieu, on effectue la recherche sur trois sous-tableaux en fonction de deux valeurs se trouvant aux indices correspondant au 1/3 et 2/3 de la taille du tableau)**.**

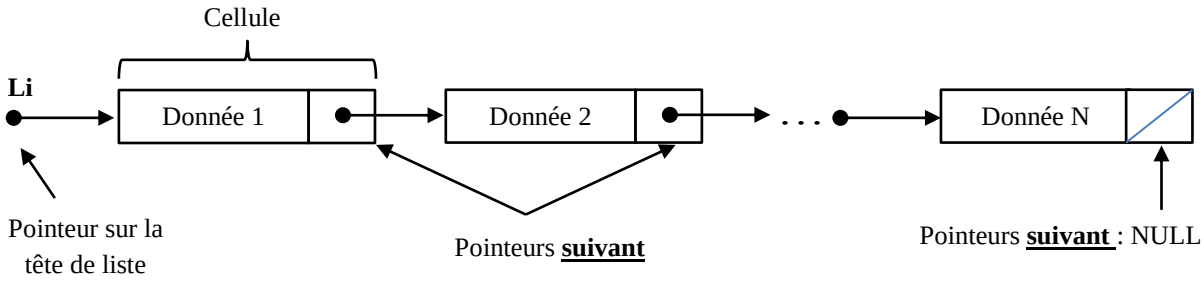
TP N°4 : LES LISTES CHAINEES

I. Rappels de Cours

1- Liste Simplement chaînée

Une liste chaînée est un ensemble de cellules liées entre elles par des pointeurs [Remy Malgouyres]. Chaque cellule est une structure contenant les champs suivants :

- une ou plusieurs données,
- un pointeur **suivant** sur la cellule suivante.



Le pointeur **L** permet d'accéder à la liste chaînée en pointant sur la première cellule. Puis on parcourt la liste d'une cellule à l'autre à travers les pointeurs **suivant** jusqu'à a fin de la liste chaînée désignée par le dernier pointeur **suivant** : NULL.

1.1 Déclaration d'une liste

Algorithme

```
TYPE Elément = enregistrement
   val : entier
   suivant: pointeur sur élément
Fin enregistrement

Cellule: pointeur sur Elément
VAR L : Cellule
```

Langage C

```
typedef struct element{

    int val;
    struct element *suivant;

}cellule;

cellule  *L
```

1.2 Initialisation d'une liste vide :

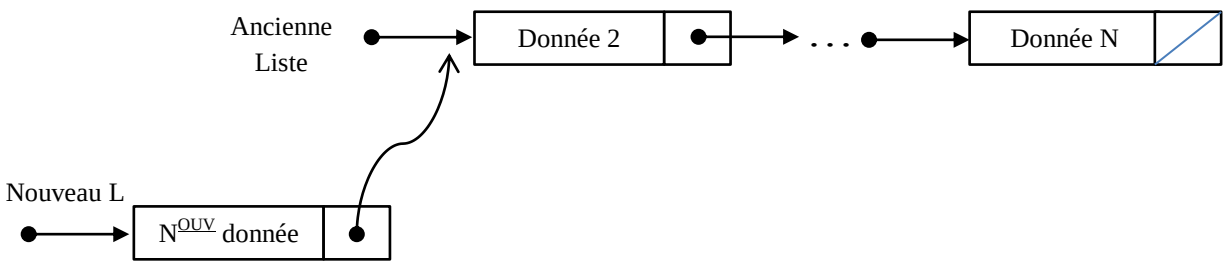
```
Fonction init_Liste (VAR L :
         Cellule): Cellule
Début
      L ← NULL
      Retourner (L)
Fin
```

```
cellule* init_Liste(cellule* L){

        L = NULL;
        return L;
}
```

1.3 Ajouter en tête de liste :

```
Fonction Ajout_en-tête(L :
Cellule, X : entier): Cellule

VAR newCellule : Cellule

Début
    newCellule ←
    Allouer Taille Cellule

   newCellule →val←X
   newCellule →suivant ←L

   Ajout_en-tête← newCellule
Fin
```

```
cellule* ajout_en_tete(cellule*
L,int X){

   cellule* newCellule=
   malloc(sizeof(cellule));

   newCellule->val=X;

   newCellule->suivant=L;

   return newCellule;

}
```



1.4 Ajouter en fin de liste

```
Fonction Ajout_Fin(L:Cellule,
         x : entier):Cellule

VAR newCellule : Cellule
    temp : Cellule

Début
   newCellule ←
   Allouer Taille Cellule

   newCellule →val←X
   newCellule →suivant ←NULL

   Si (L = NULL) alors
     Ajout_Fin ← newCellule

   Si non
    temp ← L

    Tant que(temp →suivant≠NULL)
     faire
         temp ← temp → suivant
    Fin Tant que

    temp→suivant ← newCellule
   Fin Si
   Ajout_fin←L
Fin
```

```
cellule* ajout_Fin(cellule*
L,int X){

   cellule* newCellule =
   malloc(sizeof(cellule));

   newCellule ->val=X;

   newCellule ->suivant=NULL;

   if(L==NULL)
       return newCellule;

   else{

      cellule* temp=L;

      while(temp->suivant!=NULL)
           temp=temp->suivant;

      temp->suivant= newCellule;

      return L;

    }

}
```

1.5 Suppression en tête de liste

```
Fonction Supprimer_en_tête(L:
         Cellule): Cellule

VAR  newEnTete: Cellule

Début

   Si (L≠NULL)
      newEnTete ← L→suivant
      LIBERER (L)
      Supprimer_en_tête←
      newEnTete

   sinon
      Supprimer_en_tête← NULL
   Fin si
Fin
```

```
cellule* supprimer_en_tete
(cellule* L){

  if(L!=NULL){

        cellule* newEnTete=
        L->suivant;

        free(L);

        return newEnTete;

    }

    else

        return NULL;
}
```

1.6 Suppression en fin de liste

```
Fonction Supprimer_Fin (L: Cellule): Cellule

VAR  tmp, ptmp : Cellule

Début

 Si (L = NULL) alors
  Supprimer_Fin← NULL
 Fin si

 Si (L→suivant = NULL) alors
   LIBERER(L);
   Supprimer_Fin← NULL
 Fin si

 tmp ← L;
 ptmp ← L;

 Tant que (tmp → suivant ≠
            NULL) faire
    ptmp ← tmp;
    tmp ← tmp→suivant
 Fin Tant que

 ptmp →suivant ←NULL
 LIBERER (tmp)

 Supprimer_Fin←L

Fin
```

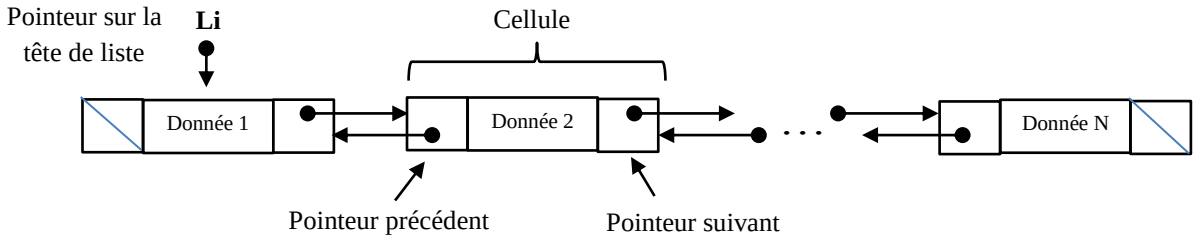
```
cellule* supprimer_Fin
(cellule* L){

   if(L== NULL)
       return NULL;

   if(L->suivant == NULL){

        free(L);
        return NULL;

    }

    cellule* tmp=L;
    cellule* ptmp=L;

    while(tmp->suivant!=NULL){

        ptmp=tmp;
        tmp=tmp->suivant;

    }

    ptmp->suivant=NULL;
    free(tmp);

    return L;

}
```

2- Liste doublement chainée

Contrairement aux listes simplement chaînées dans lesquelles le parcours ne peut se faire que dans un seul sens (du début –en tête- vers la fin), les listes doublement chaînées permettent un parcours aussi bien dans un sens que dans l'autre, ceux sont des listes bilatères. Ces listes comportent deux pointeurs :

- le pointeur « **suivant** » permet de pointer vers l'élément suivant,
- le pointeur « **precedent** » permet de pointer vers l'élément précédent.

Cette structure permet d'accélérer la recherche d'un élément.



2.1 Déclaration d'une structure liste doublement chainée

Algorithme

```
TYPE ElementListe =
     enregistrement
  val : entier
  prec: pointeur sur ElementListe
  suiv: pointeur sur ElementListe
Fin enregistrement

Elément=pointeur sur
        ElementListe

TYPE DListe= Enregistrement
 debut:pointeur sur ElementListe
 fin : pointeur sur ElementListe
Fin enregistrement

LISTE = pointeur sur DListe
VAR  L : LISTE
```

Langage C

```
typedef struct ElementListe{

   int    val;
   struct ElementListe *prec;
   struct ElementListe *suiv;

}Element;

typedef struct DList{

   struct ElementListe *debut;
   struct ElementListe *fin;

}liste;

liste* L;
```

2.2 Initialisation d'une liste

```
Fonction Initialiser_LISTE
(VAR L : LISTE) : LISTE
Début

   L ← Allouer Taille LISTE
   L → debut ← NULL
   L→ fin ← NULL
   Initialiser_LISTE ← L
Fin
```

```
liste* initialiser_Liste
       (liste* L){

    L=malloc(sizeof(liste));

    L->debut=NULL;

    L->fin=NULL;

  return L;

}
```

2.3 Tester liste vide

```
fonction estVide(L:LISTE): entier
Début
    Si(L→debut = NULL ) alors
      estVide←1
    Si Non
      estVide←0
    Fin Si
Fin
```

```
int estVide(liste* L){

    if(L->debut==NULL)
       return 1;

    else
        return 0;
}
```

2.4 Ajouter en tête de liste

```
Fonction ajout_en_tete
   (L:LISTE, X:entier) : LISTE

VAR newElement: Elément

Début
  newElement ← Allouer
               Taille Element
  newElement →val ← X
  newElement →préc ←NULL

  Si (estVide(L)=1) alors
    newElement →suiv ← NULL
    L → debut ← newElement
    L → fin ← newElement
  Sinon
    newElement→suiv←L→debut
    L→debut→prec ←newElement
    L → debut ← newElement
  Fin Si

  ajout_en_tete ← L
Fin
```

```
liste* ajout_en_tete(liste* L,
int X)
{

  Element*newElement=
  malloc(sizeof(Element));

  newElement ->val=X;

  newElement ->prec=NULL;

  if (estVide(L)){

    newElement ->suiv=NULL;
    L->debut = newElement;
    L->fin   = newElement;
  }

  else {

    newElement ->suiv=L->debut;
    L->debut->prec = newElement;
    L->debut= newElement;
  }

  return L;
}
```

2.5 Ajouter en fin de liste

```
Fonction Ajout_Fin(L:LISTE, X:
            entier) : LISTE
VAR newElement: Elément
Début
  newElement ← Allouer
               Taille Element
  newElement →val ← X
  newElement →suiv ←NULL
  Si (estVide(L)=1) alors
    newElement →prec ← NULL
    L → debut ← newElement
    L → fin ← newElement
  Sinon
    newElement →prec ← L→
fin
    L→fin→suiv ← newElement
    L → fin ← newElement
  Fin Si
  ajout_Fin ← L
Fin
```

```
liste* ajout_Fin(liste* L,int X)
{

  Element* newElement=
  malloc(sizeof(Element));

  newElement ->val=X;
  newElement ->suiv=NULL;

  if (estVide(L)){

    newElement ->prec=NULL;
    L->debut = newElement;
    L->fin = newElement;
  }

  else {

    newElement ->prec=L->fin;
    L->fin->suiv = newElement;
    L->fin= newElement;
  }

  return L;
}
```

2.6 Supprimer en tête de liste

```
fonction Supprimer_en_tete
(L: LISTE) : LISTE
VAR  suppElement : Elément
Début
 Si ( estVide(L)=0) alors
  Si(L→debut = L→fin)alors
    LIBERER (L)
    Supprimer_en_tete←
    Initialiser_Liste(L)
  Si Non
    suppElement ←L→debut
   suppElement→suiv→prec ←NULL
    L→debut←suppElement →suiv
    LIBERER (suppElement)
  Fin Si
 Fin Si
 Supprimer_en_tete← L
FIN
```

```
liste* supprimer_en_tete
(liste* L)
{
if (!estVide(L)){

 if (L->debut==L->fin){

    free(L);
    return initialiser_Liste(L);
 }

 else {

  Element* suppElement=L->debut;
  suppElement->suiv->prec=NULL;
  L->debut=suppElement->suiv;
  free(suppElement);
 }

}

return L;

}
```

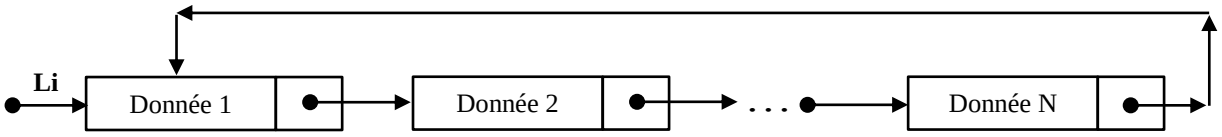
2.7 Supprimer en fin de liste

```
fonction Supprimer_Fin
(L: LISTE) : LISTE

VAR  suppElement : Elément
Début

 Si ( estVide(L)=0) alors
  Si(L→debut = L→fin)alors
     LIBERER (L)
     Supprimer_Fin←
     Initialiser_Liste(L)
   Si Non
     suppElement ←L→fin
    suppElement→prec→suiv ←
NULL
     L→fin←suppElement →prec
     LIBERER (suppElement)
  Fin Si
 Fin Si
 Supprimer_Fin← L
FIN
```

```
liste* supprimer_Fin(liste* L)
{

 if (!estVide(L)){

  if (L->debut==L->fin){

    free(L);
    return initialiser_Liste(L);
  }

  else {

   Element* suppElement=L->fin;
   suppElement->prec->suiv=NULL;
   L->fin=suppElement->prec;
   free(suppElement);
  }

 }

 return L;

}
```

3- Liste circulaire simplement chaînée

Une liste chainée circulaire est une liste dans laquelle le pointeur suivant du dernier élément pointe sur le premier élément de la liste. En arrivant au dernier élément, le déplacement recommencera au premier élément. Il s'agit donc d'une rotation.



