



REPUBLIQUE TUNISIENNE  
MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE  
LA RECHERCHE SCIENTIFIQUES ET TECHNOLOGIQUES



UNIVERSITE DE JENDOUBA  
FACULTE DES SCIENCES JURIDIQUES, ECONOMIQUES ET DE GESTION DE JENDOUBA

# **Fascicule de Travaux Dirigés**

## **Algorithmique et structures de données II**

**Adressé aux étudiants de 1<sup>ère</sup> année Licence Fondamentale en  
Informatique Appliquée à la Gestion**


**Equipe pédagogique :**

- **Riadh IMED FEREH (Maître de conférences en Informatique)**
- **Riadh BOUSLIMI (Technologue en Informatique)**

**Année Universitaire : 2008-2009**

# PREFACE

---

 e fascicule des travaux dirigés d'algorithmique et structures de données II est à l'intention des étudiants de la première année en Licence en Informatique Appliquée à la Gestion de la Faculté des Sciences Juridiques, Économique et de Gestion de Jendouba. Il aborde brièvement les thèmes les plus classiques et les plus utilisés en informatique : les enregistrements, les fichiers, la récursivité, les listes chaînées, les piles, les files et les arbres binaires de recherche.

Le fascicule comporte 5 TD avec leurs corrections qui sont réparties comme suit :

TD1 : Les enregistrements et les fichiers

TD2 : La récursivité

TD3 : Les listes chaînées

TD4 : Les piles et les files

TD5 : Les arbres binaires de recherche

Une fois que l'étudiant a obtenue une connaissance suffisante sur la manipulation des types simples, dans ce fascicule nous débiterons par un TD1 qui est consacré pour la manipulation des types complexes (les enregistrements), et sur les fichiers séquentiels. L'étudiant sera capable à la fin du TD1 à manipuler les fichiers.

Dans le TD2, nous traiterons les sous-programmes récursifs, nous allons voir de plus près le mécanisme de transformation d'un programme itérative en un programme récursif. L'étudiant dans ce TD doit savoir exécuter à la main en utilisant une pile.

Après avoir traité les programmes en version récursifs, le TD3 sera sur l'allocation dynamique après avoir vu au part avant l'allocation statique. Dans ce TD nous traiterons les listes chaines que ce soit simple, double ou circulaire. L'étudiant doit apprendre à créer une liste, la parcourir et enfin savoir comment supprimer un élément.

Le TD4 est une suite du précédent, vu qu'il s'agit d'une liste chaînée avec des stratégies d'accès, pour les piles, ils utilisent la stratégie (Last In First Out) et pour les files (First In First out). Nous définirons les différents sous-programmes qui seront utiles pour manipuler ces derniers.

A la fin nous entamerons le TD5 qui sera consacré pour la manipulation des arbres binaires de recherche. Nous allons traiter dans celui-là les différents algorithmes avancés : la rotation, la fusion, la vérification d'un arbre s'il est parfait, dégénéré,...

Enfin, nous espérons que le présent ouvrage aura le mérite d'être un bon support pédagogique pour l'enseignant et un document permettant une concrétisation expérimentale pour l'étudiant.

**Les auteurs**

**Riadh IMED Fareh**

**Riadh BOUSLIMI**

## **Table des matières**

<b>TD n° 1(Les enregistrements et les fichiers).....</b>	<b>3</b>
<b>Correction du TD n°1.....</b>	<b>5</b>
<b>TD n° 2(Récurtivité).....</b>	<b>12</b>
<b>Correction du TD n°2.....</b>	<b>14</b>
<b>TD n° 3 (Les Listes chaînées).....</b>	<b>20</b>
<b>Correction du TD n°3.....</b>	<b>22</b>
<b>TD n° 4(Les piles et les files).....</b>	<b>33</b>
<b>Correction du TD n°4.....</b>	<b>34</b>
<b>TD n° 5(Arbre binaire de recherche).....</b>	<b>37</b>
<b>Correction du TD n°5.....</b>	<b>40</b>
<b>BIBLIOGRAPHIE.....</b>	<b>53</b>



Faculté des Sciences Juridiques, Economiques  
et de Gestion de Jendouba

Année Universitaire : 2008/2009 – Semestre 2

Module : Algorithmique et structures de données II

Classe : 1<sup>ère</sup> année LFIAG

Chargé de cours : Riadh IMED FEREH

Chargé de TD : Riadh BOUSLIMI

## TD n° 1 (Les enregistrements et les fichiers)

### Objectifs

- Manipuler correctement des variables de type enregistrement.
- Comprendre les concepts de base relatifs aux fichiers.
- Manipuler des fichiers à organisation séquentielle.

### EXERCICE N°1

Créer un enregistrement nommé « **Etudiant** » qui est caractérisé par un *identifiant*, un *nom* et un *prénom*.

On vous demande de saisir 10 étudiants, les ranger dans un tableau puis les afficher.

### EXERCICE N°2

On reprend l'exercice précédent mais on rajoute en plus pour chaque étudiant ses deux notes. On vous demande de créer le nouvel enregistrement nommé « **Notes** » qui est caractérisé par **NoteCc** (Note de contrôle continu) et **NoteEx** (Note d'examen).

Modifier l'enregistrement « **Etudiant** » afin qu'elle puisse être en relation avec l'enregistrement « **Notes** ».

On vous demande de créer :

- Une procédure de saisi des étudiants ainsi leurs notes.
- Une procédure d'affiche des étudiants avec leurs notes.
- Une fonction qui renvoie l'étudiant qui a eu la meilleure note d'examen.
- Une fonction qui renvoie la moyenne générale de la classe.  
$$\text{Moyenne} = \text{noteCc} * 0.3 + \text{noteEx} * 0.7$$
- Afficher la meilleure note d'examen et la moyenne générale de la classe.

Écrire le programme principal faisant appel aux différents sous-programmes.

### EXERCICE N°3

On souhaite mémoriser des noms des personnes dans un fichier nommé « **pers.dat** ».  
On vous demande alors de créer les sous-programmes qui suivent :

- Une procédure de création du fichier qui contient les noms des personnes.
- Une procédure d'affichage des noms de personnes.
- Une fonction qui permet de chercher un nom passé en argument et qui renvoie vrai si ce dernier est existant et faux sinon.
- Une procédure qui copie les noms sans compter le nom passé en paramètre.

Écrire le programme principal faisant appel aux différents sous-programmes.

#### EXERCICE N°4

On souhaite mémoriser les étudiants de la faculté ainsi que leurs notes dans un fichier nommé « **fichetu.dat** ». Un étudiant est caractérisé par un *identifiant*, un *nom* et un *prénom*. Chaque étudiant aura deux notes : une note de contrôle contenu et une note d'examen.

##### Travail à faire :

1. Créer les enregistrements nécessaires pour élaborer ce programme.
2. Écrire une procédure permettant de saisir les notes associées à un étudiant donné en paramètre.
3. Écrire une procédure permettant de créer le fichier des étudiants.
4. Écrire une procédure qui permet de copier les étudiants qui ont eu une moyenne supérieure ou égale à 10 du fichier « **fichetu.dat** » dans un tableau des étudiants.
5. Écrire une procédure qui permet de trier un tableau d'étudiants dans l'ordre décroissant selon leurs moyennes.
6. Écrire une procédure qui permet de créer le fichier nommé « **res.dat** » qui contiendra les étudiants qui sont réussît, trié dans l'ordre décroissant.
7. Écrire une procédure qui permet d'afficher le contenu du fichier « **res.dat** ».
8. Écrire le programme principal qui fait appel aux différents sous-programmes.

## Correction du TD n°1

### EXERCICE N°1

#### **Algorithme GesEtud**

##### **Type**

```
Etudiant : Enregistrement
    Ident : Entier
    Nom : chaine[30]
    Prénom : chaine[20]
Fin Etudiant
```

```
TAB : Tableau de 10 Etudiant
```

##### **Var**

```
ET : TAB
n : Entier
```

#### **Procédure Remplissage(m : Entier ; var T : TAB)**

##### **Var**

```
i : Entier
```

##### **Début**

##### **Pour i de 1 à m faire**

```
    Ecrire("Etudiant n°",i," :")
    Ecrire("Identifiant : "),Lire(T[i].Ident)
    Ecrire("Nom : "),Lire(T[i].Nom)
    Ecrire("Prénom : "),Lire(T[i].Prénom)
```

##### **Fin Pour**

##### **Fin**

#### **Procédure Affichage(m : Entier ; var T : TAB)**

##### **Var**

```
i : Entier
```

##### **Début**

```
    Ecrire("Identifiant          Nom          Prénom : ")
    Ecrire("-----")
```

##### **Pour i de 1 à n faire**

```
    Ecrire(T[i].Ident," ",Lire(T[i].Nom," ",T[i].Prénom)
```

##### **Fin Pour**

##### **Fin**

#### **{Programme principal}**

##### **Début**

```
    n ← 10
    Remplissage(n,ET)
    Affichage(n,ET)
```

##### **Fin**

**Algorithme GesEtud****Type****Notes : Enregistrement**    **noteCc : Réel**    **noteEx : Réel****Fin Notes**

Etudiant : Enregistrement

Ident : Entier

Nom : chaîne[30]

Prénom : chaîne[20]

**Note : Notes**

Fin Etudiant

TAB : Tableau de 10 Etudiant

**Var**

ET : TAB

n : Entier

**Procédure SaisiNotes(var E : Etudiant)****Var**    **noteEntrer : Réel****Début**    **Répéter**

Ecrire("Note contrôle contenu : "), Lire(noteEntrer)

**Jusqu'à** noteEntrer ≥ 0 **ET** noteEntrer ≤ 20

E.Note.NoteCc ← noteEntrer

**Répéter**

Ecrire("Note examen : "), Lire(noteEntrer)

**Jusqu'à** noteEntrer ≥ 0 **ET** noteEntrer ≤ 20

E.Note.NoteEx ← noteEntrer

**Fin****Procédure Remplissage(m : Entier ; var T : TAB)****Var**

i : Entier

**Début**    **Pour** i de 1 à m **faire**

Ecrire("Etudiant n°", i, " :")

Ecrire("Identifiant : "), Lire(T[i].Ident)

Ecrire("Nom : "), Lire(T[i].Nom)

Ecrire("Prénom : "), Lire(T[i].Prénom)

**SaisiNotes(T[i])**    **Fin Pour****Fin**

```

Procédure AfficheNotes (E : Etudiant)
Début
    Ecrire("Note Contrôle Contenu      Note Examen ")
    Ecrire("-----")
    Ecrire(E.Note.NoteCc, "      ", E.Note.NoteEx)
Fin

Procédure Affichage (m : Entier ; T : TAB)
Var
    i : Entier
Début
    Ecrire("Identifiant      Nom      Prénom : ")
    Ecrire("-----")
    Pour i de 1 à n faire
        Ecrire(T[i].Ident, "  ", Lire(T[i].Nom, "  ", T[i].Prénom)
        AfficheNotes (T[i])
    Fin Pour
Fin

Fonction MeilleureNote (m : Entier ; T : TAB) : Réel
Var
    i : Entier
    NoteMax : Réel
Début
    NoteMax ← T[1].Note.NoteEx
    Pour i de 2 à m Faire
        Si T[i].Note.NoteEx > NoteMax Alors
            NoteMax ← T[i].Note.NoteEx
        Fin Si
    Fin Pour
    MeilleureNote ← NoteMax
Fin

Fonction MoyenneGénérale (m : Entier ; T : TAB) : Réel
Var
    i : Entier
    som : Réel
Début
    som ← 0
    Pour i de 2 à m Faire
        som ← som + 0.3 x T[i].Note.noteCc + 0.7 x T[i].Note.noteEx
    Fin Pour
    MoyenneGénérale ← som / m
Fin

{Programme principal}
Début
    n ← 10
    Remplissage (n, ET)
    Affichage (n, ET)
    Ecrire("Meilleur note examen :", MeilleureNote (n, ET),
    " Moyenne générale de la classe :", MoyenneGénérale (n, ET))
Fin

```



### EXERCICE N°3

#### **Algorithme TraitFichNom**

##### **Type**

Nom : chaîne[30]  
FichNoms : Fichier de Nom

##### **Var**

F1, F2 : FichNoms

##### **Procédure Création(Var fn : FichNoms)**

##### **Var**

n : Nom  
rep : caractère

##### **Début**

**Ouvrir**(fn, E) *//ouverture du fichier en écriture*

rep ← 'O'

**Tant que** MAJUS(rep) = 'O' **Faire**

    Ecrire("Nom : "), Lire(n)

    Ecrire(fn, n)

    Ecrire("Voulez-vous ajouter un autre nom (O/N) : ")

    Lire(rep)

**Fin Tant que**

**Fermer**(fn) *//fermeture du fichier*

##### **Fin**

##### **Procédure Affichage(fn : FichNoms)**

##### **var**

n : Nom

##### **Début**

**Ouvrir**(fn, L)

Lire(fn, n)

**Tant que** NON(FinDeFichier(fn)) **Faire**

    Ecrire(n)

    Lire(fn, n)

**Fin Tant que**

**Fermer**(fn)

##### **Fin**

##### **Fonction Recherche(x : Nom ; fn : FichNoms) : Booléen**

##### **var**

n : Nom

Trouve : Booléen

##### **Début**

**Ouvrir**(fn, L)

Lire(fn, n)

Trouve ← (n = x)

**Tant que** Trouve=faux **ET** NON(FinDeFichier(fn)) **Faire**

    Lire(fn, n)

    Trouve ← (n = x)

**Fin Tant que**

**Si** FinDeFichier(fn) **Alors**

    Recherche ← faux

**Sinon**

    Recherche ← vrai

**Fin Si**

**Fermer**(fn)

##### **Fin**

```

Procédure Copier(x : Nom ; fn : FichNoms ; var ft : FichNoms)
var
    n : Nom

Début
    Ouvrir(fn,L)
    Ouvrir(ft,E)
    //copie du fichier source vers le fichier de destination
    Lire(fn,n)
    Tant que n ≠ x ET NON(FinDeFichier(fn)) Faire
        Ecrire(ft,n)
        Lire(fn,n)
    Fin Tant que
    Si NON(FinDeFichier(fn)) Alors
        Lire(fn,n)
        //copie du reste du fichier
        Tant que NON(FinDeFichier(fn)) Faire
            Ecrire(ft,n)
            Lire(fn,n)
        Fin Tant que
    Fin Si
    Fermer(fn)
    Fermer(ft)
Fin

{Programme principal}
Début
    Création(F1)
    Affichage(F1)
    Si Recherche("Riadh",F1) Alors
        Ecrire("Riadh est existant dans le fichier")
    Sinon
        Ecrire("Riadh est non existant dans le fichier")
    Fin Si
    Copier("Riadh",F1,F2)
    Affichage(F2)
Fin

```

#### EXERCICE N°4

```

Algorithme GesEtudFichier
Type
    Notes : Enregistrement
        noteCc : Réel
        noteEx : Réel
    Fin Notes
    Etudiant : Enregistrement
        Ident : Entier
        Nom : chaine[30]
        Prénom : chaine[20]
        Note : Notes
    Fin Etudiant
    TAB : Tableau de 100 Etudiant
    FichEtud : Fichier de Etudiant

```

```

Var
    Fe,Fr : FichEtud

Procédure SaisiNotes(var E : Etudiant)
Var
    noteEntrer : Réel
Début
    Répéter
        Ecrire("Note contrôle contenu : "),Lire(noteEntrer)
    Jusqu'à noteEntrer ≥ 0 ET noteEntrer ≤ 20
    E.Note.NoteCc ← noteEnter
    Répéter
        Ecrire("Note examen : "), Lire(noteEntrer)
    Jusqu'à noteEntrer ≥ 0 ET noteEntrer ≤ 20
    E.Note.NoteEx ← noteEnter
Fin

Procédure Création(var fn : FichEtud )
Var
    Et : Etudiant
    rep : caractère
Début
    Ouvrir(fn,E) //ouverture du fichier en écriture
    rep ← 'O'
    Tant que MAJUS(rep) = 'O' Faire
        Ecrire("Identifiant : "),Lire(Et.Ident)
        Ecrire("Nom : "),Lire(Et.Nom)
        Ecrire("Prénom : "),Lire(Et.Prénom)
        SaisiNotes(Et)
        Ecrire(fn,Et)

        Ecrire("Voulez-vous ajouter un autre nom (O/N) : ")
        Lire(rep)
    Fin Tant que
    Fermer(fn) //fermeture du fichier
Fin

Procédure CopierDansTab(fn :FichEtud; var n:Entier ;var T : TAB )
var
    Et : Etudiant
    Moy : Réel
Début
    Ouvrir(fn,L)
    Lire(fn,Et)
    n ← 0
    Tant que NON(FinDeFichier(fn)) Faire
        Moy ← 0.3 x Et.Note.noteCc + 0.7 x Et.Note.noteEx
        Si Moy ≥ 10 Alors
            n ← n + 1 //incrémentatation de la taille
            T[n] ← Et //affectation de l'enregistrement au tableau
        Fin Si
        Lire(fn,Et)
    Fin Tant que
    Fermer(fn)
Fin

```

**Procédure TriBulle( n : Entier ; var T :TAB)**

**Var**

i : Entier  
aux : Etudiant  
rep : Booléen  
moy 1,moy2: Réel

**Début**

**Répéter**

rep ← faux  
**Pour** i **de** 1 **à** n **Faire**  
    moy1 ← 0.3 x T[i].Note.noteCc + 0.7 x T[i]  
    moy2 ← 0.3 x T[i+1].Note.noteCc + 0.7 x T[i+1]  
    **Si** moy1 < moy2 **Alors**  
        aux ← T[i]  
        T[i] ← T[i+1]  
        T[i+1] ← aux  
        rep ← vrai  
    **Fin Si**  
**Fin Pour**  
n ← n + 1  
**Jusqu'à** rep = faux **OU** n =1

**Fin**

**Procédure Résultat(fn :FichEtud; var fr : FichEtud)**

**Var**

i,n : Entier  
T : TAB

**Début**

**CopierDansTab**(fn,n,T)  
**TriBulle**(n,T) *//Tri dans l'ordre décroissant*  
**Ouvrir**(fr,E) *// Ouverture du fichier résultat en écriture*  
**Pour** i **de** 1 **à** n **Faire**  
    **Ecrire**(fr,T[i])  
**Fin Pour**  
**Fermer**(fr) *//fermeture du fichier*

**Fin**

**Procédure Affichage(fr : FichNoms)**

**var**

Et : Etudiant  
Moy : Réel

**Début**

**Ouvrir**(fr,L)  
**Lire**(fr,Et)  
**Tant que** **NON**(**FinDeFichier**(fr)) **Faire**  
    Moy ← 0.3 x Et.Note.noteCc + 0.7 x Et.Note.noteEx  
    **Ecrire**(Et.Ident, " ",Et.Nom, " ",Et.Prénom, " ",Moy)  
    **Lire**(fr,Et)  
**Fin Tant que**  
**Fermer**(fr)

**Fin**

*{Programme principal}*

**Début**

**Création**(Fn)  
**Affichage**(Fn)  
**Résultat**(Fn,Fr)  
**Affichage**(Fr)

**Fin**



Faculté des Sciences Juridiques, Economiques  
et de Gestion de Jendouba

Année Universitaire : 2008/2009 – Semestre 2

Module : Algorithmique et structures de données II

Classe : 1<sup>ère</sup> année LFIAG

Chargé de cours : Riadh IMED FEREH

Chargé de TD : Riadh BOUSLIMI

## TD n° 2 (Récursivité)

### Objectifs

- Résoudre des programmes récursifs.
- Comprendre la démarche de transformation d'un programme itérative en un programme récursive.
- Savoir les avantages de l'utilisation de la récursivité pour résoudre les problèmes.

### EXERCICE N°1

Écrire une fonction récursive qui retourne la somme des chiffres d'un entier N donné.

*Exemple :* ( 123 == > 1 + 2 + 3 = 6 )

### EXERCICE N°2

Écrire une fonction récursive qui calcul la factorielle d'un entier N positif.

*Exemple :* ( 5 ! = 5 x 4 x 3 x 2 x 1 = 120 )

### EXERCICE N°3

Écrire une fonction récursive qui permet de déterminer si un entier N saisi au clavier est premier ou pas. (Un nombre premier n'est divisible que par 1 ou lui-même).

### EXERCICE N°4

Écrire une procédure récursive qui permet d'inverser une chaîne de caractères sans utiliser une chaîne temporaire.

*Exemple :* information → noitamrofni

### EXERCICE N°5

Écrire une fonction récursive qui permet de vérifier si deux chaînes s1 et s2 sont anagrammes ou non.

s1 et s2 sont anagrammes s'ils se composent de même lettre.

*Exemple :* s1 = "chien" ; s2 = "niche" → vrai

### EXERCICE N°6

Écrire une fonction récursive qui permet de vérifier si un mot placé en paramètre est palindrome ou non.

*Exemples :* mot = "aziza" → vrai ; mot = "alga" → faux

### EXERCICE N°7

- Écrire une fonction récursive nommée **Rech\_seq** qui permet de chercher un entier x dans un tableau T de n entiers selon le principe de la recherche séquentielle.
- Écrire une fonction récursive nommée **Rech\_dico** qui permet de chercher un entier x dans un tableau T de n entiers selon le principe de la recherche dichotomique.

### EXERCICE N°8

Écrire une procédure récursive indirecte nommé **Tri\_Bulle** qui permet de trier un tableau T de n entiers. Utiliser les deux procédures ci-dessous :

- **Procédure Permuter**(var x, y : Entier)
- **Procédure Parcours** (i,n :Entier ; var rep : Booléen ; var T :TAB)

NB : **rep** elle est utilisée pour renvoyé s'il y'a eu une permutation au cours du parcours du tableau.

### EXERCICE N°9

Écrire une procédure récursive nommée **Anagramme** qui permet d'afficher tous les anagramme d'une chaine ch.

NB : Utiliser une permutation circulaire pour résoudre ce problème.

**Exemple** :ch="iag"

Les anagrammes de « iag » sont :

- 1) aig
- 2) agi
- 3) gai
- 4) gia
- 5) iga
- 6) iag

### EXERCICE N°10

Écrire un programme récursif permettant de dessiner une pyramide d'étoiles selon un entier n donné, avec n un nombre impair.