3.1 Déclaration d'une structure liste chainée circulaire simple

```
TYPE ElémentListe =
     enregistrement
  val:  entier
  suiv: pointeur sur élément
Fin enregistrement
Elément= pointeur sur
         ElémentListe

TYPE ListeCir_ch =
     enregistrement
  début = pointeur sur élément
  fin = pointeur sur élément
Fin enregistrement

LISTE= pointeur sur ListeCir_ch
VAR  L : LISTE
```

```
typedef struct ElementListe{

    int val;
    struct ElementListe *suiv;

}Element;

typedef struct listeCir_ch{

     Element *debut;
     Element *fin;

}liste;
```

liste* L;

3.2 Initialisation d'une liste chainée circulaire simple

```
Fonction Initialiser_LISTE
(VAR L : LISTE) : LISTE

Début

   L ← Allouer Taille LISTE
   L→ debut ← NULL
   L→ fin ← NULL
   Initialiser_LISTE ← L
Fin
```

```
liste* initialiser_Liste(liste* L){

  L=malloc(sizeof(liste));

  L->debut=NULL;

  L->fin=NULL;

  return L;
}
```

3.3 Tester liste vide

```
fonction estVide(L:LISTE): entier
Début
     Si(L→debut = NULL ) alors
       estVide ← 1
     Si Non
       estVide ← 0
     Fin Si
Fin
```

```
int estVide(liste* L){

     if(L->debut==NULL)
        return 1;

     else
          return 0;
}
```

3.4 Ajouter en tête de liste

```
Fonction ajout_en_tete (L:LISTE,
          X: entier) : LISTE

VAR newElement: Elément

Début
  newElement ← Allouer Taille
               Element
  newElement →val ← X

  Si (estVide(L)=1) alors
     newElement→suiv←
        newElement
     L → debut ← newElement
     L → fin ← newElement
  Sinon
    newElement →suiv ←
L →debut
    L→fin→suiv ← newElement
    L → debut ← newElement
  Fin Si
  ajout_en_tete ← L
Fin
```

```
liste* ajout_en_tete(liste*
L,int X){

 Element* newElement=
 malloc(sizeof(Element));

 newElement->val=X;

 if (estVide(L)){

  newElement->suiv = newElement;
  L->debut = newElement;
  L->fin   = newElement;
 }

 else{

  newElement->suiv=L->debut;
  L->fin->suiv = newElement;
  L->debut= newElement;
 }

return L;

}
```

3.5 Ajouter en fin de liste

```
Fonction ajout_Fin (L:LISTE,
          X: entier) : LISTE
VAR newElement: Elément
Début
 newElement ← Allouer Taille
              Element
 newElement →val ← X
 Si (estVide(L)=1) alors
   newElement →suiv ←
     newElement
   L → debut ← newElement
   L → fin ← newElement
 Sinon
  newElement→suiv ← L→
debut
   L→fin→suiv ← newElement
   L → fin ← newElement
 Fin Si
 ajout Fin ← L
```

```
liste* ajout_Fin(liste* L,int X)
{

 Element* newElement =
 malloc(sizeof(Element));

 newElement->val= X;

 if (estVide(L)){

  newElement->suiv = newElement;
  L->debut = newElement;
  L->fin   = newElement;
 }

 else {

  newElement->suiv=L->debut;
  L->fin->suiv = newElement;
  L->fin= newElement;
 }

 return L;

}
```

3.6 Supprimer en tête de liste

```
fonction Supprimer_en_tete
          (L: LISTE) : LISTE
VAR  suppElement : Elément
Début
 Si ( estVide(L)=0) alors
  Si(L→debut = L→fin)alors
    LIBERER (L)
    Supprimer_en_tete←
    Initialiser_Liste(L)
  Si Non
   suppElement ←L→debut
   L→debut ← suppElement →suiv
   L→fin→ suiv ← L → debut
   LIBERER (suppElement)
  Fin Si
 Fin Si
 Supprimer_en_tete← L
FIN
```

```
liste* supprimer_en_tete(liste*
L){

 if (!estVide(L)){

  if (L->debut==L->fin){

   free(L);
   return initialiser_Liste(L);
  }

  else {

   Element* supElement=L->debut;
   L->debut=supElement->suiv;
   L->fin->suiv=L->debut;
   free(supElement);
  }

 }

 return L;

}
```

3.7 Supprimer fin de liste

```
fonction Supprimer_Fin
(L: LISTE) : LISTE

VAR  supElement, tmp : Elément

Début
 Si ( estVide(L)=0) alors
  Si(L→debut = L→fin)alors
    LIBERER (L)
    Supprimer_en_tete←
    Initialiser_Liste(L)
 Si Non
  supElement ←L→fin
  tmp ←L→debut
  Tant que (tmp→suiv≠ supElement)
     tmp ←tmp→suiv
   Fin Tant que
   L→fin←tmp
   L →fin→suiv ← L→debut
   LIBERER (supElement)
  Fin Si
 Fin Si
 Supprimer_Fin← L
FIN
```

```
liste* supprimer_Fin(liste* L)
{
 if (!estVide(L)){

  if (L->debut==L->fin){

    free(L);
    return initialiser_Liste(L);
  }

  else {

    Element* supElement=L->fin;
    Element* tmp=L->debut;
    while(tmp->suiv!=supElement)
         tmp=tmp->suiv;
    L->fin=tmp;
    L->fin->suiv=L->debut;
    free(supElement);
  }

 }

 return L;
}
```

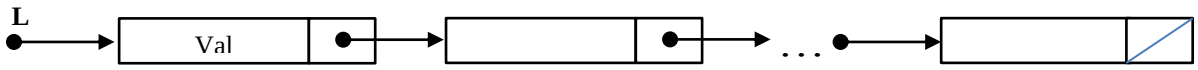
Conseils à prendre

- La fonction *malloc* permet de réserver un espace mémoire pour un type donné sans créer la liste chaînée.
- Pour des cas définis, il est nécessaire d'enregistrer la tête de liste afin d'éviter de perdre totalement la liste.
- L'utilisation de schéma, permet de faciliter la compréhension et la résolution du problème posé.

II. Enoncé TP N° 04 : Les Listes Chainées

Exercice 1

On dispose d'une liste chainée triée par ordre croissant sur le champ entier val :



1. Ecrire deux fonctions récursives *Croissant* et *Decroissant* pour l'affichage des champs val de la liste L, une dans l'ordre croissant et une autre dans l'ordre décroissant.
2. Ecrire une fonction *Inserer_Liste_Triee* qui permet d'insérer un élément dans une liste chaînée d'entier triée par ordre croissant. La liste retournée après l'insertion de l'élément doit garder le même tri.
3. Ecrire une fonction récursive *Verifier_Liste_Triee_Rec* et une autre itérative *Verifier_Liste_Triee_Ite* qui permettent de vérifier si la liste chainée est triée par ordre croissant.
4. Ecrire une fonction *Inverser_Liste* qui prend en argument une liste chaînée d'entiers triée par ordre croissant et qui inverse l'ordre de chaînage puis renvoie la liste chaînée triée dans l'ordre décroissant.
5. Ecrire une fonction *Fusionner_Liste* qui permet de fusionner deux listes triées dans une seule liste en gardant le même tri.

Exercice 2

On dispose d'une liste chainée d'entiers.

1. Ecrire une fonction *Dupliquer_Pair* qui permet de dupliquer les éléments pairs d'une liste chaînée d'entiers, et de garder les éléments impairs inchangés.

Exemple

Soit la liste < 4, 5, 8, 7, 12 >après l'exécution de la fonction la liste sera modifiée comme suit < 4, 4, 5, 8, 8, 7, 12, 12 >.

2. Une liste chainée **A** est dite incluse dans une liste chainée **B** si les éléments de **A** appartiennent à **B** dans le même ordre et successivement. Ecrire une fonction *Incluse_Liste* qui permet de vérifier si une liste **A** d'entiers est incluse dans une liste **B**.

3. Ecrire une fonction *Purger_Liste* qui permet de purger une liste (supprimer les doublons).
4. Ecrire une fonction *Intersection_Liste* qui permet de renvoyer l'intersection de deux listes **A** et **B**.
5. Ecrire une fonction *Listes_Identiques* qui permet de tester si deux listes chainées sont identiques.

Exercice 3

1. Ecrire une fonction *Convert_Liste_Simple_Double* qui permet de transformer une liste simplement chainée en une liste doublement chainée.
2. Ecrire une fonction *Tri_Croissant_ListeDB* qui permet de trier en ordre croissant une liste doublement chainée.
3. Ecrire une fonction *Inserer_ListeDB_Triee* pour insérer l'entier **X** dans une liste doublement chainée triée, de sorte que la liste restera triée.
4. Ecrire une fonction *Supprimer_Occ* qui permet de supprimer toutes les occurrences d'une valeur cherchée dans une liste doublement chainée d'entiers.

Exercice 4

On considère la liste des vainqueurs de la coupe des nations d'Afrique. On veut décrire une structure de données adéquate pour pouvoir répondre aux requêtes suivantes :

- Quelle est l'équipe qui a remporté la coupe l'année **X** ?
- Combien de fois une équipe **E** a remporté la coupe ?
- En quelle(s) année(s) une équipe **E** a remporté la coupe ?

1. Proposez une structure de données pour représenter ce problème.
2. Ecrire le programme principal qui fait appel aux fonctions précédentes.

Exercice 5

On veut créer une bibliothèque de plusieurs livres sous la forme d'une liste chainée.

On se donne la structure «livre» suivante:

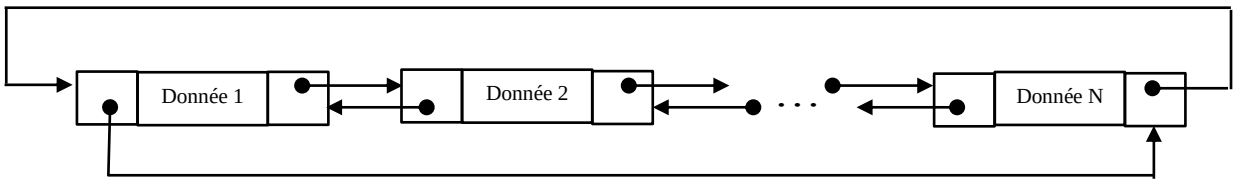
```
typedefstructlivre{
                    char  titre[20];
                      char  auteur[20];
                      char  editeur[20];
                      char  annee[10] ;
                    int    prix;
                    struct  livre  *suivant;
                    } livres;
```

1. Ecrire une fonction *Supprime* qui permet de supprimer un livre : par titre, par auteur ou par éditeur.
2. Ecrire une fonction *Chercher* qui permet de chercher un livre : par titre, par auteur ou par éditeur.
3. Ecrire une fonction *Afficher* qui permet d'afficher les livres : d'une année, d'un auteur ou d'un éditeur donnés.
4. Ecrire une fonction *Nombre* qui permet de calculer le nombre de livres : d'une année, d'un auteur ou d'un éditeur donnés.
5. Ecrire une fonction *Nombre_ex* qui permet de retourner le nombre d'exemplaires d'un livre donné.
6. Ecrire une fonction *Modifier* qui permet de modifier le prix d'un livre.
7. Ecrire une fonction *Prix* qui retourne le livre le moins cher et le livre le plus cher.
8. Ecrire une fonction *Valeur_Biblio* qui permet de calculer la somme des prix des livres de la bibliothèque.

Exercice 6

Liste circulaire doublement chaînées

La liste circulaire doublement chainée est une liste où chaque élément pointe sur l'élément suivant et sur l'élément précédent. Le parcours de cette liste se fait dans les deux sens.



Ecrire les différentes fonctions (*Déclaration*, *Création*, *Ajout* et *Suppression*) qui permettent la manipulation des listes circulaires doublement chaînées de données.

TP N°5 : LES PILES ET LES FILES

III. Rappels de Cours

1- Les piles

La Pile est une structure de données qui permet de stocker les données dans l'ordre **LIFO** (**L**ast **I**n **F**irst **O**ut = Dernier Entré Premier Sorti). L'ajout et la suppression des données se font en sommet de la pile.Le sommet de la pile est le dernier élément inséré.