**Exemple** : n=9

```
      *
     ***
    *****
   ********
  *********
 ***
*****
*****
*****
*****
*****
*****
*****
*****
*****
```

## Correction du TD n°2

### EXERCICE N°1

```
Fonction Somme(n :Entier) : Entier
Var
    s : Entier
Début
    Si n>0 Alors
        s ← s + n MOD 10
        Somme ← Somme(n DIV 10)
    Fin Si
    Somme ← s
Fin
```

### EXERCICE N°2

```
Fonction Factorielle (n :Entier) : Entier
Var
    fac : Entier
Début
    Si n>0 Alors
        fac ← fac + n
        Factorielle ← Factorielle(n - 1)
    Fin Si
    Factorielle ← fac
Fin
```

### EXERCICE N°3

```
Fonction Premier (d, n :Entier) : Entier
Début
    Si d ≤ ((N DIV 2)+1) Alors
        Si n MOD d ≠ 0 Alors
            Premier ← Premier(d+1, n)
        Sinon
            Premier ← vrai
        Fin Si
    Sinon
        Premier ← faux
    Fin Si
Fin
```

#### EXERCICE N°4

```
Procédure Inverse (var ch : chaîne)
Var
c : caractère
Début
    Si ch = "" Alors                                //Si la chaîne ch est vide on s'arrête
        Inverse ← ""
    Sinon
        c ← ch[LONG(ch)]                            // récupération du dernier caractère
        Effacer(ch, long(ch), 1)                     // Effacer le dernier caractère
        Inverse(ch)                                  // Inverser la nouvelle chaîne ch
        ch ← c + ch                                  // Concaténation
    Fin Si
Fin
```

#### EXERCICE N°5

```
Fonction Anagramme (var s1, s2 : chaîne) : Booléen
Var
c : caractère
Début
    Si LONG(s1) ≠ LONG(s2) Alors
        Anagramme ← faux
    Sinon
        Si LONG(s1) = LONG(s2) = 0 Alors
            Anagramme ← vrai
        Sinon
            p = POS(s1[1], s2) // retourne la position du 1er caractère
                               // dans la chaîne de caractère s2
            Si p = 0 Alors     // si non trouvé
                Anagramme ← faux
            Sinon
                EFFACER(s1, 1, 1) //Effacer le 1er caractère de s1
                EFFACER(s2, p, 1) //Effacer le pème caractère de s2
                Anagramme ← Anagramme(s1, s2)
            Fin Si
        Fin Si
    Fin Si
Fin
```



## EXERCICE N°6

```
Function Palindrome (mot : chaîne) : Booléen
Var
c : caractère
Début
    Si mot="" Alors                                     //Si la chaîne est vide on s'arrête
        Palindrome ← Vrai
    Sinon
        Si mot[1] = mot[LONG(mot)] Alors
            EFFACER(mot,1,1)                             //Effacer le premier caractère
            EFFACER(mot, LONG(mot), 1)                   //Effacer le dernier caractère
            Palindrome ← Palindrome(mot)
        Sinon
            Palindrome ← faux
        Fin Si
    Fin Si
Fin
```

## EXERCICE N°7

```
Function Rech_seq(i, n, x : Entier ; T : TAB) : Booléen
Début
    Si i ≤ n Alors
        Si T[i] = x Alors
            Rech_seq ← Vrai
        Sinon
            Rech_seq ← Rech_seq(i+1, n, x, T)
        Fin Si
    Sinon
        Rech_seq ← Faux
    Fin Si
Fin
```

```
Function Rech_dico(g, d, x : Entier ; T : TAB) : Booléen
Var
    m : Entier
Début
    Si g > d Alors
        Rech_dico ← faux
    Sinon
        m ← (d + g) DIV 2
        Si T[m] = x Alors
            Rech_dico ← Vrai
        Sinon Si T[m] > x Alors
            Rech_dico ← Rech_dico(g, m-1, x, T)
        Sinon
            Rech_dico ← Rech_dico(m+1, d, x, T)
        Fin Si
    Fin Si
Fin
```

## EXERCICE N°8

```
Procédure Permuter(var x, y : Entier)
Var
    Aux : Entier
Début
    aux ← x
    x ← y
    y ← aux
Fin

Procédure Parcours(i,n :Entier ; var rep : Booléen ; var T :TAB)
Début
    Si i<n Alors
        Si T[i] >T[i+1] Alors
            Permuter(T[i],T[i+1])
            rep← Vrai
        Fin Si
    Fin Si
Fin

Procédure TriBulle(n : Entier ; Perm : Booléen var T :TAB)
Var
    Perm : Booléen
Début
    //S'il n'a pas eu de permutation et n>1 on recommence
    //le parcours si non on arrête le traitement
    Si ((n>1) ET (Perm = vrai)) Alors
        Perm← Faux
        Parcours(1,n,Perm,T)
        TriBulle(n-1,Perm,T)
    Fin Si
Fin
```

## EXERCICE N°9

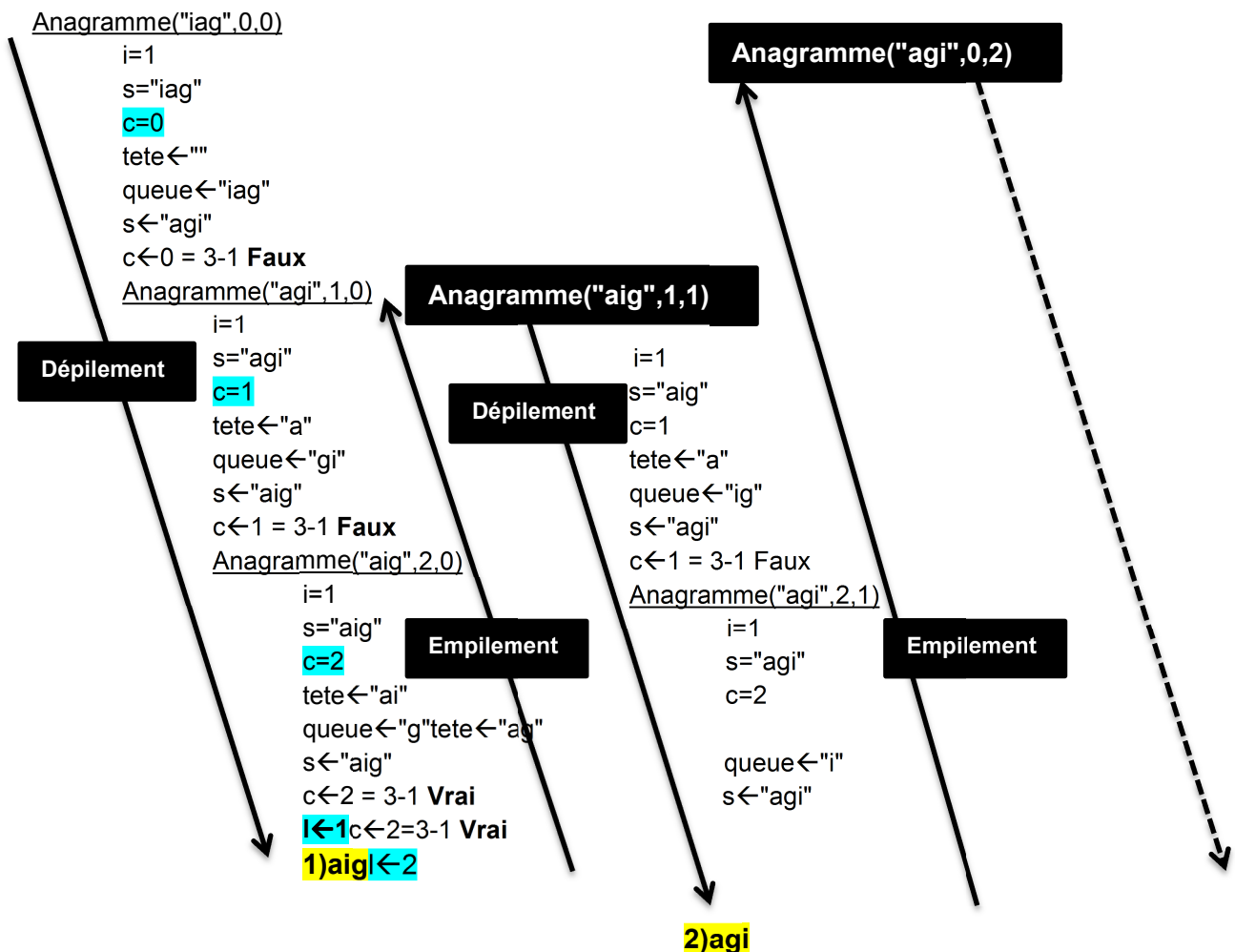
```
Procédure PermutCirc(var ch :chaine)
Début
    //Si la longueur de la chaine est plus qu'un caractère
    Si LONG(ch)>1 Alors
        //concaténation du dernier caractère avec le reste de la chaine
        Ch ← ch[LONG(ch)] + SOUS-CHAINE(ch,2, LONG(ch)-1)
    Fin Si
Fin
```

```

Procédure Anagramme(s : chaîne ; c : Entier ; var l : Entier)
Var
    i : Entier
    tete, queue : chaîne
Début
    Pour i de 1 à LONG(s) - c
        tete ← SOUS-CHAÎNE(s, 1, c)
        queue ← SOUS-CHAÎNE(s, c+1, LONG(s)-c)
        s = tete + PermutCirc(queue)
        Si c = LONG(s) - 1 Alors
            l ← l + 1
            Ecrire(l, " ", s)
        Sinon
            Anagramme(s, c + 1, l)
        Fin Si
    Fin Pour
Fin

```

### Trace d'exécution



## EXERCICE N°10

### Algorithme Pyramide

**Var**

n : Entier                      *//variable globale*

**Fonction Saisie() : Entier**

**Var**

m : Entier

**Début**

Ecrire("Entrer un nombre impair :")

Lire(m)

**Si** m MOD 2=0 **Alors**

**Saisie**←**Saisie**()

**Sinon**

**Saisie**← m

**Fin Si**

**Fin**

**Procédure Espace(i :Entier)**

**Début**

**Si** i>=1 **Alors**

        Ecrire(" ")

**Etoile**(i-1)            *//appel récursive*

**Fin Si**

**Fin**

**Procédure Etoile(j :Entier)**

**Début**

**Si** j>=1 **Alors**

        Ecrire("\*")

**Etoile**(j-1)            *//appel récursive*

**Fin Si**

**Fin**

**Procédure Dessiner(k , m :Entier)**

**Début**

**Si** k<=m **Alors**

**Espace**(m-k)            *// écriture des espaces au début de la ligne*

**Etoile**(k)                *// écriture des étoiles au début de la ligne*

**Etoile**(k-1)            *// écriture des étoiles à la fin de la ligne*

        Ecrire("\n")            *// retour à la ligne*

**Fin Si**

**Fin**

*{Programme principal}*

**Début**

    n ←**Saisie**()                *//appel de la fonction récursive qui permet de saisir n*

**Dessiner**(1,n)              *// appel de la procédure récursive qui va dessiner le pyramide*

**Fin**



Faculté des Sciences Juridiques, Economiques  
et de Gestion de Jendouba

Année Universitaire : 2008/2009 – Semestre 2

Module : Algorithmique et structures de données II

Classe : 1<sup>ère</sup> année LFIAG

Chargé de cours : Riadh IMED FEREH

Chargé de TD : Riadh BOUSLIMI

## TD n° 3 (Les Listes chaînées)

### Objectifs :

- *Savoir déclarer, construire des listes chaînées.*
- *Manipuler, traiter et apprendre à utiliser des listes chaînées.*
- *Distinguer entre les différents types de listes chaînées.*

### QUESTIONS DE COURS

- 1) *Qu'est-ce qu'un pointeur ?*
- 2) *Quelle est la différence entre une structure de données statique et une structure de données dynamique ?*

### EXERCICE N°1 (RAPPEL DU COURS)

- 1) Définir une liste simple chaînée composé d'une valeur entière et d'un pointeur vers l'élément suivant ;
- 2) Déclarer une variable de ce type défini dans la question 1) ;
- 3) Écrire une procédure permettant de créer une liste chaînée de n entiers.
  - **Procédure CréerListe(n : Entier ; var L : Liste)**
- 4) Écrire deux procédures l'une itérative et l'autre récursive permettant d'afficher les éléments de la liste.
  - **Procédure AffichageIter(L : Liste)**
  - **Procédure AffichageRecu(L : Liste)**
- 5) Écrire une fonction récursive qui permet de rechercher un élément x dans la liste.
  - **Fonction Recherche(x : Entier ; L : Liste) : Booléen**
- 6) Écrire une procédure qui permet d'ajouter une tête de la liste.
  - **Procédure AjouterTete(x : Entier ; var L : Liste)**
- 7) Écrire une procédure qui supprimer un élément de la liste.
  - **Procédure Supprimer(x : Entier ; var L : Liste)**

### EXERCICE N°2

Écrire une procédure nommée **Inverse** qui permet d'inverser une liste sans utiliser un variable temporaire.