1.1 Déclaration d'une structure Pile

```
TYPE Elément_Pile = enregistrement
   val: entier
   suiv: pointeur sur Elément_Pile
Fin enregistrement

Elément= pointeur sur Elément_Pile

TYPE pile = enregistrement
    sommet = pointeur sur élément
Fin enregistrement

Pile= pointeur sur pile
VAR  P : Pile
```

```
typedef struct Element_Pile {

        int val;
        struct Element_Pile *suiv;

  }Element;

typedef struct pile{

        Element *sommet;

  }Pile;

Pile * P ;
```

1.2 Initialisation d'une Pile

```
Fonction initialiser_Pile(VAR P
:Pile) : Pile
Début
      P ←  Allouer taille Pile
      P →sommet ← NULL
     initialiser_Pile ← P
Fin
```

```
Pile* initialiser_Pile(Pile* P){

        P=malloc(sizeof(Pile));

        P->sommet = NULL;

        return P;
}
```

1.3 Tester Pile Vide

```
fonction Pile_Vide(P:PILE): entier
Début
   Si ( P→sommet = NULL ) alors
            Pile_Vide ← 1
   Sinon
            Pile_Vide ← 0
    Fin Si
Fin
```

```
int Pile_Vide(Pile* P){
        if(P->sommet==NULL)
                return 1;
         else
                return 0;
}
```

1.4 Empiler

```
Fonction Empiler (P : Pile, X :
          entier) : Pile
VAR  newElement : Elément
Début
   newElement ←Allouer Taille
               Elément
   newElement → val ←X
   newElement →suiv ← P →sommet
   P → sommet ← newElement
   Empiler ←P
FIN
```

```
Pile* empiler(Pile* P,int X){
   Element* newElement=
   malloc(sizeof(Element));
   newElement->val=X;
   newElement->suiv=P->sommet;
   P->sommet=newElement;
   return P;
}
```

1.5 Dépiler

```
Fonction dépiler (P : Pile): Pile
VAR  supElement : Elément
Début
   si(Pile_Vide(P)=1) alors
      écrire('Pile est vide')
      dépiler ← P
   fin si

   supElement ←P → sommet
   P→sommet ←P→sommet→suiv
   LIBERER(supElement)
   dépiler←P
Fin
```

```
Pile* depiler(Pile* P){
   if (Pile_Vide(P)){
      printf(" Pile est vide.\n");
      return P;
   }
   Element* supElement=P->sommet;
   P->sommet=P->sommet->suiv;
   free(supElement);
   return P;
}
```

2.1 Sommet de la Pile

```
Fonction Sommet_Pile (P : Pile):
Entier
Début
   si(Pile_Vide(P)=1) alors
        Sommet_Pile ← -1
   fin si
   Sommet_Pile ← P→sommet→val
Fin
```

```
int Sommet_Pile(Pile* P){
        if (Pile_Vide(P))
                return -1;
        return
                P->sommet->val;
}
```

2- Les Files

La File est une structure de données qui permet de stocker les données dans l'ordre **FIFO** (**F**irst **I**n **F**irst **O**ut = Premier Entrée Premier Sorti). L'insertion des données se fait à la fin de la File, mais la suppression se fait au début de la File. La tête de la File (sommet) est le premier élément inséré.

2.1 Déclaration d'une structure File

```
TYPE Elément_File = enregistrement
    val: entier
    suiv: pointeur sur Elément_File
Fin enregistrement

Elément= pointeur sur Elément_File

TYPE file = enregistrement
   tete  = pointeur sur Elément
   queue = pointeur sur Elément
Fin enregistrement

File = pointeur sur file
VAR  F : File
```

```
typedef struct Element_File {

        int val;
        struct Element_File *suiv;

    }Element;

typedef struct file{

        Element *tete;
        Element *queue;

    }File;

File * F ;
```

2.2 Initialisation d'une File

```
Fonction iniitaliser_FILE
          ( VAR F :File) : File
Début
      F ← Allouer taille File
      F → tete ← NULL
      F → queue ← NULL
     initialiser_FILE ← F
Fin
```

```
File* initialiser_File(File* F){

    F=malloc(sizeof(File));

    F->tete   = NULL;

    F->queue  = NULL;

    return F;

}
```

2.3 Tester File Vide

```
fonction File_Vide(F: File):
         entier
Début
    Si ( F→tete = NULL ) alors
          File_Vide ← 1
    Sinon
          File_Vide ← 0
    Fin Si
Fin
```

```
int File_Vide(File* F){

  if(F->tete==NULL)

        return 1;
  else

        return 0;
}
```

2.4 Enfiler

```
Fonction Enfiler (F : File, X :
                  entier) : File
VAR  newElement : Elément
Début
   newElement←Allouer Taille
             Elément
   newElement → val ← X
   newElement → suiv ← NULL

   si(File_Vide(P)=1) alors
      F → tete ←  newElement
      F → queue ← newElement
   sinon
      F→queue→suiv ←newElement
      F→queue← newElement
   Fin si
   Enfiler ← F
FIN
```

```
File* Enfiler(File* F,int X){

    Element* newElement=
    malloc(sizeof(Element));

    newElement->val=X;
    newElement->suiv=NULL;

    if (File_Vide(F)){
        F->tete  = newElement;
        F->queue = newElement;
    }
    else {
        F->queue->suiv=newElement;
        F->queue=newElement;
    }

    return F;
}
```

2.5 Défiler

```
Fonction défiler (F : File): FILE
VAR  supElement:Elément
Début
   si(File_Vide(F)=1) alors
      écrire('File est vide')
      défiler ← F
   fin si

   supElement ←F → tete
   F→tete ←F→tete→suiv
   LIBERER(supElement)
   défiler←F
Fin
```

```
File* Defiler(File* F){
   if (File_Vide(F)){
        printf("File est vide.\n");
        return F;
   }
   Element* supElement = F->tete;
   F->tete =F->tete->suiv;
   free(supElement);
   return F;
}
```

2.6 Tete de la File

```
Fonction Tete_File (F : File):
Entier
Début
  si(File_Vide(F)=1) alors
    Tete_File ← -1
  fin si

  Tete_File ← F →tete→val
Fin
```

```
int Tete_File(File* F){
        if (File_Vide(F))
                return -1;
        return F->tete->val;
}
```

2.7 Queue de la File

```
Fonction Queue_File (F : File):
Entier
Début
  si(File_Vide(F)=1) alors
    Queue_File ← -1
  fin si

  Queue_File ← F→queue→val

Fin
```

```
int Queue_File(File* F){
        if (File_Vide(F))
                return -1;
        return F->queue->val;
}
```

IV. Enoncé TP N° 05 : Les Piles et Files

Exercice 1

L'objectif de cet exercice est d'utiliser une liste doublement chaînée pour implémenter une file d'éléments entiers. Pour cela, la structure d'une file à deux pointeurs est recommandée (**debut**: positionné sur le plus ancien élément de la file et **fin** : positionné sur le plus récent).

1. Proposer une structure de données pour présenter cette file.
2. Ecrire une fonction E*nfiler_Sans_Fin* pour enfiler un élément (sans utiliser le pointeur fin).
3. Ecrire une fonction E*nfiler_Avec_Fin* pour enfiler un élément (en utilisant le pointeur fin).
4. Ecrire une fonction *Défiler* pour retourner un élément.

Exercice 2

1. Soient deux files **F1** et **F2** et une pile **P**. La file **F1** contient des éléments entiers, **F2** et **P** sont vides. Ecrire une fonction nommée *Deplacer_File* qui dépose dans **F2** les éléments impairs de **F1** et laisse dans **F1** les éléments pairs dans l'ordre inverse.

   Exemple

   - **F1** a pour éléments [1,2,3,4,5,6] (1 est le 1er élément de **F1**).
   - Après l'exécution de la fonction *Deplacer_File* on aura : **F1** [6,4,2] et **F2** [1,3,5].

2. Soit deux files **F1** et **F2** d'entiers triées par ordre croissant, Ecrire une fonction *Fusionner_Files* qui permet de fusionner les deux files **F1** et **F2** afin d'obtenir une seule file **F** triée par ordre croissant.
3. Soient deux piles **P1** et **P2** et une file **F**. La pile **P1** contient des entiers, **P2** et **F** sont vides. Ecrire une fonction *Rotation_Pile* qui permet d'effectuer la rotation de **N** éléments de la pile **P1**. Cette fonction doit avoir comme arguments la pile **P1** et l'entier N et devra retournée la pile obtenue après rotation.

Exemple

- Si **P1** contient les éléments [2,6,9,8,12] (12 est le sommet de **P**1) et **N = 2**,
- Après l'exécution de la fonction *rotation* on aura : **P1** [8,12,2,6,9]

Exercice 3

L'objectif de cet exercice est d'écrire une fonction permettant l'évaluation d'une expression arithmétique en notation postfixée. Dans la notation postfixée, les opérateurs sont donnés après opérandes. Ainsi, l'expression $((A + B) * C)$ sera notée $AB + C *$. Cette notation permet de supprimer les parenthèses et les ambigüités liées aux règles de priorité des opérateurs.

Exemple

$\boldsymbol{F} = 3/(8 + 1 - 6) * (9 - 4) \quad \rightarrow \quad E = 381 + 6 -/ 94 -*$ la valeur de ***F*** est **5**.

1. En faisant appel aux fonctions de base d'une pile (Créer une pile vide, empiler un élément, dépiler un élément et retourner le sommet d'une pile), écrire une fonction *Post_Fixe* qui permet l'évaluation d'une expression mathématique saisie par l'utilisateur en se basant sur l'algorithme ci-dessous.

```
algorithme Expression
var     p : pile;
        exp : tableau de caractères;
        i, x, y : entiers;
Début
 P = pilevide();
 i = 0;
Lire(exp);  //Parcours de l'expression de gauche à droite
Tant que (i <Longueur(exp)) faire
        op = exp[i];
        si(op est un nombre) alors
        empiler(P, op);
        sinon
        x = depiler(P);
        y = depiler(P);
        z = x op y;
        empiler(P, z);
        Finsi
 i = i + 1;
 Fin Tant que
Ecrire(sommet(P));
Fin.
```

Exercice 4

L'objectif de cet exercice est d'écrire une fonction permettant d'effectuer un tri par insertion d'un ensemble d'entiers en utilisant les piles.

Soit une pile **A** d'éléments entiers. On désire construire une pile **B** triée contenant les éléments **A** ayant le minimum au sommet. En faisant appel aux fonctions de base d'une pile (Créer une pile vide, Empiler un élément, dépiler un élément, Retourner le sommet d'une pile, Tester si une pile est vide et Afficher les éléments d'une pile), écrire une fonction *Trier_Pile* qui utilise une pile intermédiaire **C** vide au début en se basant sur l'algorithme ci-dessous.

```
Algorithme Tri;
Var     A, B, C : pile;
        x : entier;
Début
 //Les piles B et C sont vide au début
Tant que (A n'est pas vide) faire
        Si (B est vide ou sommet(A) ≤ sommet(B)) alors
                x = depiler(A);
                empiler(B, x);
                 Tant que (C n'est pas vide) faire
                        x = depiler(C);
                        empiler(B, x);
                Fin Tant que
        Else
                x = depiler(B);
                empiler(C, x);
        Finsi
Fin Tant que
Fin
```

Remarque : Pour les exercices 2, 3 et 4, utiliser les fonctions de manipulation des piles et des files en représentation chaînée sans donner leurs codes correspondants

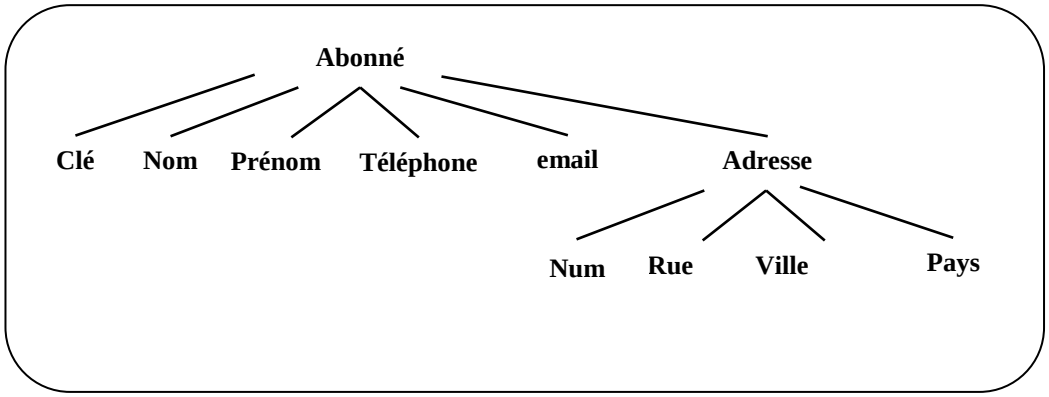
TP N° 06 : LES ARBRES

I. Rappels de Cours

1- Les Arbres

L'arbre est une structure de données composée d'un ensemble de nœuds. Ces nœuds contiennent les données à traiter et les pointeurs vers d'autres nœuds (d'autres sous-arbres)[**Livre** Michel Divay].