### EXERCICE N°3

Écrire une procédure nommée **TriBulle** qui permet de trier une liste chaînée selon le principe de tri à bulle.

### EXERCICE N°4

Écrire une procédure qui permet de concaténer deux listes chaînées L1 et L2 d'entiers dans une troisième liste L3. Il faut traiter toutes les contraintes possibles.

### EXERCICE N°5

On dispose de deux listes L1 et L2 triés qui sont triés dans l'ordre croissant. Écrire une procédure **fusion** qui permet de fusionner deux listes L1 et L2 dans la liste L1.

### EXERCICE N°6

Écrire une procédure qui permet de supprimer les doublons dans une liste chaînée triée dans l'ordre croissant qui contient des caractères alphabétiques. Comment peut-on éliminer les doublons si la liste n'était pas triée ?

### EXERCICE N°7

Écrire une fonction qui permet de vérifier si une liste est palindrome ou non.

Exemple :



→ Cette liste est palindrome



→ Cette liste n'est pas palindrome

### EXERCICE N°8

Une liste doublement chaînée est une liste qui admet, en plus de permettre l'accès au suivant d'un élément, permet l'accès au précédent d'un élément.

- Quel est l'intérêt de ce type de liste par rapport aux listes simplement chaînées ?
- Écrivez les fonctions et procédures suivantes en mettant éventuellement à jour les primitives précédentes :
  1. **Fonction premier(L : Liste) : Liste** renvoie le premier élément de L.
  2. **Fonction dernier(L : Liste) : Liste** renvoie le dernier élément de L. Proposer deux solutions l'une en connaissant la queue de la liste et la deuxième en ne connaissant pas cette dernière.
  3. **Fonction estVide(L : Liste) : Booléen** renvoie vrai si la liste est vide et faux sinon.
  4. **Procédure supprimerPremier(var L : Liste)** supprime le premier élément de L.
  5. **Procédure ajouterAprès(P, Q : Liste ; var L : Liste)** ajoute P dans L après Q.
  6. **Procédure supprimer(x : Entier ; var L : Liste)** supprime x dans L.

### EXERCICE N°9

1. Écrire la procédure **AjouterTete** qui permet d'ajouter au début d'une liste circulaire L un entier e.
2. Écrire la procédure **AjouterFin** qui permet d'ajouter à la fin d'une liste circulaire L un entier e.
3. Écrire une procédure **Affiche** qui affiche la liste circulaire qui lui est passée en argument.

## Correction du TD n°3

### QUESTIONS DE COURS

1) Qu'est-ce qu'un pointeur ?

**R :** Un **pointeur** *P* est une variable statique dont les valeurs sont des adresses.

2) Quelle est la différence entre une structure de données statique et une structure de données dynamique ?

**R :** La différence entre une structure de données statique et une structure de données dynamique est la taille, la première est définie au début en spécifiant la taille qui est fixe, par contre pour la deuxième la taille est variable et qui est basée sur l'allocation dynamique.

### EXERCICE N°1 (RAPPEL DU COURS)

1)

**Type**

```
Liste : ^cellule  
  
Cellule : Enregistrement  
|  
|   val  : Entier  
|  
|   suiv : Liste  
|  
Fin Cellule
```

2)

**Var**

```
L : Liste
```

3)

**Procédure CréerListe(n : Entier ; var L : Liste)**

**Var**

```
Tete, p : Liste  
i : Entier
```

**Début**

```
    Allouer(Tete)  
    Ecrire("Entrer élément Tête :")  
    Lire(Tete^.val)  
    Tete^.suiv ← nil  
    L ← Tete  
  
    Pour i de 2 à n faire  
    |  
    |   Allouer(p)  
    |   Ecrire("Entrer élément n° :", i)  
    |   Lire(p^.val)  
    |   p^.suiv ← nil  
    |   L^.suiv ← p  
    |   L ← p  
    Fin Pour
```

**Fin Pour**

```
L ← Tete //se pointer sur la tête de la liste
```

**Fin**

4)

```

Procédure AffichageIter(L : Liste)
Var
    p : Liste
Début
    p ← L
    Tant que p ≠ nil Faire
        Ecrire(p^.val)
        p ← p^.suiv
    Fin Tant que
Fin

```

```

Procédure AffichageRecu(L : Liste)
Début
    Si L ≠ nil Alors
        Ecrire(p^.val)
        AffichageRecu(p^.suiv)
    Fin Si
Fin

```

5)

**Version itérative**

```

Fonction Recherche(x : Entier ; L : Liste) : Booléen
Var
    p : Liste
Début
    p ← L
    Tant que ((p ≠ nil) ET (p^.val ≠ x)) Faire
        p ← p^.suiv
    Fin Tant que
    Si p = nil Alors
        Recherche ← Faux
    Sinon
        Recherche ← Vrai
    Fin Si
Fin

```

**Version récursive**

```

Fonction Recherche(x : Entier ; L : Liste) : Booléen
Début
    Si L = nil Alors
        Recherche ← Faux
    Sinon
        Si L^.val = x Alors
            Recherche ← Vrai
        Sinon
            Recherche ← Recherche(x, L^.suiv)
        Fin Si
    Fin Si
Fin

```



6)

```

Procédure AjouterTete(x : Entier ; var L : Liste)
Var
    Tete : Liste
Début
    Allouer(Tete)
    Tete^.val ← x
    Tete^.suiv ← L
    L ← Tete
Fin

```

7)

**Version itérative**

```

Procédure Supprimer(x : Entier ; var L : Liste)
Var
    P, Q : Liste
Début
    Si L = nil Alors
        Ecrire("Liste vide, impossible de supprimer ", x)
    Sinon
        Si L^.val = x Alors
            P ← L //mémorisation de l'élément à supprimer
            L ← L^.suiv
        Sinon
            P ← L^.suiv
            Tant que ((P ≠ nil) ET (P^.val ≠ x)) Faire
                Q ← P
                P ← P^.suiv
            Fin Tant que
            Si P ≠ nil Alors
                Q^.suiv ← P^.suiv
                Libérer(P)
            Fin Si
        Fin Si
    Fin Si
Fin

```

**Version récursive**

```

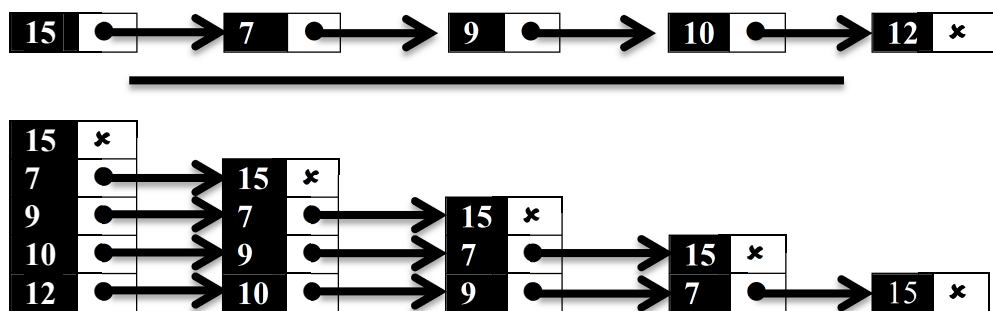
Procédure Supprimer(x : Entier ; var L : Liste)
Var
    P : Liste
Début
    Si L ≠ nil Alors
        Si L^.val = x Alors
            P ← L
            L ← L^.suiv
            Libérer(P)
        Sinon
            Supprimer(x, L^.Suiv)
        Fin Si
    Fin Si
Fin

```

## EXERCICE N°2

```
Procédure Inverse (var L : Liste)
Var
  P, Q : Liste
Début
  P ← nil
  Tant que L ≠ nil Faire
    Q ← L^.suiv
    L^.suiv ← P
    P ← L
    L ← Q
  Fin Tant que
Fin
```

Trace d'exécution de l'exemple ci-dessous



## EXERCICE N°3

```
Procédure TriBulle(var L : Liste)
Var
  P, Q : Liste
  Temp : Entier
Début
  P ← L
  Q ← L^.suiv
  Répéter
    rep ← faux
    P ← L // pointeur sr le premier élément
    Q ← L^.suiv // pointeur sur le deuxième élément
    Tant que Q ≠ nil Faire
      Si P^.val > Q^.val Alors
        Temp ← P^.val
        P^.val ← Q^.val
        Q^.val ← Temp
        rep ← vrai
      Fin Si
    Fin tant que
  Jusqu'à rep = faux
Fin
```

Permutation

#### EXERCICE N°4

```
Procédure Concatener(L1,L2 : Liste ; var L3 : Liste)
Début
    Si L1 = nil ET L2 = nil Alors
        L3 ← nil
    Sinon Si L1 ≠ nil ET L2 = nil Alors
        L3 ← L1
    Sinon Si L1 = nil ET L2 ≠ nil Alors
        L3 ← L2
    Sinon
        //affectation de la première liste L1
        L3 ← L1
        //se pointer sur le dernier élément de L3
        Tant que L3^.suiv ≠ nil Faire
            L3 ← L3^.suiv
        Fin Tant que
        //affectation de la liste L2 à la fin de L3
        L3^.suiv ← L2
        //se pointer sur la tête de la liste
        L3 ← L1
    Fin Si
Fin
```

#### EXERCICE N°5

```
Procédure Fusion(var L1 :Liste ; L2 : Liste)
Var
Tete, P, Q : Liste
Début
    Si L1 = nil Alors //Si L1 est vide
        L1 ← L2
    Sinon
        Si L2 ≠ nil Alors
            //Fixation de la position de la tête de la liste
            Si L1^.val ≤ L2^.val Alors
                Tete ← L1
            Sinon
                Tete ← L2
            Fin Si
            Q ← Tete //mémorisation de l'élément précédent
            Tant que L1 ≠ nil ET L2 ≠ nil Faire
                Si L1^.val > L2^.val Alors
                    P ← L2
                    L2 ← L2^.suiv
                    Q^.suiv ← P
                    P^.suiv ← L1
                    Q ← P
```

```

Sinon Si L1^.val > L2^.val Alors
    P ← L2
    L2 ← L2^.suiv
    Q ← L1
    L1 ← L1^.suiv
    Q^.suiv ← P
    P^.suiv ← L1
Sinon
    Q ← L1 //mémorisation de l'élément précédent
    L1 ← L1^.suiv
Fin Si
Fin Tant que
L1 ← Tete //pointer L1 à la tête de la liste
Fin Si
Fin Si
Fin