Exemple



L'arbre de la figure ci-dessus peut être représenté sous la forme suivante :

(Abonné (Clé, Nom, Prénom, Téléphone, email, Adresse (Num, Rue,Ville, Pays))).

Feuille : On dit qu'un nœud est une feuille quand il ne pointe vers aucun autre nœud (Clé, Nom, Prénom, Téléphone, email, Num, Rue,Ville, Pays).

Racine : On dit qu'un nœud est une racine quand il n'est pointépar aucun autre nœud (Abonné).

Niveau : Le niveau de la racine est 0. Tout nœud est d'augmenter de 1 par rapport au nœud qu'il dépend. (Adresse a le **niveau 1**, Pays a le **niveau 2**).

Hauteur (profondeur): C'est le niveau maximum de la branche la plus longue. La hauteur de Abonné est 2.

Degré d'un nœud : c'est le nombre de successeurs de ce nœud.

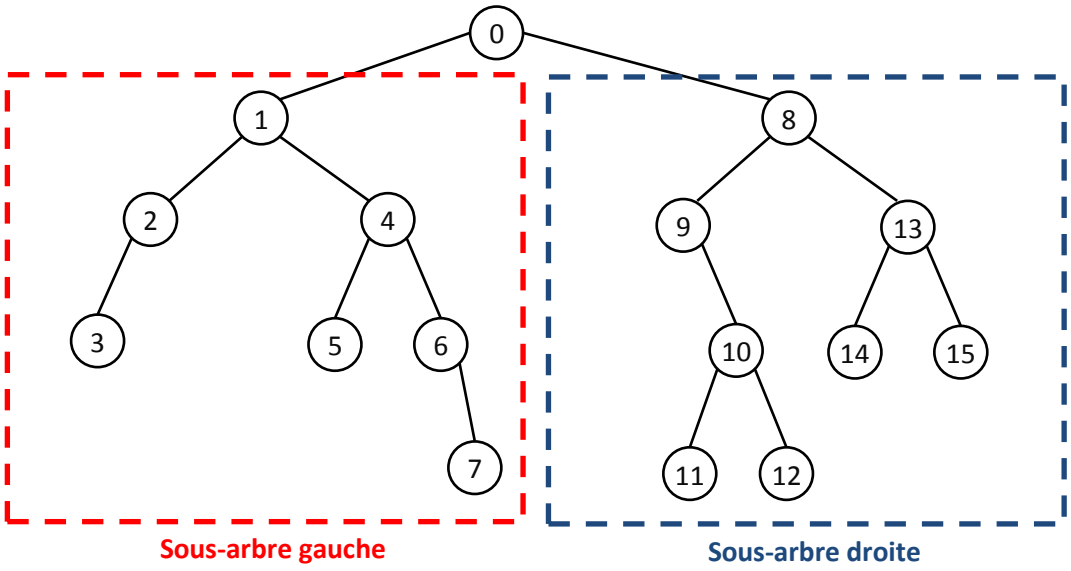
Degré d'un arbre:sur la figure ci-dessus, l'arbre a un degré égal à 6 (6 : est le degré maximum des nœuds de l'arbre). On dit qu'on a un arbre 6-aire.

Taille:C'est le nombre total des nœuds d'un arbre. La taille de l'arbre de la figure est 11.

2- Arbres Binaires

L'arbre binaire est un arbre ordonné tel que chaque nœuds a au plus 2 fils (fils gauche et fils droit. Si le nœud n'a qu'un seul fils, on distingue le fils gauche du fils droit**.**

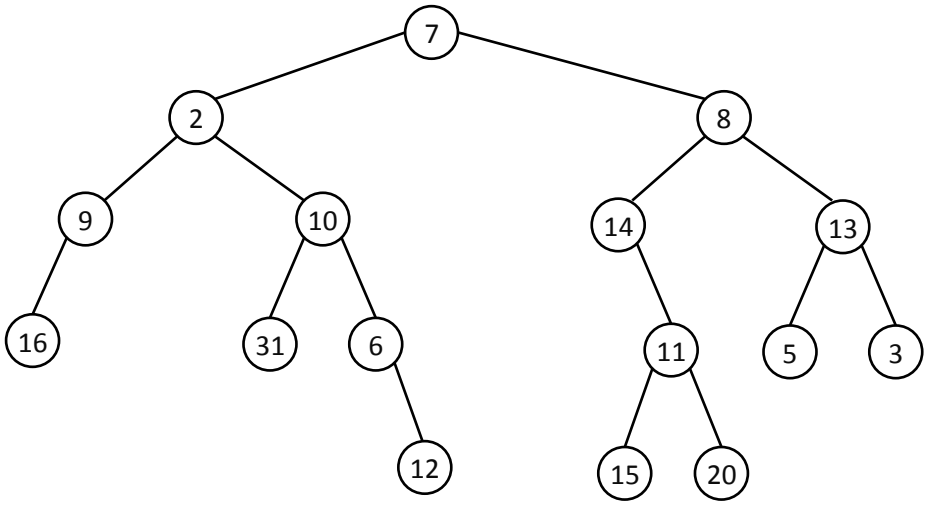
Exemple



II. Enoncé TP N° 06 : Les Arbres

Exercice 1

1. Donner un modèle de structure permettant de représenter un arbre binaire en représentation chaînée dont les éléments sont des entiers.
2. Ecrire une fonction *Creer_Noeud* qui permet de créer un nœud d'un arbre binaire. Cette fonction doit avoir comme arguments la valeur du nœud, la racine de son sous arbre gauche et la racine de son sous arbre droit et devra retourner le nœud créé.
3. Ecrire une fonction récursive *Nombre_Noeuds* qui permet de calculer le nombre de nœuds (la taille) d'un arbre binaire.
4. Ecrire les fonctions qui permettent de parcourir et d'afficher les nœuds d'un arbre binaire de manière : infixe, prefixe et postfixe.
5. Ecrire une fonction récursive *Afficher_Feuilles_Arbre* qui permet d'afficher les feuilles d'un arbre binaire.
6. Ecrire une fonction récursive *Hauteur_Arbre* qui permet de calculer la hauteur d'un arbre binaire. Cette fonction doit avoir comme argument la racine de l'arbre et devra retourner la hauteur obtenue.
7. Ecrire une fonction récursive *Nbr_Feuilles_Hauteur* qui permet de calculer le nombre de feuilles à une hauteur donnée dans un arbre, sachant que la hauteur de la racine est 0.
8. Dans le programme principal utiliser la fonction *Creer_Noeud* pour créer l'arbre binaire ci-dessous et proposer un menu pour tester les fonctions précédentes.



Exercice 2

1. On appelle Arbre Binaire de Recherche ABR un arbre binaire dans lequel les valeurs contenues dans le sous arbre gauche sont inférieures ou égales à la racine, et les valeurs contenues dans le sous arbre droit sont supérieurs à la racine. Ecrire une fonction *Arbre_ABR* qui teste si un arbre binaire donné en entrée est un Arbre Binaire de Recherche.
2. On appelle Arbre Binaire Parfait un arbre binaire dans lequel toutes les feuilles ont la même hauteur dans l'arbre. Ecrire une fonction récursive *Arbre_Parfait* qui permet de tester si un Arbre Binaire est Parfait. Utiliser la fonction *Hauteur_Arbre*.
3. On appelle Arbre Binaire Complet un arbre binaire tel que chaque nœud interne (non feuille) à exactement deux fils. Ecrire une fonction récursive *Arbre_Complet* qui permet de tester si un Arbre Binaire est Complet.
4. Un Arbre Binaire est dit Parfaitement Equilibré si pour chaque nœud de l'arbre, la différence entre le nombre de nœuds du sous arbre gauche et le nombre de nœuds du sous arbre droit est d'au plus un. Ecrire une fonction récursive *Arbre_Parfait_Equilibre* qui permet de vérifier si un Arbre Binaire est Parfaitement Equilibré en faisant appel à la fonction *Nombre_Noeuds*.
5. Un Arbre Binaire est dit AVL si pour chaque nœud de l'arbre, la différence entre la hauteur du sous arbre gauche et la hauteur du sous arbre droit est d'au plus un. Ecrire une fonction récursive *Arbre_AVL* qui permet de vérifier si un Arbre Binaire est AVL en faisant appel à la fonction *Hauteur_Arbre*.

Exercice 3

1. Ecrire une fonction *Ajouter_Noeud_ABR* qui permet d'ajouter un nœud dans un ABR. Cette fonction doit avoir comme arguments la racine de l'arbre et la valeur du nœud à ajouter.

2. En faisant appel à la fonction *Ajouter_Noeud_ABR*, écrire une fonction *Fusionner_ABR* qui permet de fusionner deux ABRs. Cette fonction doit avoir comme arguments les racines des deux ABRs et devra retourner l'ABR obtenu.

3. Ecrire deux fonctions m*in_ABR* et *Max_ABR* permettant de rechercher respectivement le minimum et le maximum d'un ABR. Ces fonctions doivent avoir comme argument un ABR et devront retourner respectivement la plus petite et la plus grande valeur contenue dans l'ABR.

4. Ecrire une fonction récursive *Rechercher_Noeud_ABR* qui permet de rechercher une valeur **V** dans un arbre binaire de recherche.

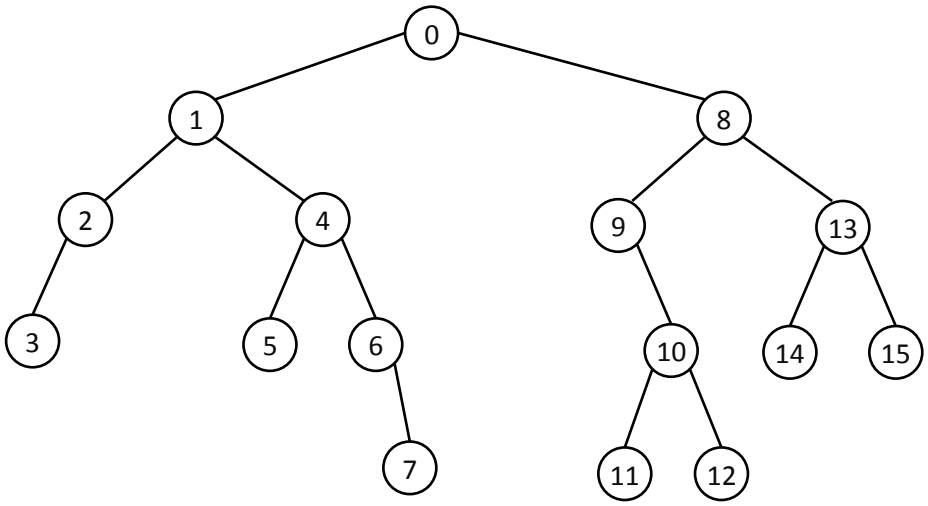
5. Ecrire une fonction *Valeur_Sup_ABR* qui permet de calculer la plus petite valeur supérieure à une valeur dans un ABR en faisant appel à la fonction *min_ABR*.

Exercice 4

A- Le parcours en largeur consiste à parcourir l'arbre niveau par niveau. Les nœuds de niveau 0 sont d'abord parcourus puis les nœuds de niveau 1 et ainsi de suite. Dans chaque niveau, les nœuds sont parcourus de la gauche vers la droite. Le parcours en largeur de l'arbre ci-dessous parcours les nœuds dans l'ordre [0,1,8,2,4,9,13,3,5,6,10,14,15,7,11,12].

Ecrire une fonction *Parcours_Largeur* qui permet de parcourir l'arbre binaire en largeur en se basant sur le principe suivant :

- si l'arbre n'est pas vide, enfiler la racine,
- pour chaque nœud défilé pour traitement, les fils gauche et droite sont enfilés successivement s'ils existent.
- ce processus est répéter (étape 2) tant que la file n'est pas vide.



B- Le parcours en pronfondeur préfixe consiste à parcourir l'arbre où chaque nœud est traité d'abord, puis ces fils gauche, ensuite ces fils droits s'ils existent. Le parcours en profondeur de l'arbre ci-dessous parcours les nœuds dans l'ordre [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15].

Ecrire une fonction *Parcours_Profondeur_Prefixe* qui permet de parcourir l'arbre binaire en profondeur en se basant sur le principe suivant :

- si l'arbre n'est pas vide, empiler la racine,

- pour chaque nœud dépilé pour traitement, les fils droit et gauche sont empilés successivement s'ils existent.
- ce processus est répéter (étape 2) tant que la pile n'est pas vide.