```

### EXERCICE N°6

Liste  
triée

```

Procédure SupprimeDoublons(var L :Liste)
Var
Q , P : Liste
Début
    Q ← L
    Si L ≠ nilAlors
        Tant que Q^.suiv ≠ nilFaire
            P ← Q
            Q ← Q^.suiv
            Si Q^.val = P^.valAlors
                //on supprime de L l'élément pointé par Q.
                Q^.suiv ← P^.suiv
                Libérer(P)
            Fin Si
        Fin Tant que
    Fin Si
Fin

```

Liste  
non  
triée

```

Procédure SupprimeDoublons(var L :Liste)
Var
    Q : Liste
    Vpred : caractère
Début
    Q ← L
    Si L ≠ nilAlors
        Tant que Q^.suiv ≠ nilFaire
            Vpred ← Q^.val
            Q ← Q^.suiv
            Si Q^.val = Vpred Alors
                Supprimer(Vpred, Q)
            Fin Si
        Fin Tant que
    Fin Si
Fin

```

Cette procédure  
se charge de  
parcourir la liste  
Q en supprimant  
toutes les valeurs  
de Vpred

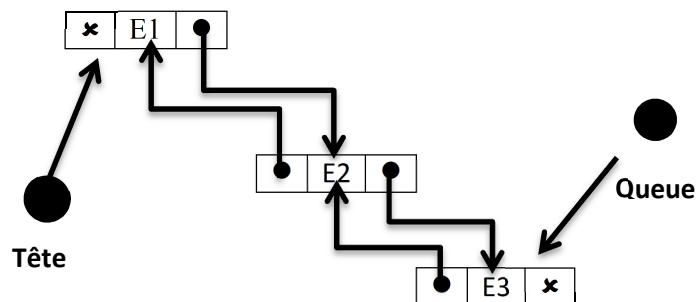
## EXERCICE N°7

```
Fonction Palindrome(L : Liste) :Booléen
Var
    Stop : Booléen
    Deb, Fin, Q : Liste
Début
    Deb ← L           // Pointeur sur le premier élément
    Fin ← nil          // Pointeur sur le dernier élément
    Stop ← faux        // Signale l'interruption du traitement
    Tant que Deb^.suiv ≠ Fin ET NON(Stop) Faire
        // Récupérer le pointeur sur le dernier élément de la liste
        Q ← P
        Tant que Q^.suiv ≠ Fin Faire
            Q ← Q^.suiv
        Fin Tant que

        Si P^.val = Q^.val Alors
            Fin ← Q //se pointer sur le dernier élément de la liste L
        Sinon
            Stop ← vrai //arrêt du traitement
        Fin Si
        Si Deb ≠ Fin Alors
            Deb ← Deb ^.suiv
        Fin Si
    Fin Tant que
    Palindrome ← NON(Stop)
Fin
```

## EXERCICE N°8

- L'intérêt d'une liste doublement chaînée par rapport à une liste chaînée simple c'est pour accélérer la recherche d'un élément.



```
Type
    Liste : ^cellule
    Cellule : Enregistrement
        |
        |   pred : Liste
        |   val  : Entier
        |   suiv : Liste
        |
        |___ Fin Cellule
Var
    L : Liste
```

1)

```

Fonction Premier(L : Liste) : Liste
Début
    Si L ≠ null Alors
        Premier ← L
    Sinon
        Premier ← nil
    Fin Si
Fin

```

2)

*Solution n° 1 : On connait la queue de la liste*

```

Fonction Dernier(Queue : Liste) : Liste
Début
    Dernier ← Queue
Fin

```

*Solution n° 2 : On ne connait pas la queue de la liste*

```

Fonction Dernier(L : Liste) : Liste
Var
    P : Liste
Début
    P ← L
    Tant que P^.suiv ≠ null Faire
        P ← P^.suiv
    Fin Tant que
    Dernier ← P
Fin

```

3)

```

Fonction estVide(L : Liste) : Booléen
Début
    Si L = nil Alors
        estVide ← vrai
    Sinon
        estVide ← faux
    Fin Si
Fin

```

4)

```

Procédure supprimerPremier(var L : Liste)
Var
    P : Liste
Début
    Si L ≠ nil Alors
        P ← L
        L ← L^.suiv
        L^.pred ← nil
        Libérer(P)
    Fin Si
Fin

```

5)

```

Procédure ajouterAprès(P,Q : Liste ; var L : Liste)
Var
    D :Liste
Début
    D ← L
    Tant que D ≠ Q ET D ≠ nil Faire
        D ← D^.suiv
    Fin Tant que
    Si D = Q Alors
        D ← D^.suiv
        D^.pred ← p
        P^.suiv ← D
        Q^.suiv ← P
        P^.pred ← Q
    Sinon
        Ecrire("Ajout impossible, élément non existant")
    Fin Si
Fin

```

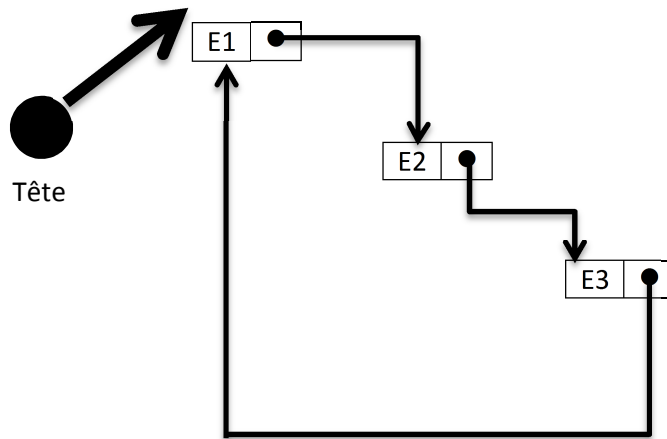
6)

```

Procédure Supprimer(x : Entier ; var L : Liste)
Var
    P,Q,D : Liste
Début
    Si L = nil Alors
        Ecrire("Liste vide, impossible de supprimer ",x)
    Sinon
        Si L^.val = x Alors
            P ← L
            L ← L^.suiv
        Sinon
            P ← L^.suiv
            Tant que ((P ≠ nil) ET (P^.val ≠ x)) Faire
                Q ← P
                P ← P^.suiv
            Fin Tant que
            Si P ≠ nil Alors
                D ← P^.suiv
                D^.pred ← Q
                Q^.suiv ← D
                Libérer(P)
            Fin Si
        Fin Si
    Fin Si
Fin

```

## EXERCICE N°9



*Une liste circulaire est une structure de données dynamique dont le dernier élément de la liste pointe sur la tête de la liste.*

```
Procédure AjouterTête(e : Entier ; var L : Liste)
```

```
Var
```

```
    N,Tete,Queue: Liste
```

```
Début
```

```
    Allouer(N)
```

```
    N^.val ← e
```

```
    Si L = nil Alors
```

```
        L ← N
```

```
        N^.suiv ← L
```

```
        L^.suiv ← N
```

```
    Sinon
```

```
        Tete ← L
```

```
        Queue ← L
```

```
        Tant que Queue^.suiv ≠ Tete Faire
```

```
            Queue ← Queue^.suiv
```

```
        Fin Tant que
```

```
        Queue^.suiv ← N
```

```
        N^.suiv ← Tete
```

```
        Tete ← N
```

```
        L ← Tete
```

```
    Fin Si
```

```
Fin
```

**Remarque :** ajouter au début ou à la fin d'une liste chaînée circulaire, c'est d'appliquer le même algorithme que celui d'ajouter tête de liste.



```

Procédure affiche(L : Liste)
Var
    P :Liste
Début
    Si L = nil Alors
        Ecrire("Liste vide")
    Sinon
        P ← L//Tête de la liste
        // Tant que la Queue de la liste est différente de la Tête de la liste
        Tant que P^.suiv ≠ P Faire
            Ecrire(P^.val)
            P ← P^.suiv
        Fin Tant que
    Fin Si
Fin

```



Faculté des Sciences Juridiques, Economiques  
et de Gestion de Jendouba

Année Universitaire : 2008/2009 – Semestre 2

Module : Algorithmique et structures de données II

Classe : 1<sup>ère</sup> année LFIAG

Chargé de cours : Riadh IMED FEREH

Chargé de TD : Riadh BOUSLIMI

## TD n° 4 (Les piles et les files)

### Objectifs

- Définir et manipuler une pile et une file
- Savoir différencier entre la structure d'une pile et la structure de celle d'une file.

### EXERCICE N°1(LES PILES)

- 1) Rappeler la définition d'une pile et donner les contraintes d'accès à cette dernière.
- 2) Créer la structure d'une pile.
- 3) Définir une variable quelconque de type pile.
- 4) Écrire la procédure **InitialiserPile**(var **P : Pile**) qui permet de créer une pile vide.
- 5) Écrire la fonction **EstPileVide**(**P : Pile**) :**Booléen** qui permet de vérifier si une pile est vide.
- 6) Écrire la procédure **Empiler**(**x : Entier ; var P : Pile**) qui permet d'ajouter l'élément *x* au sommet de la pile.
- 7) Écrire la procédure **Dépiler**(var **x : Entier ; var P : Pile**) qui permet de supprimer le sommet de la pile et de le mettre la valeur dans la variable *x*.

### EXERCICE N°2(LES FILES)

- 1) Qu'est-ce qu'une file. Présenter un graphe illustrant l'accès à cette dernière.
- 2) Créer la structure d'une file.
- 3) Définir une variable quelconque de type file.
- 4) Écrire la procédure **InitialiserFile**(var **F : File**) qui permet de créer une file vide.
- 5) Écrire la fonction **EstFileVide**(**F : File**) qui permet de vérifier si une file est vide.
- 6) Écrire la procédure **Enfiler**(**x : Entier ; var F : File**) qui permet d'ajouter l'élément *x* au sommet de la file.
- 7) Écrire la procédure **Défiler**(var **x : Entier ; var F : File**) qui permet de supprimer le sommet de la file et de le mettre la valeur dans la variable *x*.

### EXERCICE N°1(LES PILES)

- 1) Une pile est une structure de données dynamique (liste chaînée) dont l'insertion et la suppression d'un élément se font toujours en tête de liste.

On peut résumer les contraintes d'accès par le principe « dernier arrivé, premier sorti » qui se traduit en anglais par : *Last In First Out*.

2)

```
Type  
  Pile : ^cellule  
  cellule : Enregistrement  
    |  
    val : Entier  
    suiv : Pile  
Fin cellule
```

3)

```
Var  
  P : Pile
```

4)

```
Procédure InitialiserPile(var P : Pile)  
Début  
  |  
  P ← nil  
Fin
```

5)

```
Fonction EstPileVide(P : Pile) : Booléen  
Début  
  |  
  Si P = nil Alors  
    |  
    EstPileVide ← vrai  
  Sinon  
    |  
    EstPileVide ← faux  
  Fin Si  
Fin
```

6)

```
Procédure Empiler(x : Entier ; var P : Pile)  
Var  
  Nv : Pile  
Début  
  |  
  Allouer(Nv)  
  Nv^.val ← x  
  Nv^.suiv ← P  
  P ← Nv  
Fin
```

7)

```
Procédure Dépiler(var x : Entier ; var P : Pile)  
Var  
  S : Pile  
Début  
  |  
  Si NON(EstPileVide(P)) Alors  
    |  
    S ← P  
    x ← P^.val  
    P ← P^.suiv  
    Libérer(S)  
  Fin Si  
Fin
```

## EXERCICE N°2(LES FILES)

1) Une file est une structure de données dynamique (liste chaînée) dont les contraintes d'accès sont définies comme suit :

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

On peut résumer les contraintes d'accès par le principe « premier arrivé, premier sorti » qui se traduit en anglais par : **Fast In First Out** .

2)

### **Type**

```
Liste : ^cellule
cellule : Enregistrement
|       val  : Entier
|       suiv : Liste
Fin cellule

File : Enregistrement
|    Tête  : Liste
|    Queue : Liste
Fin File
```

3)

### **Var**

```
F : File
```

4)

### **Procédure InitialiserFile(var F : File)**

#### **Début**

```
F.Tête ← nil
F.Queue ← nil
```

#### **Fin**

5)

### **Fonction EstFileVide(F : File) : Booléen**

#### **Début**

```
Si F.Tête = nil Alors
    EstPileVide ← vrai
Sinon
    EstPileVide ← faux
Fin Si
```

#### **Fin**

6)

### **Fonction Enfiler(x : Entier ; var F : File)**

#### **var**

```
Nv : Liste
```

#### **Début**

```
Allouer(Nv)
Nv^.val ← x
Nv^.suiv ← nil
Si F.Queue ≠ nil Alors
    F.Queue^.suiv ← Nv
Sinon
    F.Tête ← Nv
Fin Si
F.Queue ← Nv
```

#### **Fin**

7)

```
Procédure Défiler(var B : Arbre ; var F : File)
var
    P : Liste
Début
    Si F.Tête ≠ nil Alors
        P ← F.Tête
        B ← P^.val
        F.Tête ← F.Tête^.suiv
        Libérer(P)
    Fin Si
    Si F.Tête = nil Alors
        F.Queue ← nil
    Fin Si
Fin
```



Faculté des Sciences Juridiques, Economiques  
et de Gestion de Jendouba

Année Universitaire : 2008/2009 – Semestre 2

Module : Algorithmique et structures de données II

Classe : 1<sup>ère</sup> année LFIAG

Chargé de cours : Riadh IMED FEREH

Chargé de TD : Riadh BOUSLIMI

## TD n° 5 (Arbre binaire de recherche)

### Objectifs

- Définir, créer et manipuler un arbre binaire de recherche.
- Savoir comment parcourir un arbre binaire de recherche ?
- Trouver un élément dans un arbre.
- Savoir passer d'une structure dynamique d'un arbre vers une structure statique représentée par un vecteur.
- Savoir fusionner deux arbres.
- Connaître les différents types d'arbres : dégénéré, complet ou parfait.

### EXERCICE N°1(RAPPEL)

- 1) Définir un arbre binaire de recherche ?
- 2) Créer la structure d'un arbre qui admet comme arguments une valeur entière, un pointeur vers le fils gauche et un pointeur vers le fils droit ;
- 3) Déclarer une variable de cette structure 2) ;
- 4) Écrire une procédure **CréerElement**(*x : Entier ; var B : Arbre*) qui permet de créer un élément *x* dans l'arbre.
- 5) Dessiner un arbre qui contient les valeurs qui suit : 14, 8, 4, 9, 3, 1, 20, 30, 25, 50.
- 6) Écrire une fonction **EstVide**(*B : Arbre*) : **Booléen** qui renvoie vrai si l'arbre est vide et faux si non.
- 7) Écrire une fonction **EstUneFeuille**(*B : Arbre*) : **Booléen** qui renvoie vrai si le sommet de l'arbre n'admet aucun fils et faux si non.
- 8) Écrire une fonction **Recherche**(*x : Entier ; B : Arbre*) : **Booléen** qui permet de chercher un élément *x* dans l'arbre et de renvoyer vrai si ce dernier est existant et faux si non.
- 9) Écrire et dessiner les graphes illustrant les différents types de parcours en profondeur vues en cours.
- 10) Écrire une procédure **SupprimerElement**(*x : Entier ; var B : Arbre*) qui permet de supprimer un élément *x* de l'arbre.

### EXERCICE N°2(LES MESURES)

- 1) Écrire une fonction qui calcule la taille d'un arbre binaire.
- 2) Écrire une fonction qui calcule la hauteur d'un arbre binaire.
- 3) Écrire une fonction qui calcule le nombre de nœuds externes (feuilles) d'un arbre binaire.
- 4) Écrire une fonction qui calcule le nombre de nœuds internes d'un arbre binaire.
- 5) Écrire une fonction qui calcule la longueur de cheminement externe d'un arbre binaire.

### EXERCICE N°3(PARCOURS EN LARGEUR)

On souhaite parcourir et calculer la largeur d'un arbre. Pour cela nous allons implémenter les sous-programmes suivants :

- 1) Écrire une procédure **InitialiserFile**(var *F* : File) qui permet d'initialiser la file d'attente.
- 2) Écrire une fonction **FileEstVide**(*F* : File) : Booléen qui vérifie si la file d'attente est vide. Elle renvoie vrai si c'est le cas et faux sinon.
- 3) Écrire une procédure **Enfiler**(*B* : Arbre ; var *F* : File) qui permet d'ajouter un arbre à la file d'attente.
- 4) Écrire une procédure **Défiler**(var *B* : Arbre ; var *F* : File) qui permet de renvoyer et supprimer le premier élément de la file d'attente.
- 5) Écrire une procédure **ParcoursEnLargeur**(*B* : Arbre ; var *n* :Entier) qui permet de parcourir l'arbre en largeur et de renvoyer la largeur de ce dernier.

### EXERCICE N°4 (DU DYNAMIQUE VERS LE STATIQUE)

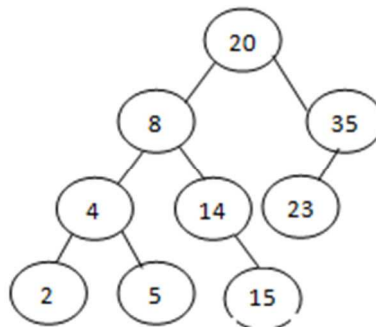


Figure 1 – Arbre pour représentation hiérarchique

Écrire une procédure qui construit un vecteur contenant les éléments d'un arbre binaire. Pour indiquer un arbre vide, il suffit de la représenter par le symbole  $\emptyset$ .

Par exemple, ci-dessous la représentation de l'arbre de la figure 1 :

1	2	3	4	5	6	7	8	9	10	11	12	13
22	8	35	4	14	23	$\emptyset$	2	5	$\emptyset$	15	$\emptyset$	$\emptyset$

### EXERCICE N°5 (ROTATIONS)

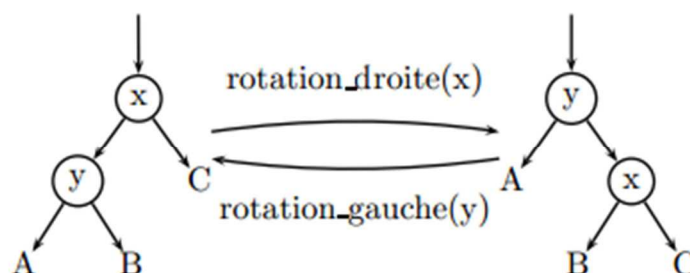


Figure 2 : Rotations gauche et droite

La figure ci-dessus montre un exemple de rotation de à droite et une rotation à gauche. On souhaite dans ce cas créer les deux procédures qui suivent :

- 1) Procédure *rotation\_droite*( var  $B$  : *Arbre*) qui effectue une rotation à droite et renvoie le nouvel arbre.
- 2) Procédure *rotation\_gauche*(var  $B$  : *Arbre*) qui effectue une rotation à gauche et renvoie le nouvel arbre.

### EXERCICE N°6 (COPIE)

Écrire une procédure qui permet de copier un arbre binaire A dans un deuxième arbre B.

### EXERCICE N°7 (FUSION)

Écrire une procédure qui permet de fusionner deux arbres binaires A et B, et de renvoyer un arbre C qui contient les deux arbres. Discuter les différents cas possibles.

### EXERCICE N°8 (DEGENERE, PARFAIT OU COMPLET)

On dispose des deux fonctions hauteur et taille. On souhaite écrire des fonctions qui permettent de vérifier si un arbre :

1. est *dégénéré* :
  - Un arbre *dégénéré* est un arbre dont tous les nœuds internes sont des points simples.
  - L'arbre  $B$  est dégénéré si  $\text{taille}(B) = \text{hauteur}(B) + 1$ .
  - a) Écrire la première solution en utilisant les deux fonctions taille et hauteur.
  - b) Écrire à nouveau sans utiliser les deux fonctions taille et hauteur.
2. est *complet* :
  - Un arbre dont tous les niveaux sont remplis est *complet*.
  - L'arbre  $B$  est complet si  $\text{taille}(B) = 2^{\text{hauteur}(B) + 1} - 1$ .
  - a) Écrire la première solution en utilisant la fonction hauteur.
  - b) Écrire la deuxième solution sans utiliser la fonction hauteur.
3. est *parfait* :
  - Un arbre est *parfait* si tous ses niveaux sont remplis, sauf le dernier dans lequel les feuilles sont rangées le plus à gauche possible.Écrire une fonction permettant de vérifier si un arbre est parfait.

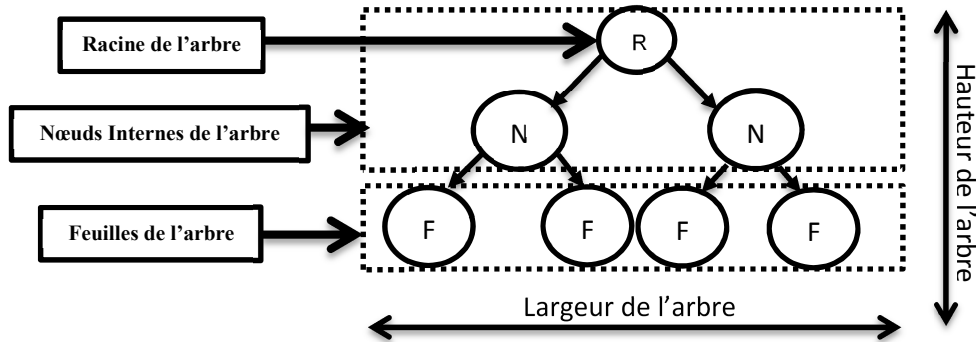


## Correction du TD n°5

### EXERCICE N°1 (RAPPEL DU COURS)

1) Qu'est-ce qu'un arbre binaire de recherche ?

**R** : Un **arbre binaire** B est un ensemble de nœuds qui est soit vide, soit composé d'une racine et de deux arbres binaires disjoints appelés **sous-arbre droit** et **sous-arbre gauche**.



- Un **nœud interne** est un sommet qui a au moins un fils (gauche ou droit ou les deux).
- Une **feuille** est un sommet qui n'a pas de fils.
- La **hauteur d'un sommet x** est la longueur (en nombre d'arcs) du plus long chemin de x à une feuille.
- La **hauteur d'un arbre** est égale à la hauteur de la racine.