SOLUTIONS DES TPS

TP N° 01 : Les Fonctions

Exercice 1

```
void Supprimer_Doublons(int T[], int *n){
        int i,j,k;
        for(i=0;i<*n;i++){
            for(j=i+1;j<*n;j++){
                if (T[i]==T[j]){
                   for (k=j;k<*n;k++)
                        T[k]=T[k+1]; /*supprimer le doublon par
                                       Decalage */
                        j--;     /*repositionner j sur la
                                   position precedente pour
                                   eliminer les doublons
                                   successifs */
                        (*n)--; /*reduire la taille du tableau*/
                }
            }
        }
}
```

Exercice 2

1.

```
        /*Version Récursive*/
int NB_Chiffres_REC(int n){
        if(n/10==0)
                return1;
        return 1 + NB_Chiffres_REC(n/10);
}
     /*Version Itérative*/
int NB_Chiffres(int n){
        int nb=1;
        while (n/10 !=0){
                nb++;
                n=n/10;
        }
        return nb;
}
```

2.

```
void Trier_Nombres(int t[],int n){
        int i, j, x;
        for(i =0; i < n-1; i++)
                for(j = i +1; j < n; j++)
                   if(NB_Chiffres(t[i]) > NB_Chiffres(t[j])){
                                x= t[i];
                                t[i]= t[j];
                                t[j]= x;
                    }
}
```

Exercice 3

```
void Rotation_Tab(int t1[],int t2[],int n,int p){
        int i,j,x;
        if(p<0){
                for (i = 0; i < n; i++){
                        /*Calculer la novelle position j de
                          chaque élément de T1. Le reste de la
                          division est toujours inférieur à la
                          taille du tableau*/
                        j = (i + n + p) % n;
                        t2[j]=t1[i];
                }
        }
        else{
                for (i = 0; i < n; i++){
                        j = (i + p) % n;
                        t2[j]=t1[i];
                }
        }
}
```

Exercice 4

```
void Tasse_Tab(int t[],int *n){
   int i;
    for(i=0;i<(*n);i++){
         if(t[i]==0){
             int j=i;
             while(j<(*n-1)){
                t[j]=t[j+1];
                j++;
             }
             i--;     /* traiter le cas du 0 successif */
             (*n)--; /* réduire la taille du tableau */
         }
    }
}
```

Exercice 5

```
int Suite_Croissante(int t[],int n){
        int i,nb=1;
        for(i=0;i<n;i++){
                if(t[i]<=t[i+1])
                        printf("%d-",t[i]);
                else{
                        printf("%d \n",t[i]);
                        nb++;
                }
        }
        return nb;
}
```

Exercice 6

```
int rech_Element(int t[], int n, int x){
     int *p, res;
     res = 0;
     for(p = t; p < t + n; p++)
        if (*p == x){
                res = 1;
                break;
        }
     return res;
}
int rech_tab(int t1[], int t2[], int n, int m){
    int i, c;
    c = 0;
    for(i = 0; i < m; i++)
        if(rech_Element(t1, n, t2[i]) == 1)
                c = c + 1;
    return c ;
}
```

Exercice 7

```
int Carre_Magique(int n,int M[n][n]){
        int smL=0,smC=0,smD1=0,smD2=0,sP;
        int i,j;
        for ( i = 0;i<n;i++){
                smL=0;
                for ( j=0;j<n;j++)
                        smL+=M[i][j];
                        if (i==0)
                                sP=smL;
                        else{
                                if (sP!=smL)
                                        return 0;
                                else
                                        sP=smL;
                        }
        }
        for ( i = 0;i<n;i++){
                smC=0;
                for ( j=0;j<n;j++)
                        smC+=M[j][i];
                        if (i==0)
                                sP=smC;
                        else{
                                if (sP!=smC)
                                        return 0;
                                else
                                        sP=smC;
                        }
        }
        for( i=0 ;i<n ;i++){
                smD1+=M[i][i];
                smD2+=M[i][n-1-i];
        }
        if (smD1!=smL && smD1!=smC && smD1!=smD2)
                return 0;
        return 1;
}
```

TP N° 02 : La Récursivité

Exercice 1

```
/* Cette fonction vérifie si un caractère est déjà affiché */
int Sauter_Affichage(char ch[], int i, int j){
        if(j<i)
                if(ch[i]==ch[j])
                        return 0;
                else
                        return Sauter_Affichage(ch,i,j+1);
        else return 1;
}

void Nombre_Occ(char ch[], int i, int j, int c){
        if(i<strlen(ch)){
                if(j<strlen(ch)){
                     if(ch[i]==ch[j])
                              return Nombre_Occ (ch,i,j+1,c+1);
                     else
                              return Nombre_Occ (ch,i,j+1,c);
                }
                else{
                     if(Sauter_Affichage(ch,i,0))
                        printf("caractere <%c> = %d\n",ch[i],c);
                     return Nombre_Occ(ch,i+1,0,0);
                }
        }
}
```

Exercice 2

```
1- void Afficher_tab(int t[],int n){
        if(n>=0){
            Afficher(t,n-1);
            printf("%d",t[n]);
        }
   }
2-  int Somme(int t[],int n){
        if(n>=0)
            return t[n]+Somme(t,n-1);
        return 0;
   }
3-  int Occurrence(int t[],int n,int v){
        if(n<0)
              return 0;
        else{
              if(t[n]==v)
                   return 1 + Occurrence (t,n-1,v);
              return Occurrence (t,n-1,v);
        }
   }
```

```
4- int kiem_chiffre(int n,int k){
       if(k ==1)
           return n%10;
       return kiem_chiffre(n/10,k-1);
   }
5- void Classer(int T [ ] ,int N, int v){
       if (N<0)
           T[N+1]=v  ;
       else
           if ( T[N-1] > v ){
               T[N] = T[N-1] ;
               Classer( T,N-1,v) ;
           }
       else
         T[N]=v ;
   }
6- void Commun(int t1[],int n,int t2[],int m){
      if(n>=0 && m >=0){
          if(t1[n]==t2[m]){
              printf(" %d\t",t1[n]);
              return Commun (t1,n-1,t2,m-1);
          }
          if(t1[n]<t2[m])
              return Commun(t1,n,t2,m-1);
          else
              return Commun(t1,n-1,t2,m);
      }
   }
```

Exercice 3

```
/*La fonction max renvoie l'indice du plus grand élément du tableau*/
int max(int tab[],int taille){
        int i=0,indice_max=0;
        while(i <taille){
                if(tab[i]> tab[indice_max])
                indice_max= i;
                i++;
        }
        return indice_max;
}

void echanger(int tab[],int x,int y){
        int tmp;
        tmp= tab[x];
        tab[x]= tab[y];
        tab[y]=tmp;
}

void Trier(int tab[],int taille){
        if(taille <=1)
                return;
        /*on échange le dernier élément avec le plus grand*/
        echanger(tab, taille-1,max(tab, taille));
        return Trier(tab, taille-1);
}
```

Exercice 4

```
int Nbr_Commun(int T1[ ], int T2[ ], int N, int M){
        if((N<0)|| (M<0))
                return 0;
        if(T1[N]==T2[M]){
                printf ("%d \n", T1[N]);
                return 1+ Nbr_Commun( T1,T2,N-1,M-1);
        }
        else{
                if(T1[N]<T2[M])
                        return Nbr_Commun( T1,T2,N,M-1);
                else
                        return Nbr_Commun( T1,T2,N-1,M);
        }
}
```

Exercice 5

```
int Verifier_Redondant(int tab[], int i, int j, int n){
        if (i>n)
                return 1;
        if (j>n)
                return 0;
        if ((tab[i]==tab[j])&&(i!=j))
           /* traiter l'élément suivant (i+1) */
                Verifier (tab,i+1,0,n);
        else
           /* traiter l'élément courant (i) */
                Verifier (tab,i,j+1,n);
}
```

TP N° 03 : Le Tri des Tableaux

Exercice 1

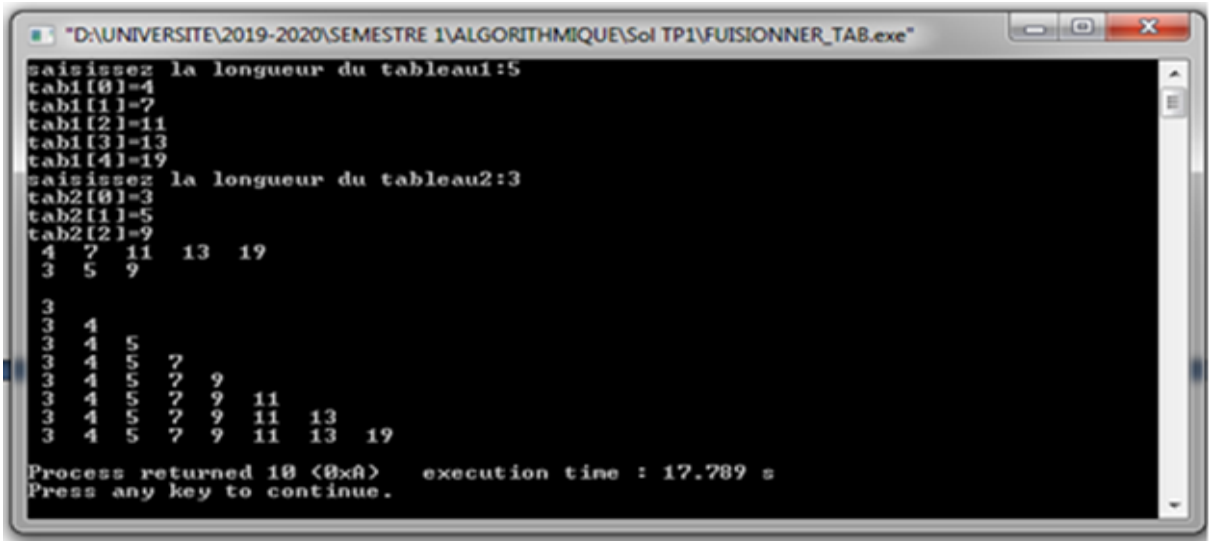
1.
```
void Affichage_tab(int t[], int n){
        int x;
        for(x=0;x<n;x++)
        printf(" %d ",t[x]) ;
        printf("\n");
}
void Inserer_Trier(int tab[], int n){
        int l,i,j,k,temp,nb;
        nb = 0;
        for(i = 0; i < n ; i++){
                printf("tab[%d]=  ",i);
                scanf("%d",&tab[i]) ;
                nb++;
                for (k = 1; k < nb; k++){
                        j=k;
                        temp=tab[k];
                        while ((j>0)&&(tab[j-1]>temp)){
                                /*faire le décalage des éléments*/
                                tab[j] = tab[j-1];
                                j--;
                        }
                        tab[j] =temp;
                }
                Affichage_tab(tab,nb);
        }
}
```

2.
```
void Fusionner(int T1[],int T2[],int T3[],int n, int m){
        int i=0,j=0,r=0,k=0,l=n+m;
        while ((j<n) && (k<m)){
                if(T1[j]<T2[k]){
                        T3[r]=T1[j];
                        r++;
                        j++;
                }
                else{
                        T3[r]=T2[k];
                        k++;
                        r++;
                }
        }
        /*s'il reste des éléments dans T1 seulement*/
        while (j<n){
                T3[r]=T1[j];
                r++;
                j++;
        }
        /*s'il reste des éléments dans T2 seulement*/
        while (k<m){
                T3[r]=T2[k];
                r++;
                k++;
        }
}
```

3- Le déroulement de la fonction fusion pour les deux tableaux suivants :

T1	4	7	11	13	19

T2	3	5	9



Exercice 2

```c
#include <time.h>
void Afficher_tab(int t[],int n){
        int i;
        printf("\n< ");
        for(i=0;i<n;i++)
                printf("%d - ",t[i]);
        printf(" >\n");
}

void Tri_Bulle(int t[],int n){
        int i,j,temp;
        long int temp1,temp2;
        printf("\nTableau Avant le TRI à BULLE : \n");
        Afficher_tab(t,n);
        printf("\n=========Debut du Traitement==========\n");
        temp1=clock();
        for (i=n-1;i>=1;i--){
        for (j=0;j<=i-1;j++){
                        if (t[j]>t[j+1]){
                        temp = t[j];
                        t[j]=t[j+1];
                        t[j+1]=temp;
                        Afficher_tab(t,n);
                        }
        }
        }
        temp2=clock();
        printf("\n=========Fin du Traitement==========\n");
        printf("\nTableau Apres le TRI à BULLE : \n");
        Afficher_tab(t,n);
        printf("\nLe Temps d'Execution du TRI à BULLE: %ld
        ms\n",temp2-temp1);
}
```

```
void Tri_Selection(int t[],int n){

        int i,j,pg,temp;
        long int temp1,temp2;
        printf("\nTableau Avant le TRI par SELECTION : \n");
        Afficher_tab(t,n);

        printf("\n=========Debut du Traitement==========\n");
        temp1=clock();
        i=n-1;
        while(i>0){
                pg=0;
                for(j=0;j<=i;j++){
                        if(t[j]>t[pg])
                        pg=j;
                }
                temp=t[pg];
                t[pg]=t[i];
                t[i]=temp;
                i--;
                Afficher_tab(t,n);
        }
        temp2=clock();
        printf("\n===========Fin du Traitement==========\n");

        printf("\nTableau Apres le TRI par SELECTION : \n");
        Afficher_tab(t,n);
        printf("\nLe Temps d'Execution du Tri par SELECTION :
        %ld ms \n",temp2-temp1);
}


void Tri_Insersion(int t[],int n){

        int i,j,val;
        long int temp1,temp2;
        printf("\nTableau Avant le TRI par INSERTION : \n");
        Afficher_tab(t,n);

        printf("\n==========Debut du Traitement=========\n");
        temp1=clock();
        for(i=1;i<=n-1;i++){
                val=t[i];
                j=i;
                while(j>0 && (t[j-1]>val) ){
                        t[j]=t[j-1];
                        j=j-1;
                }
                t[j]=val;
                Afficher_tab(t,n);
        }
        temp2=clock();
        printf("\n==========Fin du Traitement==========\n");

        printf("\nTableau Apres le TRI par INSERTION : \n");
        Afficher_tab(t,n);
        printf("\nLe Temps d'Execution du TRI par INSERTION :
        %ld ms \n",temp2-temp1);
}
```