2)

#### Type

```
Arbre : ^Nœud
Nœud : Enregistrement
    Val : Entier
    FilsG : Arbre
    FilsD : Arbre
Fin Nœud
```

3)

#### Var

```
A : Arbre
```

4)

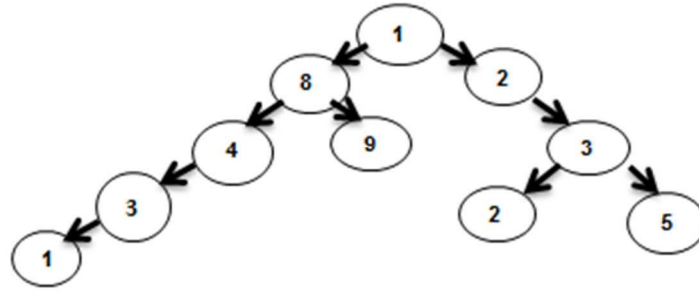
#### Procédure CréerElement(x : Entier ; var B : Arbre)

##### Début

```
Si B = nil Alors
    Allouer(B)
    B^.val ← x
    B^.FilsG ← nil
    B^.FilsD ← nil
Sinon
    Si B^.val > x Alors
        CréerElement(x, B^.FilsG)
    Fin Si
    Si B^.val < x Alors
        CréerElement(x, B^.FilsD)
    Fin Si
Fin Si
```

##### Fin

5)



1)

```

Fonction EstVide(B :Arbre) : Booléen
Début
    Si B = nil Alors
        EstVide ← faux
    Sinon
        EstVide ← vrai
    Fin Si
Fin
  
```

2)

```

Fonction EstUnFeuille(B :Arbre) : Booléen
Début
    Si B^.FilsG = nil ET B^.FilsD = nil Alors
        EstUnFeuille ← vrai
    Sinon
        EstUnFeuille ← faux
    Fin Si
Fin
  
```

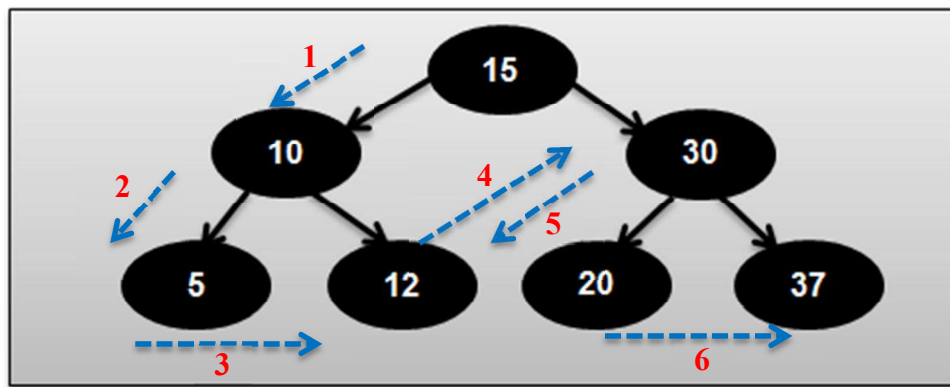
3)

```

Fonction Recherche(x : Entier ; B : Arbre) : Booléen
Début
    Si B=nil Alors
        Recherche ← faux
    Sinon
        Si B^.val = x Alors
            Recherche ← vrai
        Sinon Si B^.val > x Alors
            Recherche ← Recherche(x, B^.FilsG)
        Sinon
            Recherche ← Recherche(x, B^.FilsD)
        Fin Si
    Fin Si
Fin
  
```

4)

1. La première stratégie de parcours d'un arbre binaire de recherche est dite « en profondeur d'abord » ou dans l'ordre préfixé.



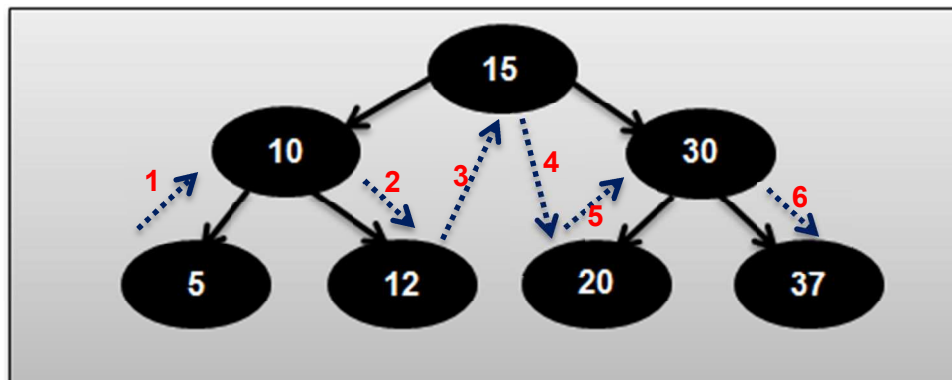
Résultat : 15 10 5 12 30 20 37

```

Procédure ParcoursPréfixe(B : Arbre)
Début
    Si B ≠ nilAlors
        Ecrire(B^.val)
        ParcoursPréfixe(B^.FilsG)
        ParcoursPréfixe(B^.FilsD)
    Fin Si
Fin

```

2. La deuxième stratégie de parcours d'un arbre binaire de recherche est dite : «parcours de l'arbre dans l'ordre **infixe** ou symétrique ». Le parcours donne des valeurs triées dans l'ordre croissant.



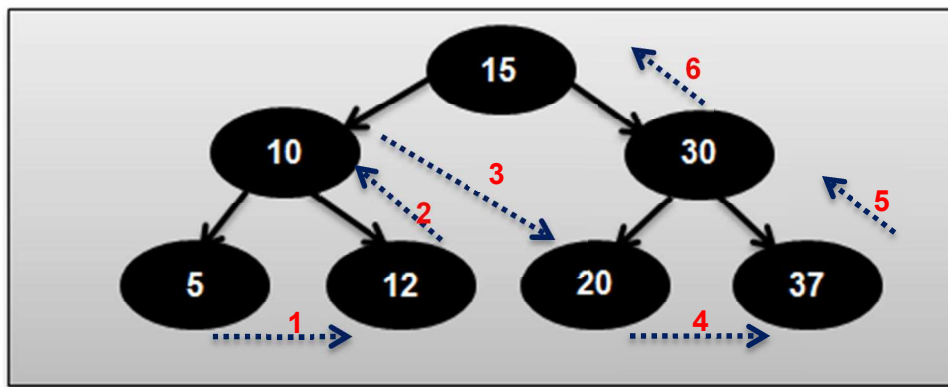
Résultat : 5 10 12 15 20 30 37

```

Procédure ParcoursInfixé(B : Arbre)
Début
    Si B ≠ nilAlors
        ParcoursInfixé(B^.FilsG)
        Ecrire(B^.val)
        ParcoursInfixé(B^.FilsD)
    Fin Si
Fin

```

3. La troisième stratégie de parcours d'un arbre binaire de recherche est dite : «parcours de l'arbre dans l'ordre **postfixé** ».



Résultat : 5 10 12 20 37 30 15

```

Procédure ParcoursPostfixé(B : Arbre)
Début
  Si B ≠ nil Alors
    ParcoursPostfixé(B^.FilsG)
    ParcoursPostfixé(B^.FilsD)
    Ecrire(B^.val)
  Fin Si
Fin
  
```

5) Le principe de suppression doit obéir aux constatations suivantes :

La suppression commence par la recherche de l'élément. Une fois trouvé ce dernier :

- si c'est une feuille, on la retire sans problèmes
  - si c'est un sommet qui n'a qu'un fils, on le remplace par ce fils
  - si c'est un sommet qui a deux fils, on a deux solutions :
    1. le remplacer par le sommet de plus grande valeur dans le sous arbre gauche.
    2. le remplacer par le sommet de plus petite valeur dans le sous arbre droit.
- ⇒ Pour simplifier le travail nous allons commencer par écrire deux fonctions : la première renvoie l'élément qui a la plus grande valeur dans le sous arbre gauche ; la deuxième renvoie l'élément qui a la plus petite valeur dans le sous arbre droit.

```

Fonction PlusPetitSousArbreDroit(B : Arbre)
Début
  Si B^.FilsG ≠ nil Alors
    PlusPetitSousArbreDroit ← PlusPetitSousArbreDroit(B^.FilsG)
  Sinon
    PlusPetitSousArbreDroit ← B
  Fin Si
Fin
  
```

```

Fonction PlusGrandSousArbreGauche(B : Arbre)
Début
  Si B^.FilsD ≠ nil Alors
    PlusGrandSousArbreGauche ← PlusGrandSousArbreGauche(B^.FilsD)
  Sinon
    PlusGrandSousArbreGauche ← B
  Fin Si
Fin
  
```

```

Procédure Supprimer(x : Entier ; var B : Arbre)
Var
    P, Q : Arbre
Début
    Si B = nil Alors
        Ecrire ("Arbre vide ", x, " est introuvable")
    Sinon
        Si B^.val = x Alors
            Si B^.FilsG = nil ET B^.FilsD = nil Alors
                // Si c'est une feuille
                Libérer(B)
            Sinon Si B^.FilsG ≠ nil ET B^.FilsD = nil Alors
                // Si le sommet admet un sous arbre gauche
                B ← B^.FilsG
            Sinon Si B^.FilsG = nil ET B^.FilsD ≠ nil Alors
                // Si le sommet admet un sous arbre gauche
                B ← B^.FilsD
            Sinon Si B^.FilsG ≠ nil ET B^.FilsD ≠ nil Alors
                // Si le sommet admet deux fils
                // On cherche le plus petit ou le plus grand
                P ← PlusPetitSousArbreDroit(B^.FilsD)
                // ou aussi on peut chercher le plus grand
                // P ← PlusGrandSousArbreGauche(B^.FilsG)
                Q ← P
                Q^.FilsG ← B^.FilsG
                Q^.FilsD ← B^.FilsD
                Libérer(P)
                B ← Q
            Fin Si
        Sinon Si B^.val > x Alors
            Supprimer(x, B^.FilsG)
        Sinon
            Supprimer(x, B^.FilsD)
        Fin Si
    Fin Si
Fin

```

## EXERCICE N°2(LES MESURES)

1)

```

Fonction Taille (B : Arbre) : Entier
Début
    Si B = nil Alors
        Taille ← 0
    Sinon
        Taille ← 1 + Taille(B^.FilsG) + Taille(B^.FilsD)
    Fin Si
Fin

```

2)

```

Fonction Max(x,y :Entier) : Entier
Début
    Si x>y Alors
        Max ← x
    Sinon
        Max ← y
    Fin Si
Fin

Fonction Hauteur(B : Arbre) : Entier
Début
    Si B = nil Alors
        Hauteur ← 0
    Sinon
        Hauteur ← 1 + Max(Hauteur(B^.FilsG),Hauteur(B^.FilsD))
    Fin Si
Fin

```

3)

```

Fonction NombreDeNoeudsExternes(B : Arbre) : Entier
Début
    Si B = nil Alors
        NombreDeNoeudsExternes ← 0
    Sinon
        Si EstUneFeuille(B) Alors
            NombreDeNoeudsExternes ← 1
        Sinon
            NombreDeNoeudsExternes ←
                NombreDeNoeudsExternes(B^.FilsG) +
                NombreDeNoeudsExternes(B^.FilsD)
        Fin Si
    Fin Si
Fin

```

4)

```

Fonction NombreDeNoeudsInternes(B : Arbre) : Entier
Début
    Si B = nil Alors
        NombreDeNoeudsInternes ← 0
    Sinon
        Si EstUneFeuille(B) Alors
            NombreDeNoeudsInternes ← 0
        Sinon
            NombreDeNoeudsInternes ← 1 +
                NombreDeNoeudsInternes(B^.FilsG) +
                NombreDeNoeudsInternes(B^.FilsD)
        Fin Si
    Fin Si
Fin

```

5)

**Spécification** : on additionne les profondeurs des feuilles de  $B$  (non vide),  $prof$  étant la profondeur de la racine de  $B$ .

```

Fonction LongueurCheminArbre( $B$  : Arbre ;  $prof$  : Entier)
Début
    Si  $B = \text{nil}$  Alors
        LongueurCheminArbre  $\leftarrow 0$ 
    Sinon
        Si  $B^{\wedge}.\text{FilsG} = B^{\wedge}.\text{FilsD}$  Alors
            LongueurCheminArbre  $\leftarrow prof$ 
        Sinon
            LongueurCheminArbre  $\leftarrow$ 
                LongueurCheminArbre( $B^{\wedge}.\text{FilsG}, prof+1$ ) +
                LongueurCheminArbre( $B^{\wedge}.\text{FilsD}, prof+1$ )
        Fin Si
    Fin Si
Fin