4. déroulement d'exécution de chaque fonction

```
Tableau Avant le TRI ó BULLE :
20 - 9 - 29 - 17 - 3 - 7 -
================Debut du Traitement==================
9 - 20 - 29 - 17 - 3 - 7 -
9 - 20 - 17 - 29 - 3 - 7 -
9 - 20 - 17 - 3 - 29 - 7 -
9 - 20 - 17 - 3 - 7 - 29 -
9 - 17 - 20 - 3 - 7 - 29 -
9 - 17 - 3 - 20 - 7 - 29 -
9 - 17 - 3 - 7 - 20 - 29 -
9 - 3 - 17 - 7 - 20 - 29 -
9 - 3 - 7 - 17 - 20 - 29 -
3 - 9 - 7 - 17 - 20 - 29 -
3 - 7 - 9 - 17 - 20 - 29 -
================Fin du Traitement==================
Tableau Apres le TRI ó BULLE :
3 - 7 - 9 - 17 - 20 - 29 -
Le Temps d'Execution du TRI ó BULLE : 16 ms
```

```
Tableau Avant le TRI par SELECTION :
20 - 9 - 29 - 17 - 3 - 7 -
================Debut du Traitement==================
20 - 9 - 7 - 17 - 3 - 29 -
3 - 9 - 7 - 17 - 20 - 29 -
3 - 9 - 7 - 17 - 20 - 29 -
3 - 7 - 9 - 17 - 20 - 29 -
3 - 7 - 9 - 17 - 20 - 29 -
================Fin du Traitement==================
Tableau Apres le TRI par SELECTION :
3 - 7 - 9 - 17 - 20 - 29 -
Le Temps d'Execution du Tri par SELECTION : 6 ms
```

```
Tableau Avant le TRI par INSERTION :
20 - 9 - 29 - 17 - 3 - 7 -
================Debut du Traitement=================
9 - 20 - 29 - 17 - 3 - 7 -
9 - 20 - 29 - 17 - 3 - 7 -
9 - 17 - 20 - 29 - 3 - 7 -
3 - 9 - 17 - 20 - 29 - 7 -
3 - 7 - 9 - 17 - 20 - 29 -
================Fin du Traitement==================
Tableau Apres le TRI par INSERTION :
3 - 7 - 9 - 17 - 20 - 29 -
Le Temps d'Execution du TRI par INSERTION : 2 ms
```

Exercice 3

```c
void echanger(int tableau[], int a, int b){
        int temp = tableau[a];
        tableau[a] = tableau[b];
        tableau[b] = temp;
}

int partitionner(int *tableau, int p, int r){
        int pivot = tableau[p], i = p-1, j = r+1;
        int temp;
        while (1) {
                do
                        j--;
                while (tableau[j] > pivot);
                do      i++;
                while (tableau[i] < pivot);
                if (i < j)
                        echanger(tableau,i,j);

                else
                        return j;
        }
}

void Tri_Rapide(int *tableau, int p, int r){
        int q;
        if (p < r) {
                q = partitionner(tableau, p, r);
                Tri_Rapide (tableau, p, q);
                Tri_Rapide (tableau, q+1, r);
        }
}
```

Exercice 4

1.

```
void Rech_Dicho( int tab[] , int X , int debut , int fin ){
        int milieu = (debut + fin) / 2 ;

        if ( debut == fin )
                printf(" le valeur '%d' n'existe pas !\n", X);
        else
                if ( tab [milieu] == X)
                      printf(" le valeur '%d' existe au
                      tab[%d] \n", X , milieu );
                else{
                     if( tab [milieu] < X )
    /* traiter le sous tableau droite par rapport au milieu */
                          Rech_Dicho(tab,X, milieu+1,fin);
                     else
    /* traiter le sous tableau gauche par rapport au milieu */
                          Rech_Dicho(tab,X,debut,milieu-1);
                }
}
```

2.

```
void Rech_Trico(int tab[], int X,int debut, int fin){

        if (debut > fin ){
                printf(" le valeur '%d' n'existe pas !\n", X);
                return;
        }

        int taille=(fin - debut) +1;
        int M1= debut + taille/3;
        int M2= debut + 2*(taille/3);

        if (tab[M1] == X){
           printf(" le valeur '%d' existe au tab[%d]\n",X,M1);
           return;
        }

        if (tab[M2] == X){
           printf(" le valeur '%d' existe au tab[%d]\n",X,M2);
           return;
        }

        else if (X < tab[M1])
                /* Rechercher dans le 1 tier de tableau */
                return Rech_Trico(tab,X,debut,M1-1);

             else if ((X>tab[M1])&&(X<tab[M2]))
                     /* Rechercher dans le 2 tier de tableau */
                         return Rech_Trico(tab,X,M1+1,M2-1);

                  else
                     /* Rechercher dans le 3 tier de tableau */
                         return Rech_Trico(tab,X,M2+1,fin);
}
```

TP N° 04 : Les Listes Chainées

Exercice 1

1.

```
typedef struct element{
        int val;
struct element *suiv;
}cellule;

void Croissant(cellule* L){
        if (L != NULL){
         /* l'affichage précède l'appel de la fonction */
                printf("%d  ",L->val);
                Croissant(L->suiv);
        }
}

void Decroisant(cellule* L){
        if (L != NULL){
                Decroisant(L->suiva);
       /* l'affichage après l'appel de la fonction */
                printf("%d  ",L->val);
        }
}
```

2.

```
cellule* Inserer_Liste_Triee(cellule* debut, int x){
        cellule * ptmp, * tmp, *nouv;
        if(debut == NULL){
                nouv = (cellule *)malloc(sizeof(cellule));
                nouv->val = x;
                nouv->suiv = NULL;
                return nouv;
        }
        if (debut->val >= x)
                return ajout_en_tete(debut,x);
        else{
             ptmp = debut;
             tmp = debut->suiv;
             while(tmp != NULL){
               if(tmp->val < x){
                      ptmp = tmp;
                     tmp = tmp->suiv;
                }
                else{
                     nouv = (cellule *)malloc(sizeof(cellule));
                    nouv->val = x;
                    ptmp->suiv = nouv;
                    nouv->suiv = tmp;
                    ptmp = nouv;
                    return debut;
                }
             }
             if (tmp==NULL)
                 return ajout_Fin(debut,x);
         }
}
```

3.

```
                    /* Version Récursive */
int Verifier_Liste_Triee_Rec(cellule* L){
        if (L==NULL)
                return 1;

        else if (L->suiv==NULL)
                return 1;

             else if (L->val <= L->suiv->val)
         /* Faire appel à la fonction avec l'élément suivant */
                        return Verifier_Liste_Triee_Rec(L-
>suiv);
                      else
                        return 0;
}

                    /* Version Itérative */
int Verifier_Liste_Triee_Ite(cellule* L){
        cellule* tmp, *ptmp;
        if (L==NULL)
                return 1;
        else if (L->suiv==NULL)
                return 1;
             else{
                tmp=L;
                ptmp=L->suiv;
                while (ptmp!=NULL){
                        if (tmp->val <= ptmp->val){
                                tmp = ptmp;
                                ptmp = ptmp->suiv;
                        }
                        else
                                return 0;
                }
                return 1;
             }
}
```

4.

```
cellule* Inverser_Liste(cellule* L){
        if(L==NULL)
                return NULL;
        else{
                cellule* tmp=L;
                cellule* ptmp=tmp;
                L=L->suiv;
                tmp->suiv=NULL;
                while(L!=NULL){
                        tmp=L;
                        L=L->suiv;
         /* Inverser le chainage entre 2 éléments successifs */
                        tmp->suiv=ptmp;
                        ptmp=tmp;
                }
                return tmp;
        }
```

5.

```
                    /* Version Itérative */
cellule* Fusionner_Liste(cellule* L1, cellule* L2){

        cellule* nv=NULL;
        cellule* temp=L1;
        cellule* temp1=L2;

        while(temp!=NULL && temp1!=NULL){
            if(temp->val < temp1->val){
                 nv=ajout_Fin(nv,temp->val);
                 temp=temp->suiv;
            }
            else {
                 nv=ajout_Fin(nv,temp1->val);
                 temp1=temp1->suiv;
            }
        }

        while(temp!=NULL){
            nv=ajout_Fin(nv,temp->val);
            temp=temp->suiv;
        }

        while(temp1!=NULL){
            nv=ajout_Fin(nv,temp1->val);
            temp1=temp1->suiv;
        }

        return nv;
}

                /* Version Récursive */
cellule* Fusionner_Liste_Rec(cellule* L1,cellule* L2){

        if(estVide(L1))
                return L2;

        if(estVide(L2))
                return L1;

        if(L1->val < L2->val){
                L1->suiv= Fusionner_Liste_Rec (L1->suiv,L2);
                return L1;
        }

        else{
                L2->suiv= Fusionner_Liste_Rec (L1,L2->suiv);
                return L2;
        }
}
```

Exercice 2

1.

```
cellule* Dupliquer_Pair(cellule * L){
        cellule * tmp, * nouv;
        tmp = L;
        while(tmp!=NULL){
                if(tmp->val % 2 == 0){
                    nouv = (cellule *)
                    malloc(sizeof(cellule));
                    nouv->val = tmp->val;
                    nouv->suiv = tmp->suiv;
                    tmp->suiv = nouv;
                    tmp = nouv->suiv;
                }
                else
                    tmp = tmp->suiv;
        }
        return L;
}
```

2.

```
int Incluse_Liste(cellule * A, cellule * B){
        cellule * tmp, * ptr;
        tmp = B;
        while(tmp!=NULL){
                ptr=A;
                while(ptr != NULL){
                        if(ptr->val == tmp->val){
                                ptr = ptr->suiv;
                                tmp = tmp->suiv;
                        }
                        else
                                break;
                }
                if(ptr == NULL)
                        return 1;
                else
                        if(tmp->val != A->val)
                                tmp = tmp->suiv;
        }
        return 0;
}
```

3.

```
cellule* Purger_Liste(cellule* L){
        if(L == NULL || L->suiv == NULL )
                return L;
        cellule* tmp1 = L;
        cellule* tmp2 =L;
        while (tmp1 != NULL){
                cellule* tmp3= tmp1->suiv;
                int V = tmp1->val;
                while(tmp3 !=NULL){
                        if(tmp3->val == V ){
                                tmp2 ->suiv = tmp3->suiv;
                                free(tmp3);
                         /* Eliminer les doublons successifs */
                                tmp3=tmp2 ->suiv;
                        }
                        else {
                                tmp2=tmp3 ;
                                tmp3 =  tmp3 -> suiv;
                        }
                }
                tmp1 = tmp1->suiv;
                tmp2 = tmp1;
        }
        return L;
}
```

4.

```
cellule* Intersection_Liste(cellule* L1, cellule* L2){
        if(L1 == NULL || L2 == NULL)
                return NULL;
        /* Supprimer les doublons de L1 */
        L1=Purger_Liste(L1);
        /* Supprimer les doublons de L2 */
        L2=Purger_Liste(L2);

        cellule* Bdebut=L2;
        cellule* resultat =NULL;
        while(L1!= NULL){
                int V = L1->val;
                while (L2 != NULL){
                    if(L2->val == V){
                        cellule* tmp = malloc(sizeof(cellule));
                        tmp->val = V;
                        tmp->suiv = resultat;
                        resultat = tmp;
                    }
                    L2 = L2->suiv;
                }
                L1 = L1->suiv;
                L2 = Bdebut;
        }
        return resultat;
}
```

5.

```
                    /* Version Itérative */
int Listes_Identiques( cellule* L1, cellule* L2){

        cellule * tmp1, *tmp2;
        tmp1 =L1 ;
        tmp2 =L2 ;

        while ((tmp1 != NULL) && (tmp2 != NULL) &&
                (tmp1->val==tmp2->val)){
                tmp1 =tmp1->suiv;
                tmp2 =tmp2->suiv;
        }

        if ((tmp1 == NULL) && (tmp2==NULL))
                return 1;
        else
                return 0;
}

                    /* Version Récursive */
int Listes_Identiques_Rec( cellule* L1, cellule* L2){

        if ((L1 == NULL) && (L2==NULL))
                return 1;

        else if ((L1 == NULL) || (L2==NULL))
                return 0;

             else if  (L1->val!=L2->val)
                        return 0;

                  else
                      return Identique_Rec(L1->suiv, L2-suiv);

}
```

Exercice 3

1.

```
typedef struct ElementListe{
                    int val;
                    struct ElementListe *prec;
                    struct ElementListe *suiv;
                  }Element;

typedef struct listedb_ch{
                   Element *debut;
                   Element *fin;
                  }liste;

liste* Convert_Liste_Simple_Double(liste* LDB,cellule* LSC){
        cellule* tmpSC=LSC;
         liste*   tmpDB=LDB;

        while(tmpSC!=NULL) {
        //ajouterFinListe(tmpDB,tmpSC->val);
                Element* newElement=malloc(sizeof(Element));
                newElement->val=tmpSC->val;
                newElement->suiv=NULL;
                if (estVide(tmpDB)){
                        newElement->prec=NULL;
                        tmpDB->debut = newElement;
                        tmpDB->fin = newElement;
                }
                else {
                        newElement->prec=tmpDB->fin;
                        tmpDB->fin->suiv = newElement;
                        tmpDB->fin= newElement;
                }
                tmpSC=tmpSC->suiv;
        }
        return tmpDB;
}
```

2.

```
liste* Tri_Croissant_ListeDB(liste* L){
        if(estVide(L)|| (L->debut->suiv== NULL))
                return L;
        Element* temp=L->debut;
        Element* ntemp=temp->suiv;
        int valeur;
        while(temp->suiv!=NULL){
                if(temp->val > ntemp->val){
                        valeur=temp->val;
                        temp->val=ntemp->val;
                        ntemp->val=valeur;
                }
                if(ntemp->suiv!=NULL)
                        ntemp=ntemp->suiv;
                else{
                        temp=temp->suiv;
                        ntemp=temp->suiv;
                }
        }
        return L;
}
```

3.

```
liste* Inserer_ListeDB_Triee(liste* L, int x){
        Element * ptmp, * tmp, *nouv;

           /* Insérer dans une liste vide */
        if(L->debut==NULL){
                nouv = (Element *)malloc(sizeof(Element));
                nouv->val = x;
                nouv->suiv = NULL;
                L->debut=nouv;
                L->fin=nouv;
                return L;
        }
          /* Insérer au début de la liste */
        if (L->debut->val >= x)
                return ajout_en_tete(L,x);

        else{
                ptmp = L->debut;
                tmp  = L->debut->suiv;
                while(tmp != NULL){
                        if(tmp->val < x){
                                ptmp = tmp;
                                tmp = tmp->suiv;
                        }
                        else{
                                nouv = (Element *)malloc
                                       (sizeof(Element));
                                nouv->val = x;
                                ptmp->suiv = nouv;
                                nouv->prec=ptmp;
                                nouv->suiv = tmp;
                                tmp->prec=nouv;
                                ptmp = nouv;
                                return L;
                        }
                }
               /* Insérer à la fin de la liste */
                if (tmp==NULL)
                        return ajout_Fin(L,x);
        }
}
```

4.

```
liste* Supprimer_Occ(liste *L, int valeur){

        Element *temp=L->debut;
        Element *tempP=L->debut;
        Element *tempF=L->fin;

        if (estVide(L))
                return L;

        /* La liste contient un seul élément égal à la valeur
                        supprimée */
        if ((temp==tempF)&& (temp->val==valeur)){
                free(L);
                return initialiserListe(L);
        }

        while(temp->suiv!=NULL){

                if(temp->val==valeur){

                        tempP->suiv=temp->suiv;
                 /* Supprimer le 1 élément début de la liste */
                        if (temp==L->debut){
                                tempP=temp->suiv;
                                L->debut=tempP;
                        }
                        else
                                temp->suiv->prec=tempP;
                        free(temp);
                        temp=tempP;

                        if (temp==tempF){
                                free(L);
                                return initialiserListe(L);
                        }
                }

                else{
                        tempP=temp;
                        temp=temp->suiv;
                }
        }

        /* Supprimer le deriner élément */
        if(temp->val==valeur){
                tempP->suiv=NULL;
                L->fin=tempP;
                free(temp);
        }
        return L;
}
```

Exercice 6

1.

```
typedef struct ElementListe{
                int val;
                struct ElementListe *prec;
                struct ElementListe *suiv;
}Element;

typedef struct listecirulairedb_ch{
                Element *debut;
                Element *fin;
}liste;
```

2.

```
liste* initialiser_Liste(liste* L){

        L=malloc(sizeof(liste));
        L->debut=NULL;
        L->fin=NULL;

        return L;
}
```

3.

```
int estVide(liste* L){
        if(L->debut==NULL)
                return 1;
        else
                return 0;
}
```

4.

```
liste* ajout_en_tete(liste* L,int valeur){

        Element* newElement=malloc(sizeof(Element));
        newElement->val=valeur;

        if (estVide(L)){
                newElement->prec=newElement;
                newElement->suiv=newElement;
                L->debut = newElement;
                L->fin = newElement;
        }

        else {
                newElement->prec=L->fin;
                newElement->suiv=L->debut;
                L->debut->prec = newElement;
                L->fin->suiv = newElement;
                L->debut= newElement;
        }

        return L;
}
```

5.

```
liste* ajout_Fin(liste* L,int valeur){

        Element* newElement=malloc(sizeof(Element));
        newElement->val=valeur;

        if (estVide(L)){
                newElement->prec=newElement;
        newElement->suiv=newElement;
        L->debut = newElement;
        L->fin = newElement;
        }

        else {
        newElement->prec=L->fin;
        newElement->suiv=L->debut;
        L->debut->prec = newElement;
        L->fin->suiv = newElement;
        L->fin= newElement;
        }

        return L;
}
```

6.

```
liste* supprimer_en_tete(liste* L){

        if (!estVide(L)){
        if (L->debut==L->fin){
                        free(L);
                        return initialiser_Liste(L);
        }
        else {
                        Element* supElement=L->debut;
                        supElement->suiv->prec=L->fin;
                        L->fin->suiv=supElement->suiv;
                        L->debut=supElement->suiv;
                        free(supElement);
        }
        }
        return L;
}
```

7.

```
liste* supprimer_Fin(liste* L){

        if (!estVide(L)){
        if (L->debut==L->fin) {
                        free(L);
                        return initialiserListe(L);
        }
        else {
                        Element* supElement=L->fin;
                        supElement->prec->suiv=L->debut;
                        L->debut->prec=supElement->prec;
                        L->fin=supElement->prec;
                        free(supElement);
        }
        }
        return L;
}
```

TP N° 05 : Les Piles et Files

Exercice 1

1.

```
typedef struct Element_File{
                int val;
                struct Element_File *prec;
                struct Element_File *suiv;
        }Element;

typedef struct file_DB{
                Element *debut;
                Element *fin;
        }File;

File* initialiserFile(File* F){
         F=malloc(sizeof(File));
         F->debut = NULL;
         F->fin  = NULL;
         return F;
}

int File_Vide(File* F){
        if(F->debut==NULL)
                return 1;
        else
                return 0;
}
```

2.

```
File* Enfiler_Sans_Fin(File* F,int X){

        Element* newElement=malloc(sizeof(Element));

        newElement->val= X;
        newElement->prec=NULL;
        newElement->suiv=NULL;

        if (File_Vide(F)) {
                F->debut = newElement;
                F->fin   = newElement;
        }
        else {
                Element* Temp = F->debut;

                /* Parcourir la file jusqu'à la fin */
                while (Temp->suiv!=NULL)
                        Temp=Temp->suiv;

                /* Ajouter l'élément à la fin de la file */
                Temp->suiv = newElement;
                newElement->prec=Temp;
                F->fin  = newElement;
        }
        return F;
}
```

3.

```
File* Enfiler_Avec_Fin(File* F,int X){

        Element* newElement=malloc(sizeof(Element));

        newElement->val=X;
        newElement->prec=F->fin;
        newElement->suiv=NULL;

        if (File_Vide(F)) {
                F->debut = newElement;
                F->fin   = newElement;
        }

        else {
              /*Pointer sur  la fin de la file et ajouter
                        l'élément*/
                F->fin->suiv = newElement;
                F->fin = newElement;
        }

        return F;
}
```

4.

```
File* Defiler(File* F){

        if (FileVide(F)){
                printf("Rien a DEFILER, la File est vide.\n");
                return F;
        }

        if (F->debut == F->fin){
                free(F);
                return initialiserFile(F);
        }

        Element* Temp = F->debut;
        Temp->suiv->prec=NULL;
        F->debut=Temp->suiv;
        free(Temp);

        return F;

}
```

Exercice 2

1.

```
File* Deplacer_File(File* F1){

        File* F2 = initialiser_File(F2);
        Pile* P  = initialiser_Pile(P);

        int x;

        while(!FileVide(F1)){
                x = TeteFile(F1);
                if(x %2 == 0)
                        P  = empiler(P, x);
                else
                        F2 = Enfiler(F2, x);
                Defiler(F1);
        }

        while(!PileVide(P)){
                x  = SommetPile(P);
                F1 = Enfiler(F1, x);
                P  = depiler(P);
        }
        return F2;
}
```

2.

```
File* Fusionner_Files(File* F1, File* F2){
        int x;
        File* F = initialiser_File(F);

        /* Tant que le deux files ne sont pas vide */
        while(!FileVide(F1) && !FileVide(F2)){
                if(TeteFile(F1) <= TeteFile(F2)){
                        x  = TeteFile(F1);
                        F1 = Defiler(F1);
                }
                else{
                        x  = TeteFile(F2);
                        F2 = Defiler(F2);
                }
                F = Enfiler(F, x);
        }

        /* Tant qu'il y a des éléments dans F1 seulement */
        while(!FileVide(F1)){
                x  = TeteFile(F1);
                F1 = Defiler(F1);
                F  = Enfiler(F, x);
        }

        /* Tant qu'il y a des éléments dans F2 seulement */
        while(!FileVide(F2)){
                x  = TeteFile(F2);
                F2 = Defiler(F2);
                F  = Enfiler(F, x);
        }
        return F;
}
```

3.

```
Pile* Rotation_Pile(Pile* P1, int n){

        Pile* P2 = initialiserPile(P2);
        File* F  = initialiserFile(F);

        int x,i;

        for(i = 0; i < n; i++){
                x  = SommetPile(P1);
                P1 = depiler(P1);
                P2 = empiler(P2, x);
        }

        while(!PileVide(P1)){
                x  = SommetPile(P1);
                P1 = depiler(P1);
                F  = Enfiler(F, x);
        }

        while(!PileVide(P2)){
                x  = SommetPile(P2);
                P2 = depiler(P2);
                P1 = empiler(P1, x);
        }

        while(!FileVide(F)){
                x  = TeteFile(F);
                F  = Defiler(F);
                P2 = empiler(P2, x);
        }

        while(!PileVide(P2)){
                x  = SommetPile(P2);
                P2 = depiler(P2);
                P1 = empiler(P1, x);
        }

        return P1;
}
```

Exercice 3

```
int Calcul_Exp(int a, int b, char op){
        if (op=='+') return a+b;
        if (op=='-') return b-a;
        if (op=='*') return a*b;
        if (op=='/') return b/a;
}

int ch_int(char c){
        if (c=='0') return 0;
        if (c=='1') return 1;
        if (c=='2') return 2;
        if (c=='3') return 3;
        if (c=='4') return 4;
        if (c=='5') return 5;
        if (c=='6') return 6;
        if (c=='7') return 7;
        if (c=='8') return 8;
        if (c=='9') return 9;
}

int Post_Fixe(char exp[]){

        Pile* P=initialiser_Pile(P);
        int i, x, y, z;
        i = 0;

        while (i<strlen(exp)){

                if (isdigit(exp[i]))
                        P = empiler(P,ch_int(exp[i]));

                else{
                        x = Sommet_Pile(P);
                        P = depiler(P);
                        y = Sommet_Pile(P);
                        P = depiler(P);
                        z = Calcul_Exp(x,y,exp[i]);
                        P = empiler(P, z);
                }

                i++;
        }

        return Sommet_Pile(P);
}
```

Exercice 4

```
Pile* Trier_Pile(Pile* A){

        Pile* B = initialiser_Pile(B);
        Pile* C = initialiser_Pile(C);
        int x;

        while (!(PileVide(A))){

           if((PileVide(B))||(SommetPile(A)<= SommetPile(B))){

           /* Déplacer les éléments de A vers B */
                x = SommetPile(A);
                A = depiler(A);
                B = empiler(B, x);

            /* Vider la pile C en déplaçant ses éléments vers
                     la pile B  */
                while (!(PileVide(C))){
                        x = SommetPile(C);
                        C = depiler(C);
                        B = empiler(B, x);
                }
           }

           else{

             /* Empiler le sommet de la pile B dans la pile C
                          */
                x = SommetPile(B);
                B = depiler(B);
                C = empiler(C, x);
           }
        }

        return B;
```

TP N° 06 : Les Arbres

Exercice 1

1.

```
typedef struct noeud{

        int valeur;
        struct noeud * gauche;
        struct noeud * droite;
}noeud;
```

2.

```
noeud* Creer_Noeud(int val, noeud* g, noeud* d){

        noeud* x;
        x = (noeud *) malloc(sizeof(noeud));
        x->valeur = val;
        x->gauche = g; /* Sous arbre gauche (S.A.G) */
        x->droite = d; /* Sous arbre droit (S.A.D) */
        return x;

}
```

3.

```
int Nombre_Noeuds(noeud* racine){

        if (racine == NULL)
                return 0;
        else

            /* Calculer les nœuds du S.A.G + les nœuds S.A.D */
                return 1 + Nombre_Noeuds(racine->gauche)
                         + Nombre_Noeuds (racine->droite);
}
```