```

### EXERCICE N°3(PARCOURS EN LARGEUR)

*Avant d'entamer l'implémentation de la procédure de parcours en largeur nous devons initialement définir les différentes structures nécessaires.*

```

Type

Arbre : ^nœud
nœud : Enregistrement
    FilsG : Arbre
    val : Entier
    FilsD : Arbre
Fin nœud

Liste : ^cellule
cellule : Enregistrement
    val : Arbre
    suiv : Liste
Fin cellule

File : Enregistrement
    Tête : Liste
    Queue : Liste
Fin File

```

1)

```

Procédure InitialiserFile( $\text{var } F$  : File)
Début
     $F.\text{Tête} \leftarrow \text{nil}$ 
     $F.\text{Queue} \leftarrow \text{nil}$ 
Fin

```

2)

```

Fonction FileEstVide(F : File) : Booléen
Début
    Si F.Tête = nil Alors
        FileEstVide ← vrai
    Sinon
        FileEstVide ← faux
    Fin Si
Fin

```

3)

```

Procédure Enfiler(B : Arbre ; var F : File)
Var
    P :Liste
Début
    Allouer(P)
    P^.val ← B
    P^.suiv ← nil
    Si F.Queue ≠ nil Alors
        F.Queue^.suiv ← P
    Sinon
        F.Tête ← P
    Fin Si
    F.Queue ← P
Fin

```

4)

```

Procédure Défiler(var B : Arbre ; var F : File)
Var
    P : Liste
Début
    Si F.Tête ≠ nil Alors
        P ← F.Tête
        B ← P^.val
        F.Tête ← F.Tête^.suiv
        Libérer(P)
    Fin Si
    Si F.Tête = nil Alors
        F.Queue ← nil
    Fin Si
Fin

```



5)

```

Procédure ParcourEnLargeur(B : Arbre ; var larg :Entier)
Var
    F : File
    Larg_max : Entier

Début
    Si B = nil Alors
        Larg  $\leftarrow$  0
        Ecrire("Arbre vide")
    Sinon
        InitialiserFile(F)
        Enfiler(B, F)
        larg_max  $\leftarrow$  0
        Tant que NON(FileEstVide(F)) Faire
            Défiler(B, F)
            Si B = nil Alors
                Si larg > larg_max Alors
                    larg_max  $\leftarrow$  larg
                Fin Si
                Si NON(FileEstVide(F)) Alors
                    larg  $\leftarrow$  0
                    Enfiler(x,F)
                Fin Si
            Sinon
                larg  $\leftarrow$  larg + 1
                Ecrire(B^.val)

                Si B^.FilsG  $\neq$  nil Alors
                    Enfiler(B^.FilsG, F)
                Fin Si

                Si B^.FilsD  $\neq$  nil Alors
                    Enfiler(B^.FilsD, F)
                Fin Si
            Fin si
        Fin Tant que
        larg  $\leftarrow$  larg_max
    Fin Si
Fin

```

#### EXERCICE N°4 (DU DYNAMIQUE VERS LE STATIQUE)

```

Procédure ConstruireVecteur(B :Arbre ; var n :Entier ; var T :TAB)
Début
    Si B = nil Alors
        T[i]  $\leftarrow$   $\emptyset$ 
    Sinon
        T[i]  $\leftarrow$  B^.val
        ConstruireVecteur(B^.FilsG, 2*n, T)
        ConstruireVecteur(B^.FilsD, 2*n+1, T)
    Fin Si
Fin

```

### EXERCICE N°5 (ROTATIONS)

```
Procédure rotation_droite(var B : Arbre)
Var
    Temp : Arbre
Début
    Temp ← B^.FilsG
    B^.FilsG ← B^.FilsD
    B^.FilsD ← Temp
    B ← Temp
Fin
```

```
Procédure rotation_gauche(var B : Arbre)
Var
    Temp : Arbre
Début
    Temp ← B^.FilsD
    B^.FilsD ← B^.FilsG
    B^.FilsG ← Temp
    B ← Temp
Fin
```

### EXERCICE N°6 (COPIE)

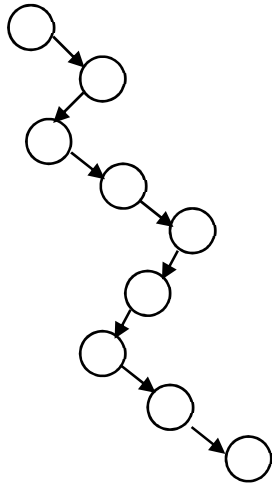
```
Procédure Copier(A : Arbre ; var B : Arbre)
Début
    Si A ≠ nil Alors
        CréerElement(A^.val, B)
        Copier(A^.FilsG, B)
        Copier(A^.FilsD, B)
    Fin Si
Fin
```

### EXERCICE N°7 (FUSION)

```
Procédure Fusion(A,B: Arbre ; var C : Arbre)
Début
    Si A ≠ nil ET B = nil Alors
        Copier(A,C)
    Sinon Si A = nil ET B ≠ nil Alors
        Copier(A,C)
    Sinon
        Si A ≠ nil ET B ≠ nil Alors
            Si A^.val > b^.val Alors
                CréerElement(A^.val,C)
                Fusion(A^.FilsG,B,C)
                Fusion(A^.FilsD,B,C)
            Sinon
                CréerElement(A^.val,C)
                Fusion(A,B^.FilsG,C)
                Fusion(A,B^.FilsD,C)
            Fin Si
        Fin Si
    Fin Si
Fin
```

### EXERCICE N°8 (DEGENERÉ, PARFAIT OU COMPLET)

1. Un arbre *dégénéré* est un arbre dont tous les nœuds internes sont des points simples.  
L'arbre  $B$  est dégénéré si  $\text{taille}(B) = \text{hauteur}(B) + 1$ .



**Figure 1 : Exemple d'arbre dégénéré**

La figure 1 montre qu'un arbre dégénéré admet pour chaque sommet un seul fils.

#### **1<sup>ère</sup> solution : sans utiliser les mesures : taille et hauteur**

**Fonction EstDegenere(B : Arbre) : Booléen**

**Début**

```
    Si B = nil Alors
        EstDegenere ← vrai
    Sinon
        Si (B^.FilsG ≠ nil) ET (B^.FilsD ≠ nil) Alors
            EstDegenere ← faux
        Sinon
            Si B^.FilsG = nil Alors
                EstDegenere ← EstDegenere(B^.FilsD)
            Sinon
                EstDegenere ← EstDegenere(B^.FilsG)
            Fin Si
        Fin Si
    Fin Si
Fin
```

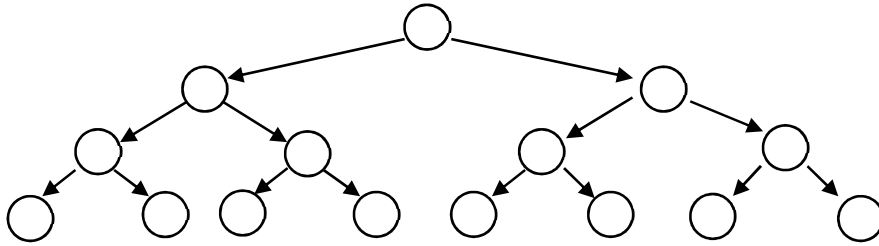
#### **2<sup>ème</sup> solution : avec les mesures : taille et hauteur**

**Fonction EstDegenere(B : Arbre) : Booléen**

**Début**

```
    Si (Taille(B) = Hauteur(B)+1) Alors
        EstDegenere ← vrai
    Sinon
        EstDegenere ← faux
    Fin Si
Fin
```

2. On appelle arbre binaire **complet** un arbre binaire tel que chaque sommet possède 0 ou 2 fils. L'arbre B est complet si  $taille(B) = 2^{hauteur(B)+1} - 1$ .



**Figure 2 : Exemple d'arbre complet**

La figure 2 montre qu'un arbre complet admet pour chaque sommet zéro fils ou deux fils.

#### 1<sup>ère</sup> solution : avec la mesure de la hauteur

Fonction EstComplet(B : Arbre, h :Entier) : Booléen

Début

Si B = nil Alors

EstComplet ← (h=-1)

Sinon

EstComplet ← EstComplet(B^.FilsG, h-1)

ET

EstComplet(B^.FilsD, h-1))

Fin Si

Fin

#### Appel de la fonction

Complet : Booléen

Complet ← EstComplet(B,Hauteur(B))

#### 2<sup>ème</sup> solution : sans utiliser la mesure de l'hauteur

La solution proposée consiste à effectuer un parcours en largeur

Fonction EstComplet(B : Arbre) : Booléen

Var

F : File

Larg, larg\_prochain : Entier

Début

Si B = nil Alors

EstComplet ← vrai

Sinon

Initialiser(F)

Enfiler(B, F)

Larg ← 0

larg\_prochain ← 1

Tant que NON(EstVide(F)) Faire

Défiler(B,F)

Si B = nil Alors

Si larg ≠ larg\_prochain Alors

ViderFile(F)

EstComplet ←faux

Fin Si

Si NON(EstVide(F)) Alors

larg\_prochain ← 2 \* larg

larg←0

Enfiler(nil, F)

Fin si

```

Sinon
    Larg ← Larg + 1
    Si B^.FilsG ≠ nil Alors
        Enfiler(B^.FilsG, F)
    Fin Si
    Si B^.FilsD ≠ nil Alors
        Enfiler(B^.FilsD, F)
    Fin Si
Fin si
Fin Tant que
EstComplet ← vrai
Fin si
Fin

```

3. On appelle arbre binaire *parfait* un arbre binaire (complet) tel que chaque sommet est le père de **deux** sous-arbres de **même hauteur (figure3)**.  
 Un arbre binaire parfait possède  $2^{h+1}-1$  sommets, où  $h$  est la hauteur de l'arbre.

```

Fonction EstParfait(B : Arbre) : Booléen
Var
    F : File
    fils_vide, parfait : Booléen
Début
    Si B = nil Alors
        EstParfait ← vrai
    Sinon
        Initialiser(F)
        Enfiler (B, F)
        fils_vide ← faux
        parfait ← faux

        Tant que NON(EstVide(F) ET NON(fils_vide) Faire
            Défiler(B,F)
            Si B^.FilsG = nil Alors
                fils_vide ← vrai
                parfait ← (B^.FilsG = nil)
            Sinon
                Enfiler (B^.FilsG, F)
                Si B^.FilsD ≠ nil Alors
                    Enfiler (B^.FilsD, F)
                Sinon
                    fils_vide ← vrai
                Fin Si
            Fin Si
        Fin Tant que

        Tant que NON(EstVide(F)) ET parfait Faire
            Défiler(B,F)
            parfait ← (B^.FilsG = B^.FilsD)
        Fin tant que

        Initialiser(F)
        EstParfait ← parfait
    Fin si
Fin

```

## **B**IBLIOGRAPHIE

- S. ROHAUT : Algorithmique et Techniques fondamentale de programmation, Edition Eni 2007.
- LIGNELET P., Algorithmique. Méthodes et modèles, Paris : Masson, 1985.
- [www.intelligentedu.com/blogs/post/free\\_computer\\_books/3760/the-algorithm-design-manual/fr/](http://www.intelligentedu.com/blogs/post/free_computer_books/3760/the-algorithm-design-manual/fr/)