4.

```
                /* PARCOURS PREFIXE */
/* on visite le nœud lui même, puis tous les nœuds de son sous
arbre gauche, puis tous les nœuds de son sous arbre droit.*/

void Prefixe(noeud* arbre){

        if (arbre != NULL){
                /* Afficher la valeur du nœud */
                printf ( "%d" , arbre->valeur) ;
                printf ( " , " ) ;

                /* Traiter le S.A.G */
                Prefixe(arbre->gauche) ;

                /* Traiter le S.A.D */
                Prefixe(arbre->droite) ;
        }
}
```

```
                        /* PARCOURS INFIXE */
/* on visite tous les nœuds du sous arbre gauche, puis le nœud
lui même, puis tous les nœuds du sous arbre droit.*/
  void Infixe(noeud* arbre){

      if (arbre != NULL){
                /* Traiter le S.A.G */
                Infixe(arbre->gauche) ;

                if (arbre->gauche != NULL)
                        printf( " , " ) ;

                /* Afficher la valeur du nœud */
                printf ( "%d" , arbre->valeur) ;

                if (arbre->droite != NULL)
                        printf ( " , " ) ;

                /* Traiter le S.A.D */
                Infixe(arbre->droite) ;
      }
  }
                        /* PARCOURS POSTFIXE */
/* on visite tous les nœuds du sous arbre gauche, puis tous les
nœuds du sous arbre droit, puis le nœud lui même.*/
  void Postfixe(noeud* arbre){

          if (arbre != NULL){
                /* Traiter le S.A.G */
                Postfixe(arbre->gauche) ;

                /* Traiter le S.A.D */
                Postfixe(arbre->droite) ;

                /* Afficher la valeur du nœud */
                printf ( "%d" , arbre->valeur) ;
                printf ( " , " ) ;
          }
  }
```

```
PARCOURS PREFIXE :
7 , 2 , 9 , 16 , 10 , 31 , 6 , 12 , 8 , 14 , 11 , 15 , 20 , 13 , 5 , 3 ,

PARCOURS INFIXE  :
16 , 9 , 2 , 31 , 10 , 6 , 12 , 7 , 14 , 15 , 11 , 20 , 8 , 5 , 13 , 3

PARCOURS POSTFIXE:
16 , 9 , 31 , 12 , 6 , 10 , 2 , 15 , 20 , 11 , 14 , 5 , 3 , 13 , 8 , 7 ,

1 - Creer_Noeud
2 - Nombres_Noeuds
3 - Afficher_Arbre
4 - Afficher_Feuille
5 - Hauteur_Arbre
6 - Nombre_Feuille
7 - Detruire_Arbre
0 - Quitter
        Choix :
```

5.

```
void Afficher_Feuilles_Arbre(noeud * racine){

        if(racine != NULL){

          if(racine->gauche == NULL && racine->droite == NULL)
                printf("%d\t", racine->valeur);

          else{
                Afficher_Feuilles_Arbre(racine->gauche);
                Afficher_Feuilles_Arbre(racine->droite);
          }

        }
}
```

6.

```
int MAX(int a, int b){

        if (a > b)
                return a;
        else
                return b;
}

int Hauteur_Arbre(noeud * racine){

        if (racine != NULL)

                return 1 + MAX(Hauteur_Arbre(racine->gauche),
                               Hauteur_Arbre(racine->droite));
        else
                return -1;
}
```

7.

```
int Nbr_Feuilles_Hauteur(noeud * r, int h, int i) {

        if(r != NULL){

           if(h == i)

                if(r->gauche == NULL && r->droite == NULL)
                   return 1;

                else
                   return 0;

           else
                return Nbr_Feuilles_Hauteur (r->gauche,h,i+1)
                      +Nbr_Feuilles_Hauteur (r->droite,h,i+1);
        }
        else
                return 0;
}
```

Exercice 2

1.

```
int Arbre_ABR(noeud* arbre , int *min , int *max) {

        *min = *max = arbre->valeur;

        if(arbre->gauche != NULL)
                if(!Arbre_ABR(arbre->gauche, min , max)||
                !( arbre->valeur>*max))
                        return 0 ;

        if (arbre->droite!= NULL)
                if(!Arbre_ABR(arbre->droite, min, max)||
                !(arbre->valeur<=*min ))
                        return 0 ;

        return 1 ;
}
```

2.

```
int Arbre_Parfait(noeud * racine){

        if(racine == NULL)
                return 1;
        else
                if((racine->gauche != NULL)&&
                   (racine->droite != NULL))
                        if(Hauteur_Arbre(racine->gauche) ==
                        Hauteur_Arbre(racine->droite)&&
                        Arbre_Parfait(racine->gauche)==1
                        && Arbre_Parfait(racine->droite)==1)
                                return 1;
                        else
                                return 0;
                else
                        if (racine->gauche == NULL)
                          return Arbre_Parfait(racine->droite);
                        else
                          return Arbre_Parfait(racine->gauche);
}
```

3.

```
int Arbre_Complet(noeud * racine){

        if (racine == NULL)
                return 1;
        else
                if ((racine->gauche == NULL) &&
                   (racine->droite == NULL))
                        return 1;
                else
                        if ((racine->gauche != NULL)&&
                           (racine->droite != NULL)&&
                            Arbre_Complet(racine->gauche)==1 &&
                            Arbre_Complet(racine->droite) == 1)
                                return 1;
                        else
                                return 0;
}
```

4.

```
int Arbre_Parfait_Equilibre(noeud * racine){

    if(racine != NULL)

        if(abs(Nombre_Noeuds(racine->gauche)-
           Nombre_Noeuds(racine->droite))<=1)

            if(Arbre_Parfait_Equilibre(racine->gauche) == 1
            && Arbre_Parfait_Equilibre(racine->droite) ==
1)
                    return 1;

            else
                    return 0;

        else
             return 0;

    else

        return 1;
}
```

5.

```
int Arbre_AVL(noeud * racine){

    if(racine != NULL)

        if(abs(Hauteur_Arbre(racine->gauche)-
           Hauteur_Arbre(racine->droite))<=1)

             if(Arbre_AVL(racine->gauche) == 1
                && Arbre_AVL(racine->droite) == 1)

                    return 1;
             else

                    return 0;
        else

             return 0;
    else

        return 1;
}
```

Exercice 3

1.

```
noeud* Ajouter_Noeud_ABR(noeud* racine, int x){

        if(racine == NULL){

                racine = (noeud *)malloc(sizeof(noeud));
                racine->valeur = x;
                racine->gauche = NULL;
                racine->droite = NULL;
        }

        else{
                if(x <= racine->valeur)
                        racine->gauche =
                        Ajouter_Noeud_ABR(racine->gauche, x);
                else
                        racine->droite =
                        Ajouter_Noeud_ABR(racine->droite, x);
        }

        return racine;
}
```

2.

```
noeud* Fusionner_ABR(noeud * A, noeud * B){

        if(B != NULL){
                /* Traiter le S.A.G */
                A = Fusionner_ABR(A, B->gauche);

               /*Ajouter le nœud de l'arbre B dans l'arbre A*/
                A = Ajouter_Noeud_ABR(A, B->valeur);

                /* Traiter le S.A.D */
                A = Fusionner_ABR(A, B->droite);
        }
        return A;
}
```

3.

```
int min_ABR(noeud * racine){

        if (racine != NULL)
                if (racine->gauche == NULL)
                        return racine->valeur;
                else
                        return min_ABR(racine->gauche);
}

int Max_ABR(noeud * racine){

        if (racine != NULL)
                if (racine->droite == NULL)
                        return racine->valeur;
                else
                        return Max_ABR(racine->droite);
}
```

4.

```
noeud*  Rechercher_Noeud_ABR(noeud * arbre , int x ) {

        if ( arbre == NULL)
                return NULL;

        if (x == arbre->valeur){
                printf("Le noeud cherche %d est trouve.\n",x);
                return arbre;
        }

        if ( x < arbre->valeur )
                /* on descend à gauche */
                return Rechercher_Noeud_ABR(arbre->gauche,x);
        else
                /* on descend à droite */
                return Rechercher_Noeud_ABR(arbre->droite,x);
}
```

5.

```
int Valeur_Sup_ABR(noeud * racine, int x) {

        if (racine != NULL){

            if(x < racine->valeur)
                return Valeur_Sup_ABR(racine->gauche, x);

            else

                if(x > racine->valeur)
                      return Valeur_Sup_ABR(racine->droite,x);

                else
                      return min_ABR(racine->droite);
        }
}
```

Exercice 4

A.

```
typedef struct noeud{
        int valeur;
        struct noeud * gauche;
        struct noeud * droite;
        }noeud;

typedef struct cellule{
        noeud* donnee;
        struct cellule* suivant;
        }Cellule;

typedef struct file{
        Cellule * tete;
        Cellule * queue;
        }File;


File Enfiler(File F, noeud * E){

        Cellule * New_Element;

        New_Element = (Cellule *)malloc(sizeof (Cellule));

        if (New_Element != NULL){
                New_Element->suivant = NULL;
                New_Element->donnee = E;
                if (F -> tete == NULL){
                        F -> tete  = New_Element;
                        F -> queue = New_Element;
                }
                else{
                        F -> queue ->suivant = New_Element;
                        F -> queue = New_Element;
                }
        }

        return F;
}

noeud * Defiler(File *F){

        Cellule* tmp;
        noeud*   SupElement;

        if (F->tete != NULL){
                SupElement = F->tete->donnee;
                tmp = F->tete->suivant;
                free(F->tete);
                F->tete = tmp;
                return SupElement;
        }
        else
                printf("La File est Vide.\n");
}
```

```
int File_Vide(File F){
        if (F -> tete == NULL)
                return 1;
        else
                return 0;
}

void Parcours_Largeur(noeud* Racine){
        File F;
        noeud* n;
        F -> tete = NULL;

        if (Racine != NULL){
                /* Enfiler la racine */
                F = Enfiler(F, Racine);

                /* Tant que la file n'est pas vide */
                while(!File_Vide(F)){
                        /* défiler le nœud */
                        n = Defiler(&F);
                        printf ("%d ", n->valeur);

                        if(n->gauche != NULL)
                                /* Enfiler le fils gauche */
                                F = Enfiler(F, n->gauche);

                        if(n->droite != NULL)
                                /* Enfiler le fils droit */
                                F = Enfiler(F, n->droite);
                }
        }
}
```

B.

```
typedef struct noeud{
        int valeur;
        struct noeud * gauche;
        struct noeud * droite;
        }noeud;

typedef struct pile{
        noeud*  donnee;
        struct  Pile* precedent;
        }Pile;
```

```
Pile* Empiler(Pile* P, noeud* donnee){

        Pile * New_Element;
        New_Element = (Pile *) malloc(sizeof(Pile));
        if (New_Element != NULL){

        New_Element->donnee = donnee;
        New_Element->precedent = P;
        return New_Element;
        }
}

noeud* Depiler(Pile** P){

        Pile*  temp;
        noeud* SupElement;

        if(P != NULL){

        SupElement = (*P)->donnee;
        temp = (*P)->precedent;
        free(*P);
        *P = temp;
        return SupElement;
        }
        else
        printf("La Pile est vide\n");
}

int Pile_Vide(Pile * P){
        if (P == NULL)
        return 1;
        else
        return 0;
}

void Parcours_Profondeur_Prefixe(noeud * Racine){
        Pile * P;
        noeud * n;
        P = NULL;

        if (Racine != NULL){
                /* Empiler la racine */
        P = Empiler(P, Racine);

        /* Tant que la pile n'est pas vide */
                while(!Pile_Vide(P)){

                        /* dépiler le noeud */
                        n = Depiler(&P);
                        printf ("%d ", n->valeur);

                        if(n->droite != NULL)
                                /* Empiler le fils droit */
                        P = Empiler(P, n->droite);

                        if(n->gauche != NULL)
                                /* Empiler le fils gauche */
                        P = Empiler(P, n->gauche);
        }
        }
}
```

REFERENCES BIBLIOGRAPHIQUES

1- Cormen, T. H. (2002). *Introduction à l'algorithmique: cours et exercices*. Paris : Dunod.

2- Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C., & Ronald L. Rivest. (2010). *Algorithmique: cours avec 957 exercices et 158 problèmes*. France : Dunod.

3- Malgouyres, R., Zrour, R., & Feschet, F. (2014). *Initiation à l'algorithmique et à la programmation en C - 3e éd.: Cours avec 129 exercices corrigés*. france : Dunod.

4- Courtin, J., & Kowarski, I. (1998). *Initiation à l'algorithme et aux structures de données*. france : Dunod.

5- Amade, B. (2019). *Introduction à la programmation*. france : MA Editions-Eska.

6- Maniez, D. (2019). *Apprendre à programmer en 10 semaines chrono*. france : Dunod.

7- Putier, S., & Rohaut, S. (2016). *Algorithmique - Techniques fondamentales de programmation - exemples en C# - (nombreux exercices corrigés) [BTS - DUT informatique]*. france : Editions ENI.

8- Delannoy, C. (2014). *Le guide complet du langage C*. france : Eyrolles.

9- Malgouyres, R., Zrour, R., & Feschet, F. (2007). *Initiation à l'algorithmique et aux structures de données en C: 118 exercices corrigés*. france : Dunod.

10- Braquelaire, A. (2005). *Méthodologie de la programmation en C: norme C 99 - API POSIX*. france : Dunod.

11- Warin, B. (2002). *L'algorithmique: votre passeport informatique pour la programmation*. france : Ellipses.